Sample Code

Playing immersive media with RealityKit

Create an immersive video playback experience with RealityKit.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

This sample shows how to build an immersive video playback experience for visionOS. It demonstrates the use of RealityKit to present multiple content types, in both windows and immersive spaces. It also presents some key factors to consider as you customize your app experience.

> Note
>
> This sample code project is associated with WWDC25 session 296: Support immersive video playback in visionOS apps.

Choose a playback approach

When it comes to providing immersive video playback on visionOS, there are a few different approaches you can take:

- 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, as it requires the least effort to adopt. For more information on using AVKit in visionOS, see Adopting the system player interface in visionOS.

- RealityKit enables immersive video playback with `VideoPlayerComponent`. RealityKit manages changes in immersive viewing mode to preserve motion comfort, but it requires a little more effort to customize playback controls for your experience.

If you're already using RealityKit in your app, `VideoPlayerComponent` may be suitable, particularly if your video content is relatively short. In that case, it may not be necessary to offer controls for skipping, or to transition between full immersion and portal window viewing. For design guidance, see Human Interface Guidelines > Playing video.

Configure video player to play immersive media

`VideoPlayerComponent` relies on three pairs of properties to play immersive media. For each pair, one property is used for mutation, and another for introspection.

Mutation	Introspection
`desiredImmersiveViewingMode`	`immersiveViewingMode`
`desiredSpatialVideoMode`	`spatialVideoMode`
`desiredViewingMode`	`viewingMode`

The following code configures a video player to present spatial video within a portal window:

```
var videoPlayerComponent = VideoPlayerComponent(avPlayer: player)

videoPlayerComponent.desiredImmersiveViewingMode = .portal
videoPlayerComponent.desiredSpatialVideoMode = .spatial
videoPlayerComponent.desiredViewingMode = .stereo

entity.components[VideoPlayerComponent.self] = videoPlayerComponent
```

Here, the app uses `desiredImmersiveViewingMode` to render the content as a portal window. It then uses `desiredSpatialVideoMode` to apply head-pose–based treatments, and specifies stereoscopic playback with `desiredViewingMode`.

Size video for the shared space

When presented in an immersive space, `VideoPlayerComponent` automatically manages projection of the video content. In a window, however, additional considerations are necessary to achieve best results.

Because portal-based presentations prefer a 16:9 aspect ratio, the sample uses `aspectRatio(_:contentMode:)`:

```
VideoPlayerView(videoModel: selection)
    .aspectRatio(CGSize(width: 16, height: 9), contentMode: .fit)
```

The sample uses the `Entity` that contains the `VideoPlayerComponent` to scale the player to fit within the default scene size. When correctly configured, the spatial video has feathered edges with rounded corners.

The sample uses a `GeometryReader3D` to determine the scene size. It then calls `scaleToFit(_:proxy:content:)` from both the `make` and `update` closures of the root `RealityView`, which ensures that the video properly resizes with the content window.

```
GeometryReader3D { geometry in
    RealityView { content in
        configureContent(content, playbackScene: appModel.playbackScene)
        scaleToFit(videoEntity, proxy: geometry, content: content)
        content.add(rootEntity)
    } update: { content in
        scaleToFit(videoEntity, proxy: geometry, content: content)
    }
```

The `scaleToFit(_:proxy:content:)` scales the `playerScreenSize` to fit the size of the containing scene.

```
func scaleToFit(_ entity: Entity, proxy: GeometryProxy3D, content: RealityViewConter
    guard let videoPlayer = videoEntity.videoPlayerComponent, videoPlayer.needsScali
        return
    }

    let frame = proxy.frame(in: .local)
    let frameSize = abs(content.convert(frame.size, from: .local, to: .scene))
    entity.scaleToFit(videoPlayer.playerScreenSize, within: frameSize)
}
```

Customize playback controls

The sample provides custom playback controls with three basic functions:

1. A toggle for controlling Play and Pause.

2. Immersion toggle.

3. Exit, when in an immersive space.

The app uses the same view, `TransportView`, regardless of whether playback occurs in a window or an immersive space.

For playback in the *Shared Space*, ornaments are ideal: they attach to windows without obscuring the content within. For more information, see Present common controls in an ornament.

```
.ornament(attachmentAnchor: .scene(.bottom)) {
    TransportView()
}
```

The *Full Space* presentation uses the same `TransportView`, but it's incorporated through composition instead. It is added to an outer type, `ImmersiveControlsView`, which is then placed within a `ViewAttachmentComponent` for use in a `RealityView`:

```
private func updateImmersiveControls(with mitigation: VideoPlayerComponent.VideoComf
    let controlsAttachment = ViewAttachmentComponent(rootView: ImmersiveControlsView
    immersiveControls.components.set(controlsAttachment)
}
```

Note

For details regarding managing changes in immersive viewing mode, see Playing immersive video.

Preserve motion comfort

Because scenes with high motion can lead to motion discomfort, be mindful of motion comfort when presenting media immersively. Two key considerations include: configuring the immersive space properly, and responding to video comfort mitigation events in a timely fashion.

When presenting your content in an `ImmersiveSpace`, use a progressive `ImmersionStyle` so that a person can turn the Digital Crown to adjust the amount of visible passthrough video:

```
PlayerImmersiveSpace(sceneIdentifier: Self.sceneID)
    .immersionStyle(
        selection: .constant(ProgressiveImmersionStyle(immersion: 0.01...1, initialA
        in: .progressive
    )
```

The VideoPlayerEvents.VideoComfortMitigationDidOccur event includes a single property, comfortMitigation. This event indicates that the system detected high motion and took steps to preserve motion comfort based on the person's preference. The following shows how the sample app subscribes to this event:

```
_ = content.subscribe(
    to: VideoPlayerEvents.VideoComfortMitigationDidOccur.self,
    on: entity
) { event in
    areTransportControlsVisible = true
    updateImmersiveControls(with: event.comfortMitigation)
}
```

The sample uses this event to advise the person that system mitigation took place, and updates ImmersiveControlsView accordingly.

See Also

Video playback

{} Destination Video

Leverage SwiftUI to build an immersive media experience in a multiplatform app.

{} Displaying video from connected devices

Show video from devices connected with the Developer Strap in your visionOS app.

{} Rendering stereoscopic video with RealityKit

Render stereoscopic video in visionOS with RealityKit.

{} Creating a multiview video playback experience in visionOS

Build an interface that plays multiple videos simultaneously and handles transitions to different experience types gracefully.

Configuring your app for media playback

Configure apps to enable standard media playback behavior.

Adopting the system player interface in visionOS

Provide an optimized viewing experience for watching 3D video content.

Controlling the transport behavior of a player

Play, pause, and seek through a media presentation.

Monitoring playback progress in your app

Observe the playback of a media asset to update your app’s user-interface state.

Trimming and exporting media in visionOS

Display standard controls in your app to edit the timeline of the currently playing media.