Sample Code

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.

Download

macOS 13.3+ | Xcode 14.3+

Overview

The vImage library provides the high-level convert-any-to-any `vImageConverter` class to convert image data between Core Video and Core Graphics formats. The convert-any-to-any functionality is suited for apps that work across different platforms where AVFoundation may provide video frames in different formats.

This sample code app uses AVFoundation to access the Mac camera and vImage to convert the camera image to an RGB image that the app displays onscreen.

Specify the pixel format

To ensure that AVCapture doesn't have to perform a conversion from the capture format to the output format, the sample code specifies the output format as the camera's active format. After declaring `videoOutput` as an `AVCaptureVideoDataOutput` instance, the following code defines the output pixel format by creating the `videoSettings` dictionary:

```
pixelFormat = CMFormatDescriptionGetMediaSubType(camera.activeFormat.formatDescripti
videoOutput.videoSettings = [kCVPixelBufferPixelFormatTypeKey as String: pixelFormat
```

Lock the Core Video pixel buffer

When the app starts the flow of data through the capture pipeline, AVFoundation calls `captureOutput(_:didOutput:from:)` for each new video frame. The following code locks the `CVPixelBuffer` structure's underlying memory to make it available exclusively to the vImage conversion function:

```
CVPixelBufferLockBaseAddress(
    pixelBuffer,
    CVPixelBufferLockFlags.readOnly)

do {
    try convertVideoFormatToRGB(cvPixelBuffer: pixelBuffer)
} catch {
    fatalError("Unable to perform conversion.")
}

CVPixelBufferUnlockBaseAddress(
    pixelBuffer,
    CVPixelBufferLockFlags.readOnly)
```

Create a Core Video-to-Core Graphics converter

The vImage convert-any-to-any function requires a converter that describes the source and destination formats. The sample code app converts a Core Video pixel buffer to a Core Graphics image. The code calls the `make(buffer:)` function to derive the source Core Video image format from the `CVPixelBuffer`. In some cases, the `vImageCVImageFormat` instance that the make function returns may have incomplete information. The following code ensures that the format has a color space and chrominance siting information:

```
guard let cvImageFormat = vImageCVImageFormat.make(buffer: cvPixelBuffer) else {
    fatalError("Unable to derive Core Video pixel format from buffer.")
}

if cvImageFormat.colorSpace == nil {
    cvImageFormat.colorSpace = CGColorSpaceCreateDeviceRGB()
}

if cvImageFormat.chromaSiting == nil {
    cvImageFormat.chromaSiting = .center
}
```

The sample app specifies a three-channel, 8-bit-per-channel `vImage_CGImageFormat` as the conversion destination format.

```
let cgImageFormat = vImage_CGImageFormat(
    bitsPerComponent: 8,
    bitsPerPixel: 8 * 3,
    colorSpace: CGColorSpaceCreateDeviceRGB(),
    bitmapInfo: CGBitmapInfo(rawValue: CGImageAlphaInfo.none.rawValue))!
```

The `make(sourceFormat:destinationFormat:flags:)` type method creates a `vImageConverter` instance from the source and destination formats.

```
converter = try? vImageConverter.make(sourceFormat: cvImageFormat,
                                      destinationFormat: cgImageFormat)

if converter == nil {
    fatalError("Unable to create Core Video to Core Graphics converter.")
}
```

Initialize the destination buffer

The destination pixel buffer contains the RGB image after conversion. The code defines it as a three-channel, 8-bit-per-channel `vImage.PixelBuffer` structure.

```
var destinationBuffer: vImage.PixelBuffer<vImage.Interleaved8x3>!
```

The first time that the app calls the conversion function, it runs the following code to initialize the destination pixel buffer with the same dimensions as the Core Video pixel buffer:

```
let size = vImage.Size(cvPixelBuffer: cvPixelBuffer)
destinationBuffer = vImage.PixelBuffer<vImage.Interleaved8x3>(size: size)
```

Initialize the source buffers

Although the sample code app knows that the Core Graphics image format requires only a single buffer at compile time, the camera's active format defines the number of source buffers and their pixel formats at runtime. Therefore, the code defines the source buffers as an array of `vImage.DynamicPixelFormat` pixel buffers.

```
var sourceBuffers: [vImage.PixelBuffer<vImage.DynamicPixelFormat>]!
```

The `vImageConverter` provides the `makeCVToCGPixelBuffers(referencing:)` function that returns an array of pixel buffers. These pixel buffers reference the underlying memory of each plane of the Core Video pixel buffer.

```
sourceBuffers = try converter.makeCVToCGPixelBuffers(referencing: cvPixelBuffer)
```

Convert the Core Video buffer contents to a Core Graphics format image

The `convert(from:to:)` function accepts the source and destination pixel buffers and converts the Core Video pixel buffer's contents to a Core Graphics image.

```
try converter.convert(from: sourceBuffers, to: [destinationBuffer])
```

Create an output Core Graphics image

Finally, the code calls `makeCGImage(cgImageFormat:)` to create a Core Graphics image that it displays in the user interface.

```
let rgbImage = destinationBuffer.makeCGImage(cgImageFormat: cgImageFormat)!
```

See Also

Core Video Interoperation

{} Using vImage pixel buffers to generate video effects

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

{} 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.

{} 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.