Sample Code

Converting side-by-side 3D video to multiview HEVC and spatial video

Create video content for visionOS by converting an existing 3D HEVC file to a multiview HEVC format, optionally adding spatial metadata to create a spatial video.

Download

macOS 15.0+ | Xcode 16.0+

Overview

In visionOS, 3D video uses the *Multiview High Efficiency Video Encoding* (MV-HEVC) format, supported by MPEG4 and QuickTime. Unlike other 3D media, MV-HEVC stores a single track containing multiple layers for the video, where the track and layers share a frame size. This track frame size is different from other 3D video types, such as *side-by-side video*. Side-by-side videos use a single track, and place the left- and right-eye images next to each other as part of a single video frame.

To convert side-by-side video to MV-HEVC, you load the source video, extract each frame, and then split the frame horizontally. Then copy the left and right sides of the split frame into the left- and right-eye layers, writing a frame containing both layers to the output.

This sample app demonstrates the process for converting side-by-side video files to MV-HEVC, encoding the output as a QuickTime file. The output is placed in the same directory as the input file, with _MVHEVC appended to the original filename.

For videos you capture with a consistent camera configuration, you can optionally add spatial metadata to the output file. *Spatial metadata* describes properties of the left- and right-eye cameras that captured the stereo scene.

Adding spatial metadata to a stereo MV-HEVC video prompts Apple platforms to consider the video as *spatial* instead of just stereo, and opts the video into visual treatments on Apple Vision Pro that can minimize common causes of stereo viewing discomfort.

To learn more about when to provide spatial metadata for a stereo MV-HEVC video and the metadata values to provide, see Creating spatial photos and videos with spatial metadata.

You can verify this sample's MV-HEVC output by opening it with the sample project from Reading multiview 3D video files.

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).

Configure the sample code project

The app takes a path to a side-by-side stereo input video file as a single command-line argument. To run the app in Xcode, select Product > Scheme > Edit Scheme (Command-<), and add the path to your file to Arguments Passed On Launch.

To also add spatial metadata to the file, add four additional arguments to Arguments Passed On Launch:

- `--spatial` (or `-s`) to indicate that you want to include spatial metadata
- `--baseline` (or `-b`) to provide a baseline in millimeters (for example, `--baseline 64.0` for a 64mm baseline)
- `--fov` (or `-f`) to provide a horizontal field of view in degrees (for example, `--fov 80.0` for an 80-degree field of view)
- `--disparityAdjustment` (or `-d`) to provide a disparity adjustment (for example, `--disparityAdjustment 0.02` for a 2% positive disparity shift)

By default, the project's scheme loads a side-by-side video from the Xcode project folder named `Hummingbird.mov`. This video is a sequence of renders of a 3D scene, showing an animated hummingbird model. By default, the app converts this example video to a stereo MV-HEVC file, without spatial metadata.

To add spatial metadata to the hummingbird video during conversion, choose Product > Scheme > Edit Scheme (Command-<), and select the checkbox to the left of the second row of arguments in the Arguments Passed On Launch field. This enables the following additional arguments: `--spatial --baseline 64.0 --fov 80.0 --disparityAdjustment 0.02`.

The `--spatial` argument tells the app to write spatial metadata to the output video. The virtual cameras used to create these renders were positioned 64mm apart with a horizontal field of view of 80 degrees, and so the value for the `--baseline` argument is `64.0`, and the value of the `--fov` argument is `80.0`.

For this video, a disparity adjustment of +2% of the image width produces a comfortable depth effect when the spatial video is presented in a window on Apple Vision Pro. This 2% disparity adjustment value positions the nearest object in the spatial video — the hummingbird — just behind the front of the window, while still keeping an effective illusion of depth between the hummingbird and the background. The scheme's arguments express the +2% disparity adjustment with a `--disparityAdjustment` value of `0.02`.

Load the side-by-side video

The app starts by loading the side-by-side video, creating an `AVAssetReader`. The app calls `loadTracks(withMediaCharacteristic:completionHandler:)` to load video tracks, and then selects the first track available as the side-by-side input.

```
let asset = AVURLAsset(url: url)
reader = try AVAssetReader(asset: asset)

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

The app also stores the frame size for the side-by-side video, and calculates the size of the output frames.

```
sideBySideFrameSize = try await videoTrack.load(.naturalSize)
eyeFrameSize = CGSize(width: sideBySideFrameSize.width / 2, height: sideBySideFrameS
```

To finish loading the video, the app creates an `AVAssetReaderTrackOutput` and then adds this output stream to the `AVAssetReader`.

```
let readerSettings: [String: Any] = [
    kCVPixelBufferIOSurfacePropertiesKey as String: [String: String]()
]
sideBySideTrack = AVAssetReaderTrackOutput(track: videoTrack, outputSettings: reader

if reader.canAdd(sideBySideTrack) {
    reader.add(sideBySideTrack)
}

if !reader.startReading() {
    fatalError(reader.error?.localizedDescription ?? "Unknown error during track rea
}
```

When creating the reader track output, the app specifies the file's pixel format and IOSurface settings in the `readerSettings` dictionary. The app indicates that output goes to a 32-bit ARGB pixel buffer, using `kCVPixelBufferPixelFormatTypeKey` with a value of `kCVPixelFormatType_32ARGB`. The sample app also manages its own pixel buffer allocations, passing an empty array as the value for `kCVPixelBufferIOSurfacePropertiesKey`.

Configure the output MV-HEVC file

With the reader initialized, the app calls the `async` method `transcodeToMVHEVC(output:spatialMetadata:)` to generate the output file. First, the app creates a new `AVAssetWriter` pointing to the video output location, and then configures the necessary information on the output to indicate that the file contains MV-HEVC video.

```
var multiviewCompressionProperties: [CFString: Any] = [
    kVTCompressionPropertyKey_MVHEVCVideoLayerIDs: MVHEVCVideoLayerIDs,
    kVTCompressionPropertyKey_MVHEVCViewIDs: MVHEVCViewIDs,
    kVTCompressionPropertyKey_MVHEVCLeftAndRightViewIDs: MVHEVCLeftAndRightViewIDs,
    kVTCompressionPropertyKey_HasLeftStereoEyeView: true,
    kVTCompressionPropertyKey_HasRightStereoEyeView: true
]
```

`kVTCompressionPropertyKey_HasLeftStereoEyeView` and `kVTCompressionPropertyKey_HasRightStereoEyeView` are `true`, because the output contains a layer for each eye. `kVTCompressionPropertyKey_MVHEVCVideoLayerIDs`, `kVTCompressionPropertyKey_MVHEVCViewIDs`, and `kVTCompressionPropertyKey_MVHEVCLeftAndRightViewIDs` define the layer and view IDs to use for multiview HEVC encoding. In the sample app, these are all the same.

The sample app uses `0` for the left eye layer/view ID and 1 for the right eye layer/view ID.

```
let MVHEVCVideoLayerIDs = [0, 1]

// For simplicity, choose view IDs that match the layer IDs.
let MVHEVCViewIDs = [0, 1]

// The first element in this array is the view ID of the left eye.
let MVHEVCLeftAndRightViewIDs = [0, 1]
```

Include spatial metadata

If the person calling this command-line app requested to add spatial metadata to the output file, and provided the necessary spatial metadata, the app converts that metadata to expected units and scales, and adds an additional compression property key for each metadata value. The app also specifies that the input uses a rectilinear projection, to indicate that it has the expected projection for spatial video.

```
if let spatialMetadata {

    let baselineInMicrometers = UInt32(1000.0 * spatialMetadata.baselineInMillimete
    let encodedHorizontalFOV = UInt32(1000.0 * spatialMetadata.horizontalFOV)
    let encodedDisparityAdjustment = Int32(10_000.0 * spatialMetadata.disparityAdjus

    multiviewCompressionProperties[kVTCompressionPropertyKey_ProjectionKind] = kCMFo
    multiviewCompressionProperties[kVTCompressionPropertyKey_StereoCameraBaseline] =
    multiviewCompressionProperties[kVTCompressionPropertyKey_HorizontalFieldOfView]
    multiviewCompressionProperties[kVTCompressionPropertyKey_HorizontalDisparityAdju

}
```

Configure the MV-HEVC input source

The app transcodes video by directly copying pixels from the source frame. Writing track data to a video file requires an `AVAssetWriterInput`. The sample app uses an `AVAssetWriterInputTaggedPixelBufferGroupAdaptor` to provide pixel data from the source, writing to the output.

```
let multiviewSettings: [String: Any] = [
    AVVideoCodecKey: AVVideoCodecType.hevc,
    AVVideoWidthKey: self.eyeFrameSize.width,
    AVVideoHeightKey: self.eyeFrameSize.height,
    AVVideoCompressionPropertiesKey: multiviewCompressionProperties
]

guard multiviewWriter.canApply(outputSettings: multiviewSettings, forMediaType: AVMe
    fatalError("Error applying output settings")
}

let frameInput = AVAssetWriterInput(mediaType: .video, outputSettings: multiviewSett
let sourcePixelAttributes: [String: Any] = [
    kCVPixelBufferPixelFormatTypeKey as String: kCVPixelFormatType_32ARGB,
    kCVPixelBufferWidthKey as String: self.sideBySideFrameSize.width,
    kCVPixelBufferHeightKey as String: self.sideBySideFrameSize.height
]

let bufferInputAdapter = AVAssetWriterInputTaggedPixelBufferGroupAdaptor(assetWrite
```

The `AVAssetWriterInput` source uses the same `outputSettings` as `videoWriter`, and the created pixel buffer adapter has the same frame size as the source. The app follows the best practice of calling `canAdd(_:)` to check the input adapter compatibility before calling `add(_:)` to use it as a source.

```
guard multiviewWriter.canAdd(frameInput) else {
    fatalError("Error adding side-by-side video frames as input")
}
multiviewWriter.add(frameInput)
```

Process input as it becomes available

The app calls `startWriting()` and `startSession(atSourceTime:)` in sequence to start the video writing process, and then iterates over available frame inputs with `requestMediaDataWhenReady(on:using:)`.

```
guard multiviewWriter.startWriting() else {
    fatalError("Failed to start writing multiview output file")
}
multiviewWriter.startSession(atSourceTime: CMTime.zero)

// The dispatch queue executes the closure when media reads from the input file are
frameInput.requestMediaDataWhenReady(on: DispatchQueue(label: "Multiview HEVC Writer
```

The closure argument of `requestMediaDataWhenReady(on:using:)` runs on the provided `DispatchQueue` when the first data read is available. The closure itself is responsible for managing resources that process the media data, and running a loop to process data efficiently.

Create the video frame transfer session and output pixel buffer pool

To perform the data transfer from the source track, the pixel input adapter requires a pixel buffer as a source. The app creates a VTPixelTransferSession to allow for reading data from the video source, and uses the AVAssetWriterInputTaggedPixelBufferGroupAdaptor's existing pixel buffer pool to allocate pixel buffers for the new multiview eye layers.

```
var session: VTPixelTransferSession? = nil
guard VTPixelTransferSessionCreate(allocator: kCFAllocatorDefault, pixelTransferSess
    fatalError("Failed to create pixel transfer")
}
guard let pixelBufferPool = bufferInputAdapter.pixelBufferPool else {
    fatalError("Failed to retrieve existing pixel buffer pool")
}
```

Copy frame images from input to output

After preparing resources, the app then begins a loop to process frames until there's no more data, or the input read has stopped to buffer data. The isReadyForMoreMediaData property of an input source is true if another frame is immediately available to process. When a frame is ready, a CVImageBuffer instance is created from it.

The app is now ready to handle sampling. If there's an available sample, the app processes it in the convertFrame method, then calls appendTaggedBuffers(_:withPresentationTime:), copying the side-by-side sample buffer's outputPresentationTimeStamp timestamp to the new multiview timestamp.

```
while frameInput.isReadyForMoreMediaData && bufferInputAdapter.assetWriterInput.isRe
    if let sampleBuffer = self.sideBySideTrack.copyNextSampleBuffer() {
        guard let imageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) else {
            fatalError("Failed to load source samples as an image buffer")
        }
        let taggedBuffers = self.convertFrame(fromSideBySide: imageBuffer, with: pix
        let newPTS = sampleBuffer.outputPresentationTimeStamp
        if !bufferInputAdapter.appendTaggedBuffers(taggedBuffers, withPresentationTi
            fatalError("Failed to append tagged buffers to multiview output")
        }
```

Input reading finishes when there are no more sample buffers to process from the input stream. The app calls markAsFinished() to close the stream, and finishWriting(completionHandler:) to complete the multiview video write. The app also calls resume() on its associated CheckedContinuation, to return to the await call, then breaks from the processing loop.

```
frameInput.markAsFinished()
multiviewWriter.finishWriting {
    continuation.resume()
}

break
```

Convert side-by-side inputs into video layer outputs

In the `convertFrame` method, the app processes the left- and right-eye images for the frame by `layerID`, using `0` for the left eye and `1` for the right. First, the app creates a pixel buffer from the pool.

```
var pixelBuffer: CVPixelBuffer?
CVPixelBufferPoolCreatePixelBuffer(kCFAllocatorDefault, pixelBufferPool, &pixelBuffe
guard let pixelBuffer else {
    fatalError("Failed to create pixel buffer for layer \(layerID)")
}
```

The method then uses its passed `VTPixelTransferSession` to copy the pixels from the side-by-side source, placing them into the created output sample buffer by cropping the frame to include only one eye's image.

```
// Crop the transfer region to the current eye.
let apertureOffset = -(self.eyeFrameSize.width / 2) + CGFloat(layerID) * self.eyeFra
let cropRectDict = [
    kCVImageBufferCleanApertureHorizontalOffsetKey: apertureOffset,
    kCVImageBufferCleanApertureVerticalOffsetKey: 0,
    kCVImageBufferCleanApertureWidthKey: self.eyeFrameSize.width,
    kCVImageBufferCleanApertureHeightKey: self.eyeFrameSize.height
]
CVBufferSetAttachment(imageBuffer, kCVImageBufferCleanApertureKey, cropRectDict as C
VTSessionSetProperty(session, key: kVTPixelTransferPropertyKey_ScalingMode, value: k

// Transfer the image to the pixel buffer.
guard VTPixelTransferSessionTransferImage(session, from: imageBuffer, to: pixelBuffe
    fatalError("Error during pixel transfer session for layer \(layerID)")
}
```

Setting aperture view properties on `CVBufferSetAttachment(_:_:_:_:)` defines how to capture and crop input images. The aperture here is the size of an eye image, and the center of the capture frame offset with `kCVImageBufferCleanApertureHorizontalOffsetKey` by `-0.5 * width` for the left eye and `+0.5 * width` for the right eye, to capture the correct half of the side-by-side frame.

The app then calls `VTSessionSetProperty(_:key:value:)` to crop the image to the aperture frame with `kVTScalingMode_CropSourceToCleanAperture`. Next, the app calls `VTPixelTransferSessionTransferImage(_:from:to:)` to copy source pixels to the destination buffer.

The final step is to create a `CMTaggedBuffer` for the eye image to return to the calling output writer.

```
let tags: [CMTag] = [.videoLayerID(Int64(layerID)), .stereoView(eye)]
let buffer = CMTaggedBuffer(tags: tags, buffer: .pixelBuffer(pixelBuffer))
taggedBuffers.append(buffer)
```

See Also

Media writing

{} Converting projected video to Apple Projected Media Profile

Convert content with equirectangular or half-equirectangular projection to APMP.

{} Writing fragmented MPEG-4 files for HTTP Live Streaming

Create an HTTP Live Streaming presentation by turning a movie file into a sequence of fragmented MPEG-4 files.

Creating spatial photos and videos with spatial metadata

Add spatial metadata to stereo photos and videos to create spatial media for viewing on Apple Vision Pro.

Tagging media with video color information

Inspect and set video color space information when writing and transcoding media.

Evaluating an app’s video color

Check color reproduction for a video in your app by using test patterns, video test equipment, and light-measurement instruments.

`class AVOutputSettingsAssistant`

An object that builds audio and video output settings dictionaries.

`class AVAssetWriter`

An object that writes media data to a container file.

`class AVAssetWriterInput`

An object that appends media samples to a track in an asset writer's output file.

`class AVAssetWriterInputPixelBufferAdaptor`

An object that appends video samples to an asset writer input.

`class AVAssetWriterInputTaggedPixelBufferGroupAdaptor`

An object that appends tagged buffer groups to an asset writer input.

`class AVAssetWriterInputMetadataAdaptor`

An object that appends timed metadata groups to an asset writer input.

`class AVAssetWriterInputGroup`

A group of inputs with tracks that are mutually exclusive to each other for playback or processing.