Sample Code

Reading multiview 3D video files

Render single images for the left eye and right eye from a multiview High Efficiency Video Coding format file by reading individual video frames.

Download

macOS 14.0+ | Xcode 15.2+

Overview

Multiview High Efficiency Video Coding (MV-HEVC) media files contain information to produce stereoscopic frames, one for the left eye and one for the right, to create an effect of depth and allow for 3D video. This is the standard format for presenting 3D video in visionOS, encoded as MPEG-4 or QuickTime files.

Previewing and testing MV-HEVC files without hardware requires the ability to load, view, and step through the video data on a timeline. This sample app opens a media file, checking for the MV-HEVC format, then presents a view containing the individual frames at the timestamp. Step through the timeline by dragging the slider to a specific timestamp, or advance to the next frame by pressing the Space bar.

For the full details of the MV-HEVC format, see Apple HEVC Stereo Video - Interoperability Profile (PDF) and ISO Base Media File Format and Apple HEVC Stereo Video (PDF).

Load and inspect the media asset

The app first displays a button labeled Open MVHEVC File. When selected, the button presents an `NSOpenPanel` for choosing video media. Next, the app initializes a `MediaDetailViewModel`, loading this file as an `AVURLAsset`. Before opening the file to present any elements for a stereo video frame, the app ensures a playable, readable file, and gets its total length in time. This is all performed in the initializer.

```
init(filename: URL) {
    asset = AVURLAsset(url: filename)
    Task { @MainActor in
        do {
            let (duration, isPlayable, isReadable) = try await asset.load(.duration,
            self.duration = duration.seconds
            self.isPlayable = isPlayable
            self.isReadable = isReadable
        } catch {
            self.error = error
        }
    }
}
```

Load track data and timestamps

After confirming the track is readable video data, the app initializes a `StereoViewModel` by calling `loadTracks(withMediaCharacteristic:completionHandler:)` requesting a `containsStereoMultiviewVideo` track. This check confirms that the file meets the MV-HEVC specification and has valid stereo data.

```
if let track = try await asset.loadTracks(withMediaCharacteristic: .containsStereoMu
    self.track = track
```

Next, the app pulls available timestamps for each frame in the track by calling `presentationTimesFor(track:asset:)`. The app places a video sample cursor at the start of the track with `makeSampleCursorAtFirstSampleInDecodeOrder()`, then creates a new `AVSampleBufferGenerator` and `AVSampleBufferRequest`.

```
guard let cursor = track.makeSampleCursorAtFirstSampleInDecodeOrder() else {
    return []
}
let sampleBufferGenerator = AVSampleBufferGenerator(asset: asset, timebase: nil)
var presentationTimes = [CMTime]()
let request = AVSampleBufferRequest(start: cursor)
var numSamples: Int64 = 0
```

To read the timestamps, obtain the sample buffer for the current cursor from `makeSampleBuffer(for:)`, then add the `presentationTimeStamp` for the frame. The cursor steps forward by calling `stepInDecodeOrder(byCount:)`, reading and caching timestamps for each

frame in the buffer. When `stepInDecodeOrder(byCount:)` returns no next frame, sample times are in the cache and reading the video track completes.

```
repeat {
    let buf = try sampleBufferGenerator.makeSampleBuffer(for: request)
    presentationTimes.append(buf.presentationTimeStamp)
    numSamples = cursor.stepInDecodeOrder(byCount: 1)
} while numSamples == 1
```

Load video layer information

After preparing timestamps, the app calls `loadVideoLayerIdsForTrack()` to get the layer IDs for the two tracks associated with the left and right eyes. The app calls `load(_:isolation:)` to retrieve metadata, then filters the layer data out of the first available track's `tagCollections`. The filter predicate is `value(onlyIfMatching:)`, extracting only video layer IDs.

```
private func loadVideoLayerIdsForTrack(_ videoTrack: AVAssetTrack) async throws -> [
    let formatDescriptions = try await videoTrack.load(.formatDescriptions)
    var tags = [Int64]()
    if let tagCollections = formatDescriptions.first?.tagCollections {
        tags = tagCollections.flatMap({ $0 }).compactMap { tag in
            tag.value(onlyIfMatching: .videoLayerID)
        }
    }
    return tags
}
```

Load video frames from buffers

With the timestamp and left eye and right eye video layers identified, `readBufferFrom Asset(at:)` calls the `readNextBufferFromAsset()` method of the app to retrieve and display the frame data. The method starts with a series of `guard` checks to ensure read access to the track, creates a local copy of the sample buffer by calling `copyNextSampleBuffer()`, and retrieves the tagged video buffers from the track.

```
guard let assetReader, let trackOutput else {
    return
}
guard assetReader.status == .reading else {
    publishState(.error(message: "UNEXPECTED STATUS \(assetReader.status)"))
```

```
    return
}
guard let sampleBuffer = trackOutput.copyNextSampleBuffer() else {
    publishState(.error(message: "READING SAMPLE BUFFER, STATUS \(assetReader.status
    return
}
guard let taggedBuffers = sampleBuffer.taggedBuffers else {
    publishState(.error(message: "SAMPLE BUFFER CONTAINS NO TAGGED BUFFERS: \(sample
    return
}
guard taggedBuffers.count == 2 else {
    publishState(.error(message: "EXPECTED 2 TAGGED BUFFERS, GOT \(taggedBuffers.cou
    return
}
```

The app parses each CMTaggedBuffer.Buffer.pixelBuffer(_:) from the returned sample buffers into an image for display using init(cvPixelBuffer:). The app creates an NSImage and sets it to the view content as either `leftEye` or `rightEye` depending on whether the view contains a stereoView(_:) for the left or right eye.

```
taggedBuffers.forEach { taggedBuffer in
    switch taggedBuffer.buffer {
    case let .pixelBuffer(pixelBuffer):
        let ciimage = CIImage(cvPixelBuffer: pixelBuffer)
        let context: CIContext = CIContext(options: nil)
        let cgImage: CGImage = context.createCGImage(ciimage, from: ciimage.extent)!
        let tags = taggedBuffer.tags
        Task {
            await MainActor.run {
                let nsImage = NSImage(cgImage: cgImage, size: NSSize(width: 320, hei
                if tags.contains(.stereoView(.leftEye)) {
                    leftEye = nsImage
                } else if tags.contains(.stereoView(.rightEye)) {
                    rightEye = nsImage
                }
            }
        }
    case .sampleBuffer(let samp):
        publishState(.error(message: "EXPECTED PIXEL BUFFER, GOT SAMPLE BUFFER \(sam
    @unknown default:
        publishState(.error(message: "EXPECTED PIXEL BUFFER TYPE, GOT \(taggedBuffer
    }
```

```
}
```

See Also

Media reading

`class AVAssetReader`

An object that reads media data from an asset.

`class AVAssetReaderOutput`

An abstract class that defines the interface to read media samples from an asset reader.

`class AVAssetReaderTrackOutput`

An object that reads media data from a single track of an asset.

`class AVAssetReaderAudioMixOutput`

An object that reads audio samples that result from mixing audio from one or more tracks.

`class AVAssetReaderVideoCompositionOutput`

An object that reads composited video frames from one or more tracks of an asset.

`class AVAssetReaderSampleReferenceOutput`

An object that reads sample references from an asset track.

`class AVAssetReaderOutputMetadataAdaptor`

An object that creates timed metadata group objects for an asset track.