Sample Code

Rendering stereoscopic video with RealityKit

Render stereoscopic video in visionOS with RealityKit.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

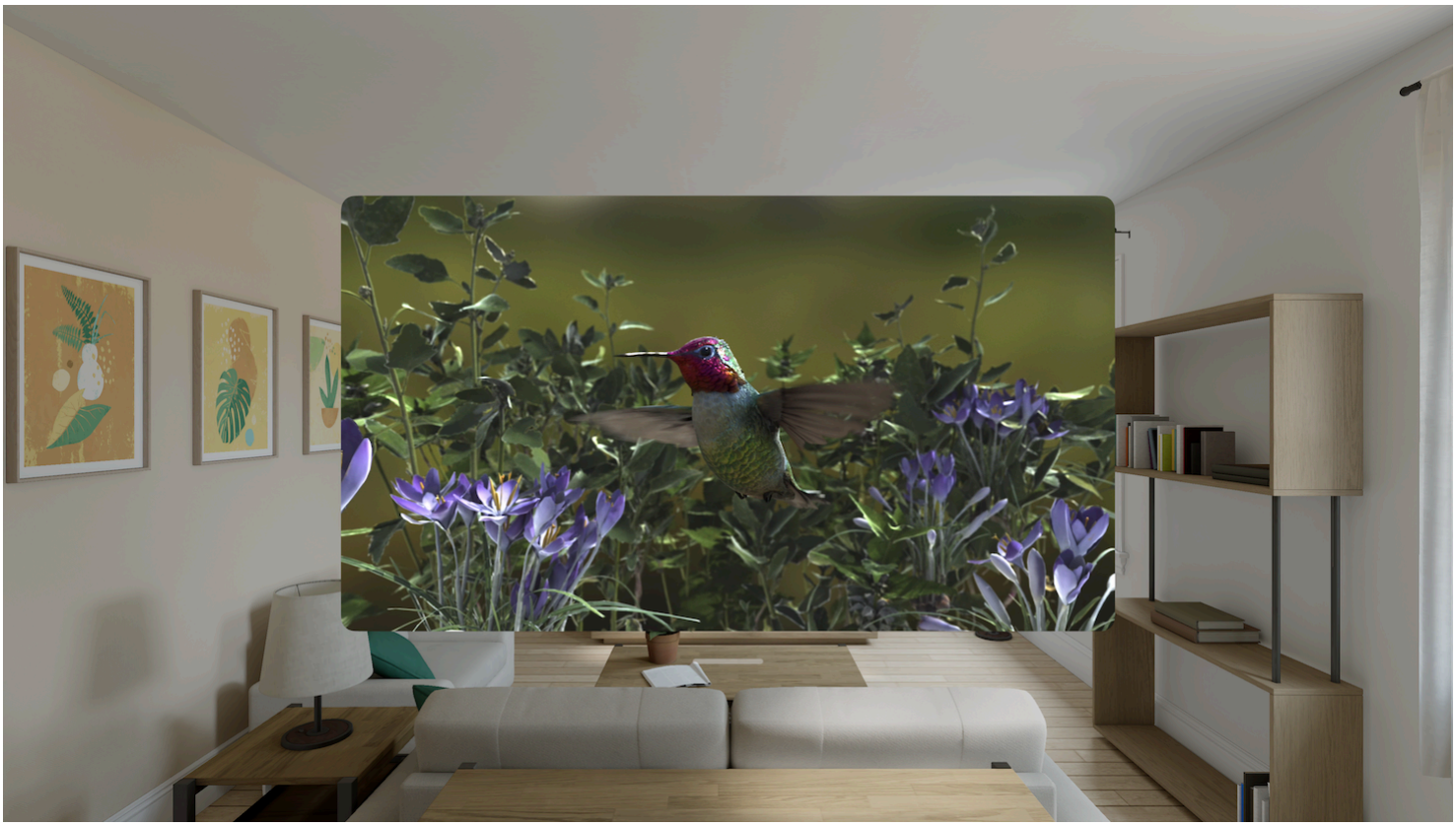
visionOS offers a range of options for programmatic video playback, including:

- AVKit provides a superior video playback experience in visionOS. With AVKit, you can present an interface that's consistent with other apps on the system, with minimal adoption effort. For more information on using AVKit in visionOS, see Adopting the system player interface in visionOS.
- RealityKit offer a greater degree of customization, using either `VideoMaterial` or `VideoPlayerComponent`.

`VideoMaterial` and `VideoPlayerComponent` offer two distinct options for controlling playback.

- `AVPlayer` is a versatile system component, appropriate for controlling both playback and timing of a media asset.
- `AVSampleBufferVideoRenderer` enables greater customization — it requires that you programmatically enqueue individual video-sample buffers for rendering. You can also use an `AVSampleBufferAudioRenderer` instance with a `AVSampleBufferRenderSynchronizer`, which affords I/O control, supports preprocessing of media data, and accommodates DRM models not supported by `AVPlayer`. An `AVSampleBufferRenderSynchronizer` can optionally synchronize audio with an `AVSampleBufferAudioRenderer`.

This sample app uses an `AVSampleBufferVideoRenderer` and a `VideoPlayerComponent` to render stereoscopic video in the *Shared Space*. Its content is a *side-by-side* video, which places the left- and right-eye images next to each other as part of a single video frame. Because the duration of the video is brief, a looping mechanism supports continuous playback.



> **Note**
>
> By default, 3D video in visionOS uses the Multiview High Efficiency Video Encoding (MV-HEVC) format, supported by MPEG4 and QuickTime. For information about converting a file like the one in this sample to MV-HEVC, see Converting side-by-side 3D video to multiview HEVC and spatial video.

Structure the app

The structure of the app is simple. `PlayerModel` is an `Observable` custom type that's injected into the SwiftUI `Environment` for visibility to the root `ContentView`. This model includes a property of type `PlayerState`, which is a Swift enumeration that reflects the current player state. It also includes an instance of type `LoopingVideoPlayer`, which exposes the underlying `AVSampleBufferVideoRenderer`.

```
/// The main app structure.
@main
struct RealityKitPlaybackApp: App {
```

```
    /// An object that controls the video playback behavior.
    @State private var player = PlayerModel()

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(player)
                .frame(width: 1600, height: 900)
        }
        // Expressly constrain window size to that of its content.
        .windowResizability(.contentSize)
        // Disable background glass.
        .windowStyle(.plain)
    }
}
```

For simplicity, the following modifiers are applied to create a window with fixed size, 16x9 aspect ratio, and absent background glass:

- `frame(width:height:alignment:)`
- `windowResizability(_:)`
- `windowStyle(_:)`

Create an entity to render video content

Use the `PlayerModel` as a property exposed from `ContentView`:

```
/// A reference to the player.
@Environment(PlayerModel.self) private var player
```

`ContentView` also defines an `Entity` to house the `RealityView` at the root of this scene:

```
/// The root entity of the scene.
private let entity = Entity()
```

In the `RealityView` `make` closure, the app instantiates a `VideoPlayerComponent`, passing the player's video renderer. It then adds the video player to the root entity. The sample uses `scaleFactor`, a type property set to `0.5`, to scale the root entity to half its default size.

```
RealityView { content in
    // Initialize the video player with the supplied renderer.
    let videoPlayerComponent = VideoPlayerComponent(videoRenderer: player.videoRende
    entity.components.set(videoPlayerComponent)

    // Scale the root entity and add it to the view.
    entity.scale = SIMD3<Float>(repeating: Self.scaleFactor)
    content.add(entity)
}
// Set the frame to 0 so that the RealityView's origin exists on the same plane as t
.frame(depth: 0)
```

Finally, the sample applies these modifiers to initialize the player, and to begin and end playback:

- onChange(of:initial:_:)
- task(priority:_:)

```
// Begin playback when ready.
.onChange(of: player.isReadyToPlay) { _, ready in
    if ready {
        player.play()
    }
}
// Monitor the scene phase and stop playback when entering the background.
.onChange(of: scenePhase) { _, scenePhase in
    if scenePhase == .background {
        player.stop()
    }
}
// Start loading the player.
.task {
    await player.load()
}
```

Prepare for continuous playback

`LoopingVideoPlayer` is a custom type that coordinates continuous playback of the sample video. To achieve this, it manages multiple instances of another custom type, `SerialProcessor`.

The player has two key properties: a video renderer and a synchronizer to control the rendering timeline:

```
/// The synchronizer that controls the underlying video renderer.
private let synchronizer = AVSampleBufferRenderSynchronizer()

/// The video renderer that enqueues individual frames for playback.
let videoRenderer = AVSampleBufferVideoRenderer()
```

When the system creates the player, it adds the renderer to the synchronizer, and initializes an `AVURLAsset` with a URL to the underlying video:

```
/// Initializes a player with the specified asset URL.
/// - Parameter assetURL: A URL for the asset that the app plays.
init(assetURL: URL) {
    synchronizer.addRenderer(videoRenderer)
    asset = AVURLAsset(url: assetURL)
}
```

To prepare for playback, the processor loads the asset duration asynchronously with `AVAsynchronousKeyValueLoading`. The sample uses duration to trigger looping at the end of each playback cycle with `addBoundaryTimeObserver(forTimes:queue:using:)`. The sample initializes the first serial processor with the video renderer and asset:

```
/// Begin loading the player.
func load() async throws {
    // Determine the duration of the underlying video asset.
    let duration = try await asset.load(.duration)

    // Use the asset duration as the boundary period with which to loop.
    synchronizer.addBoundaryTimeObserver(forTimes: [NSValue(time: duration)], queue:
        Task { @MainActor [weak self] in
            guard let self else { return }
            self.loop(rate: self.synchronizer.rate)
        }
    }

    // Prepare the processor that the app uses for the initial playback loop.
    enqueueProcessor()
}
```

Loop playback

The app begins playback by calling `loop(rate:)` for the first time. The initial rate of the render synchronizer is `0.0`, meaning that playback has stopped. Passing `1.0` starts playback at the natural rate of the media.

```
/// Begin playback by starting the loop.
func play() {
    isLooping = true
    loop(rate: 1)
}
```

The app starts the loop by dequeueing a serial processor instance. It specifies the time and rate of the render synchronizer with `setRate(_:time:)`. It then enqueues the next serial processor instance and increments the loop count:

```
/// Executes a logical playback loop.
/// - Parameter rate: The rate with which to playback content.
private func loop(rate: Float) {
    guard isLooping, let nextProcessor else {
        return
    }

    let currentProcessor = nextProcessor
    process(with: currentProcessor)
    synchronizer.setRate(rate, time: .zero)

    enqueueProcessor()
    loopCount += 1
}
```

The boundary time observer initiates subsequent loops. Playback continues until someone closes the scene. At that time, the sample calls `stop()` to dispose of the player's resources:

```
/// End playback by stopping the loop and resetting relevant state.
func stop() {
    nextProcessor = nil
    isLooping = false
    loopCount = 0
    synchronizer.rate = 0
    videoRenderer.stopRequestingMediaData()
}
```

Load the side-by-side video

Each `SerialProcessor` traverses the video track from start to finish, extracting each individual video frame for processing. Processing converts these input frames from the single-layer, side-by-side input format to a multi-layer, output format.

With side-by-side input, the sample places left- and right-eye images next to each other as part of a single frame. The sample splits the frame into separate images, copies them to distinct left- and right-eye layers, and writes them as a multi-layer frame.

The processor begins when the sample calls `loadTracks(withMediaCharacteristic:completionHandler:)` to load video tracks, and then selects the first available track as the side-by-side input.

```
// Load the asset.
guard let videoTrack = try await asset.loadTracks(withMediaCharacteristic: .visual).
    fatalError("Error loading side-by-side video input")
}
```

The processor also loads the natural size of the side-by-side video for later use:

```
// Determine the size of the video track, which reflects frame packing.
let videoFrameSize = try await videoTrack.load(.naturalSize)
```

The processor specifies IOSurface settings in its `readerSettings` dictionary. Because the sample manages its own pixel buffer allocations, it uses an empty array as the value corresponding to the `kCVPixelBufferIOSurfacePropertiesKey` key. These settings create an `AVAssetReaderTrackOutput`. To finish loading the video, the sample obtains an output provider from the `AVAssetReader`, and starts reading.

```
// Setup the asset reader.
let readerSettings: [String: Any] = [
    kCVPixelBufferIOSurfacePropertiesKey as String: [String: String]()
]
let videoTrackOutput = AVAssetReaderTrackOutput(track: videoTrack, outputSettings: r
let assetReader = try AVAssetReader(asset: asset)
let videoTrackOutputProvider = assetReader.outputProvider(for: videoTrackOutput)
try assetReader.start()
```

Create the video frame-transfer session and output pixel-buffer pool

To prepare for processing the video input, the sample creates a `VTPixelTransferSession` to read raw CVPixelBuffer input and write processed `CVPixelBuffers` as output.

```
// Setup the pixel transfer session.
var transferSession: VTPixelTransferSession?
let sessionResult = VTPixelTransferSessionCreate(
    allocator: kCFAllocatorDefault,
    pixelTransferSessionOut: &transferSession
)
guard sessionResult == kCVReturnSuccess, let transferSession else {
    fatalError("Failed to create pixel transfer session: \(sessionResult)")
}
VTSessionSetProperty(transferSession, key: kVTPixelTransferPropertyKey_ScalingMode,
```

For efficiency, the sample creates a `CVMutablePixelBuffer.Pool` to allocate pixel buffers for the processed multi-layer output. It creates a pool with attributes that include the pixel format type and size of the eye frame. The sample derives the eye frame size from the natural video size, previously loaded. It then merges these specified attributes with `recommendedPixelBufferAttributes`.

```
// Setup the pixel buffer pool.
let eyeFrameSize = CVImageSize(
    width: Int(videoFrameSize.width / stereoMetadata.horizontalScale),
    height: Int(videoFrameSize.height / stereoMetadata.verticalScale)
)
let defaultAttributes = CVPixelBufferCreationAttributes(
    pixelFormatType: CVPixelFormatType(rawValue: kCVPixelFormatType_420YpCbCr8BiPlar
    size: eyeFrameSize
)
let recommendedAttributes = videoRenderer.recommendedPixelBufferAttributes
guard let mergedAttributes = CVPixelBufferAttributes(merging: [CVPixelBufferAttribut
      let creationAttributes = CVPixelBufferCreationAttributes(mergedAttributes),
      let pixelBufferPool = try? CVMutablePixelBuffer.Pool(pixelBufferAttributes: cr
else {
    fatalError("Failed to create pixel buffer pool")
}
```

Process input as it becomes available

To begin processing, the processor waits for the video renderer to indicate that it is ready to begin rendering. The private `untilReadyForMoreMediaData()` function achieves this with a call to `requestMediaDataWhenReady(on:using:)`. As the sample reads the asset, the `videoTrackOutputProvider` supplies a stream of sample buffers for processing. As the sample receives these sample buffers, the processor calls `transform(from:with:in:)` to convert the side-by-side frame input into stereo-encoded output. The sample then enqueues the stereo-encoded frames to the video renderer. Processing concludes once the stream of sample buffers is exhausted.

```
Task {
    // Prepare the renderer for processing.
    await untilReadyForMoreMediaData()
    isProcessing = true

    // Process all read frames from the input video track.
    while videoRenderer.isReadyForMoreMediaData && isProcessing {
        while let sampleBuffer = try await videoTrackOutputProvider.next() {
            if let transformedBuffer = try transform(from: sampleBuffer, with: pixel
                videoRenderer.enqueue(transformedBuffer)
            }
        }

        // Indicate that processing is substantially complete.
        isProcessing = false
    }

    // Conclude processing.
    assetReader.cancelReading()
    VTPixelTransferSessionInvalidate(transferSession)
}
```

Transform side-by-side input to multi-layer output

The processor creates individual left- and right-eye images in the transformation function. It specifies layer ID `0` for the left eye, and `1` for the right eye.

```
let layerIDs = [0, 1]
let eyeComponents: [CMStereoViewComponents] = [.leftEye, .rightEye]
var taggedBuffers = [CMTaggedDynamicBuffer]()
```

```
for (layerID, eye) in zip(layerIDs, eyeComponents) {
    // ...
```

The function uses the `VTPixelTransferSession` to copy pixels from the side-by-side source pixel buffer, crop to the frame for the current eye, and place them into the destination pixel buffer.

```
// Crop the transfer region to the current eye.
let bufferSize = pixelBufferPool.pixelBufferAttributes.size
let apertureOffset = stereoMetadata.apertureOffset(for: bufferSize, layerID: layerID
let cropRectDict = [
    kCVImageBufferCleanApertureHorizontalOffsetKey: apertureOffset.horizontal,
    kCVImageBufferCleanApertureVerticalOffsetKey: apertureOffset.vertical,
    kCVImageBufferCleanApertureWidthKey: bufferSize.width,
    kCVImageBufferCleanApertureHeightKey: bufferSize.height
]
CVBufferSetAttachment(sourceImageBuffer, kCVImageBufferCleanApertureKey, cropRectDic
VTSessionSetProperty(transferSession, key: kVTPixelTransferPropertyKey_ScalingMode,

// Transfer the image to the pixel buffer.
pixelBuffer.withUnsafeBuffer { cvPixelBuffer in
    let transferResult = VTPixelTransferSessionTransferImage(transferSession, from:
    guard transferResult == kCVReturnSuccess else {
        fatalError("Error during pixel transfer session for layer \(layerID): \(trar
    }
}
```

The sample creates the individual left and right pixel buffers, adorns them with `CMTag` metadata, and stores them as `CMTaggedDynamicBuffer` pairs.

```
// Create and append a tagged buffer for this eye.
let tags: [CMTag] = [.videoLayerID(Int64(layerID)), .stereoView(eye), .mediaType(.vi
taggedBuffers.append(CMTaggedDynamicBuffer(tags: tags, content: .pixelBuffer(CVReadC
```

Finally, the sample combines the tagged buffers with the presentation timestamp and duration of the input sample buffer and creates the final output sample buffer.

```
let buffer = CMReadySampleBuffer(
    taggedBuffers: taggedBuffers,
    formatDescription: CMTaggedBufferGroupFormatDescription(taggedBuffers: taggedBuf
    presentationTimeStamp: cmSampleBuffer.presentationTimeStamp,
    duration: cmSampleBuffer.duration
```

```
)
```