Sample Code

Generating an animation with a Core Image Render Destination

Animate a filtered image to a Metal view in a SwiftUI app using a Core Image Render Destination.

Download

iOS 15.5+ | iPadOS 15.5+ | macOS 12.0+ | Xcode 14.0+

Overview

This sample shows how to assemble a SwiftUI app that displays a Metal view with animated images that you generate procedurally from Core Image.

To accomplish this, the sample sets up a Scene in a WindowGroup with a single content view. The sample's `ContentView` adopts the View protocol and initializes a `Renderer` using a closure to vend a CIImage. It then adds a `MetalView`, with the instantiated `Renderer`, to the content body.

The sample combines view update and state changes to produce the animation:

- For view update, the `MetalView` structure conforms to the UIViewRepresentable or NSViewRepresentable protocol of the SwiftUI life cycle.
- For state changes, the Renderer is a `StateObject` conforming to the ObservableObject protocol.

Generate an animation

The `Renderer` class generates an image for an animation frame by conforming to the MetalKit MTKViewDelegate delegate protocol. The protocol's draw(in:) function commits render destination work to the GPU using a render task in a Metal command buffer.

For more information about drawing with MetalKit see Using a Render Pipeline to Render Primitives.

MetalKit calls the `draw(in:)` delegate function of the `Renderer` automatically.

```
final class Renderer: NSObject, MTKViewDelegate, ObservableObject {
```

An image-supplying function parameterized by both timestamp and scale factor initializes the `Renderer`. This function combines checkerboard and hue-adjustment filters to generate animated checkerboard pattern images cropped to a fixed size.

```
// Create a Metal view with its own renderer.
let renderer = Renderer(imageProvider: { (time: CFTimeInterval, scaleFactor: CGFloat

    var image: CIImage

    // Animate a shifting red and yellow checkerboard pattern.
    let pointsShiftPerSecond = 25.0
    let checkerFilter = CIFilter.checkerboardGenerator()
    checkerFilter.width = 20.0 * Float(scaleFactor)
    checkerFilter.color0 = CIColor.red
    checkerFilter.color1 = CIColor.yellow
    checkerFilter.center = CGPoint(x: time * pointsShiftPerSecond, y: time * pointsS
    image = checkerFilter.outputImage ?? CIImage.empty()

    // Animate the hue of the image with time.
    let colorFilter = CIFilter.hueAdjust()
    colorFilter.inputImage = image
    colorFilter.angle = Float(time)
    image = colorFilter.outputImage ?? CIImage.empty()
```

After the sample initializes the `Renderer`, the `Renderer` makes a command buffer and gets the currentDrawable.

```
if let commandBuffer = commandQueue.makeCommandBuffer() {

    // Add a completion handler that signals `inFlightSemaphore` when Metal and the
    // finished processing the commands that the app encoded for this frame.
    // This completion indicates that Metal and the GPU no longer need the dynamic b
    // Core Image writes to in this frame.
    // Therefore, the CPU can overwrite the buffer contents without corrupting any r
```

```
    let semaphore = inFlightSemaphore
    commandBuffer.addCompletedHandler { (_ commandBuffer)-> Swift.Void in
        semaphore.signal()
    }

    if let drawable = view.currentDrawable {
```

The `Renderer` then configures a CIRenderDestination with the command buffer, `currentDrawable`, dimensions, and pixel format, along with a closure that returns the texture for the `currentDrawable`.

```
// Create a destination the Core Image context uses to render to the drawable's Meta
let destination = CIRenderDestination(width: Int(dSize.width),
                                      height: Int(dSize.height),
                                      pixelFormat: view.colorPixelFormat,
                                      commandBuffer: commandBuffer,
                                      mtlTextureProvider: { () -> MTLTexture in
    // Core Image calls the texture provider block lazily when starting a task to re
    return drawable.texture
})
```

The sample uses the render destination to create an animation frame at a specific timestamp.

Finally, the sample composites the render destination's centered image on a background and submits work to the GPU to render and present the result.

```
// Create a displayable image for the current time.
let time = CFTimeInterval(CFAbsoluteTimeGetCurrent() - self.startTime)
var image = self.imageProvider(time, contentScaleFactor, headroom)

// Center the image in the view's visible area.
let iRect = image.extent
let backBounds = CGRect(x: 0, y: 0, width: dSize.width, height: dSize.height)
let shiftX = round((backBounds.size.width + iRect.origin.x - iRect.size.width) * 0.5
let shiftY = round((backBounds.size.height + iRect.origin.y - iRect.size.height) * 0
image = image.transformed(by: CGAffineTransform(translationX: shiftX, y: shiftY))

// Blend the image over an opaque background image.
// This is needed if the image is smaller than the view, or if it has transparent pi
image = image.composited(over: self.opaqueBackground)

// Start a task that renders to the texture destination.
```

```
_ = try? self.cicontext.startTask(toRender: image, from: backBounds,
                                  to: destination, at: CGPoint.zero)

// Insert a command to present the drawable when the buffer has been scheduled for e
commandBuffer.present(drawable)

// Commit the command buffer so that the GPU executes the work that the Core Image P
```

Add an EDR effect

The sample adds an EDR effect, a shiny ripple with a bright specular highlight, to the rendered checkerboard animation in three steps:

1. Opt into EDR support for the view and set an accommodating color space and pixel format.
2. Query the EDR headroom for each frame and pass `headroom` to the image provider closure for the `Renderer`.
3. Set the peak specular highlight value to the maximum value of white with respect to the current headroom, or a reasonable default value.

For more information about adding an EDR effect, see Display EDR content with Core Image, Metal, and SwiftUI.

Configure the view for EDR support

The `MetalView` opts into EDR support setting wantsExtendedDynamicRangeContent to true on the backing CAMetalLayer. When enabled, the layer uses a wide gamut colorspace to render colors beyond SDR range. Similarly, the MTKView sets a wide gamut colorPixelFormat to render the generated EDR image.

```
if let layer = view.layer as? CAMetalLayer {
    // Enable EDR with a color space that supports values greater than SDR.
    if #available(iOS 16.0, *) {
        layer.wantsExtendedDynamicRangeContent = true
    }
    layer.colorspace = CGColorSpace(name: CGColorSpace.extendedLinearDisplayP3)
    // Ensure the render view supports pixel values in EDR.
    view.colorPixelFormat = MTLPixelFormat.rgba16Float
}
```

Query EDR headroom

The `Renderer` queries the current EDR headroom for each draw call using either maximumPotentialExtendedDynamicRangeColorComponentValue (NSScreen) or currentEDRHeadroom (UIScreen). If EDR headroom is unavailable the sample sets `headroom` to `1.0` clamping to SDR.

```
                // Determine EDR headroom and fallback to SDR, as needed.
                // Note: The headroom must be determined every frame to include char
                let screen = view.window?.screen
#if os(iOS)
                var headroom = CGFloat(1.0)
                if #available(iOS 16.0, *) {
                    headroom = screen?.currentEDRHeadroom ?? 1.0
                }
#else
                let headroom = screen?.maximumExtendedDynamicRangeColorComponentValu
#endif
```

Leverage EDR headroom

The sample's ripple effect takes a gradient shadingImage to shade the contor of the ripple so that it appears to reflect light from the upper-left corner. CILinearGradient generates the gradient shading image between the current maximum RGB white, color0, and a fully transparent clear color, color1.

```
// Compute a shading image for the ripple effect below.
// Cast light on the upper-left corner of the shading gradient image.
let angle = 135.0 * (.pi / 180.0)
let gradient = CIFilter.linearGradient()
// Create a bright white color for a specular highlight with the current
// maximum possible pixel component values within headroom
// or a reasonable alternative.
let maxRGB = min(headroom, 8.0)
gradient.color0 = CIColor(red: maxRGB, green: maxRGB, blue: maxRGB,
                          colorSpace: CGColorSpace(name: CGColorSpace.extendedLinear
gradient.color1 = CIColor.clear
gradient.point0 = CGPoint(x: sin(angle) * 90.0 + 100.0,
                          y: cos(angle) * 90.0 + 100.0)
gradient.point1 = CGPoint(x: sin(angle) * 85.0 + 100.0,
                          y: cos(angle) * 85.0 + 100.0)
```

```
let shading = gradient.outputImage?.cropped(to: CGRect(x: 0, y: 0,
                                                       width: 200, height: 200))

// Add a shiny ripple effect to the image.
let ripple = CIFilter.rippleTransition()
ripple.inputImage = image
ripple.targetImage = image
ripple.center = CGPoint(x: 256.0 * scaleFactor,
                        y: 192.0 * scaleFactor)
ripple.time = Float(fmod(time * 0.25, 1.0))
ripple.shadingImage = shading
image = ripple.outputImage ?? CIImage()

return image.cropped(to: CGRect(x: 0, y: 0,
                                width: 512.0 * scaleFactor,
                                height: 384.0 * scaleFactor))
```

See Also

Custom Render Destination

`class` **`CIRenderDestination`**

A specification for configuring all attributes of a render task's destination and issuing asynchronous render tasks.

`class` **`CIRenderInfo`**

An encapsulation of a render task's timing, passes, and pixels processed.

`class` **`CIRenderTask`**

A single render task.

`enum` **`CIRenderDestinationAlphaMode`**

Different ways of representing alpha.