Article

Adopting the system player interface in visionOS

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

Overview

The recommended way to provide a video playback interface for your visionOS app is to adopt `AVPlayerViewController`. Using this class makes it simple to provide the same playback user interface and features found in system apps like TV and Music. It also provides essential system integration to deliver an optimal viewing experience whether you're playing standard 2D content or immersive 3D video with spatial audio. This article describes best practices for presenting the player in visionOS and covers the options the player provides to customize its user interface to best fit your app.

> **Note**
>
> In addition to providing the system playback interface, you can also use `AVPlayerViewController` to present a media-trimming experience similar to QuickTime Player in macOS. See Trimming and exporting media in visionOS for more information.

Explore presentation options

Use `AVPlayerViewController` to play video in windowed environments in visionOS. It automatically adapts its user interface to best fit its presentation. For example, when you present it nested inside another view, it displays an inline user interface:

> **Note**
>
> When you present the player inline, it only displays standard 2D video. To play 3D content, present it fullscreen.

Present the player in full-screen mode by setting it as the exclusive root view of your app, or by presenting it using the `fullScreenCover(item:onDismiss:content:)` modifier. In full-screen mode, the player presents a more content-forward design that dims the environment by default to provide more suitable viewing. This provides a streamlined viewing experience for both 2D and 3D content.

Display supporting metadata

The user interface displays a title view above the transport bar when the current player item contains title and subtitle metadata. When playing live-streaming content, the title view may also display a badge to indicate the content state to the viewer.

The title view displays the values of an asset's `commonIdentifierTitle` and `iTunesMetadataTrackSubTitle` metadata items, when available. If your media doesn't provide embedded metadata, you can add supplemental metadata to display by creating instances of `AVMetadataItem`. The table below lists the metadata values the player user interface supports.

Metadata	Identifier	Type
Title	`commonIdentifierTitle`	String
Subtitle	`iTunesMetadataTrackSubTitle`	String
Artwork	`commonIdentifierArtwork`	Data
Description	`commonIdentifierDescription`	String
Genre	`quickTimeMetadataGenre`	String
Content rating	`iTunesMetadataContentRating`	String

In an app that defines a simple structure to hold string and data items, you can map its values to their appropriate metadata identifiers and build an array of metadata items:

```
private func createMetadataItems(for metadata: Metadata) -> [AVMetadataItem] {
    [
        metadataItem(key: .commonIdentifierTitle, value: metadata.title),
        metadataItem(key: .iTunesMetadataTrackSubTitle, value: metadata.subtitle),
        metadataItem(key: .commonIdentifierArtwork, value: metadata.imageData),
        metadataItem(key: .commonIdentifierDescription, value: metadata.description),
        metadataItem(key: .iTunesMetadataContentRating, value: metadata.rating),
        metadataItem(key: .quickTimeMetadataGenre, value: metadata.genre)
    ]
}

private func metadataItem(key identifier: AVMetadataIdentifier,
                          value: Any) -> AVMetadataItem {
    let item = AVMutableMetadataItem()
    item.identifier = identifier
    item.value = value as? NSCopying & NSObjectProtocol
    item.extendedLanguageTag = "und"
    // Return an immutable copy of the item.
    return item.copy() as! AVMetadataItem
}
```

To apply the metadata to the current player item, set the array of metadata items as the value of the player item's `externalMetadata` property:

```
let metadata: Metadata = // A structure that contains simple string values.
playerItem.externalMetadata = createMetadataItems(for: metadata)
```

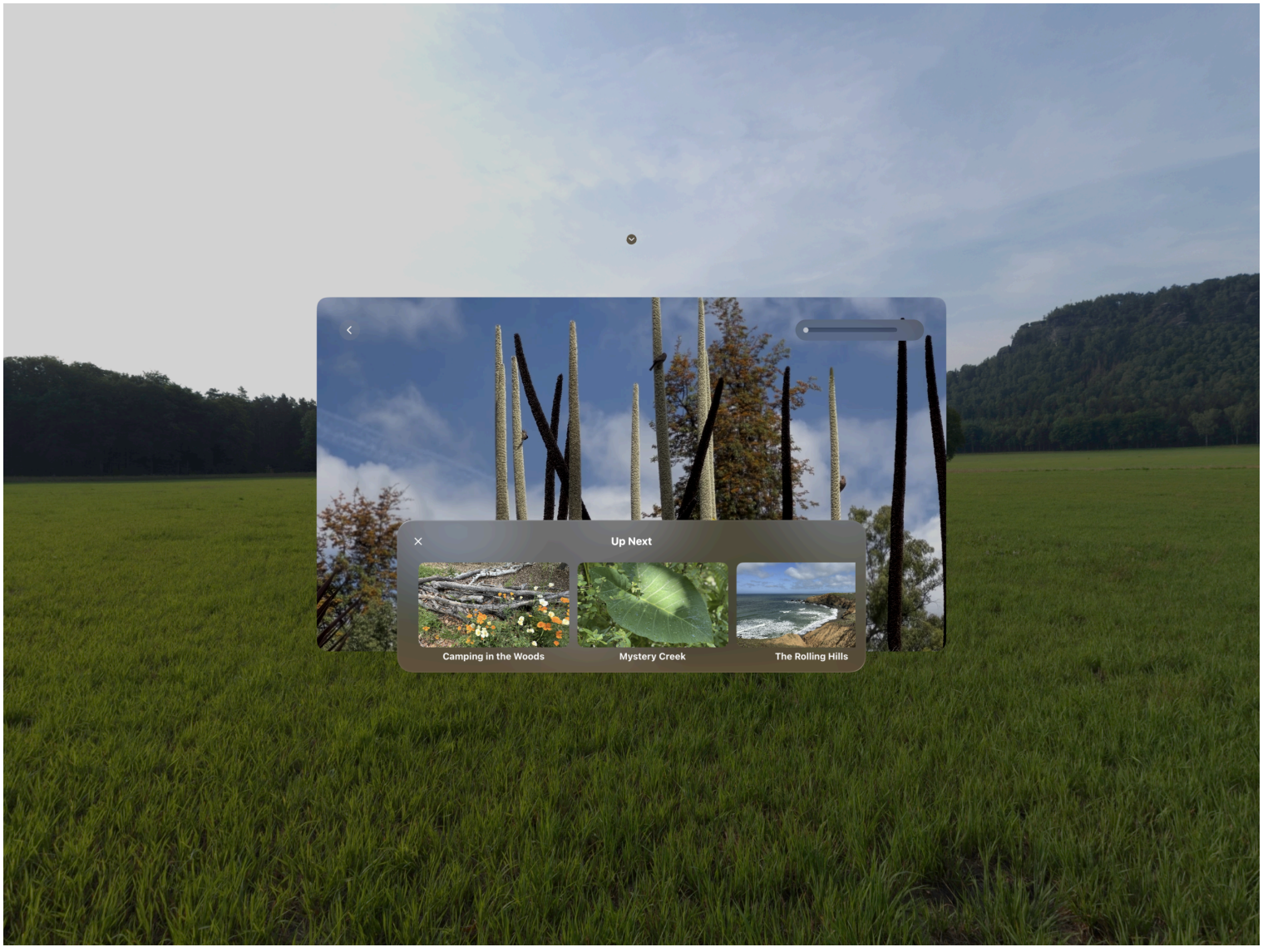
Only the title and subtitle values display in the title view. The player presents the other supported metadata values in its Info tab, which the section below describes.

Display custom informational views

The visionOS player UI can display one or more content tabs in the user interface to show supporting information or related content. By default, the player presents an Info tab when an asset contains embedded metadata or when you set external metadata on the player item, as the Display supporting metadata section above describes.

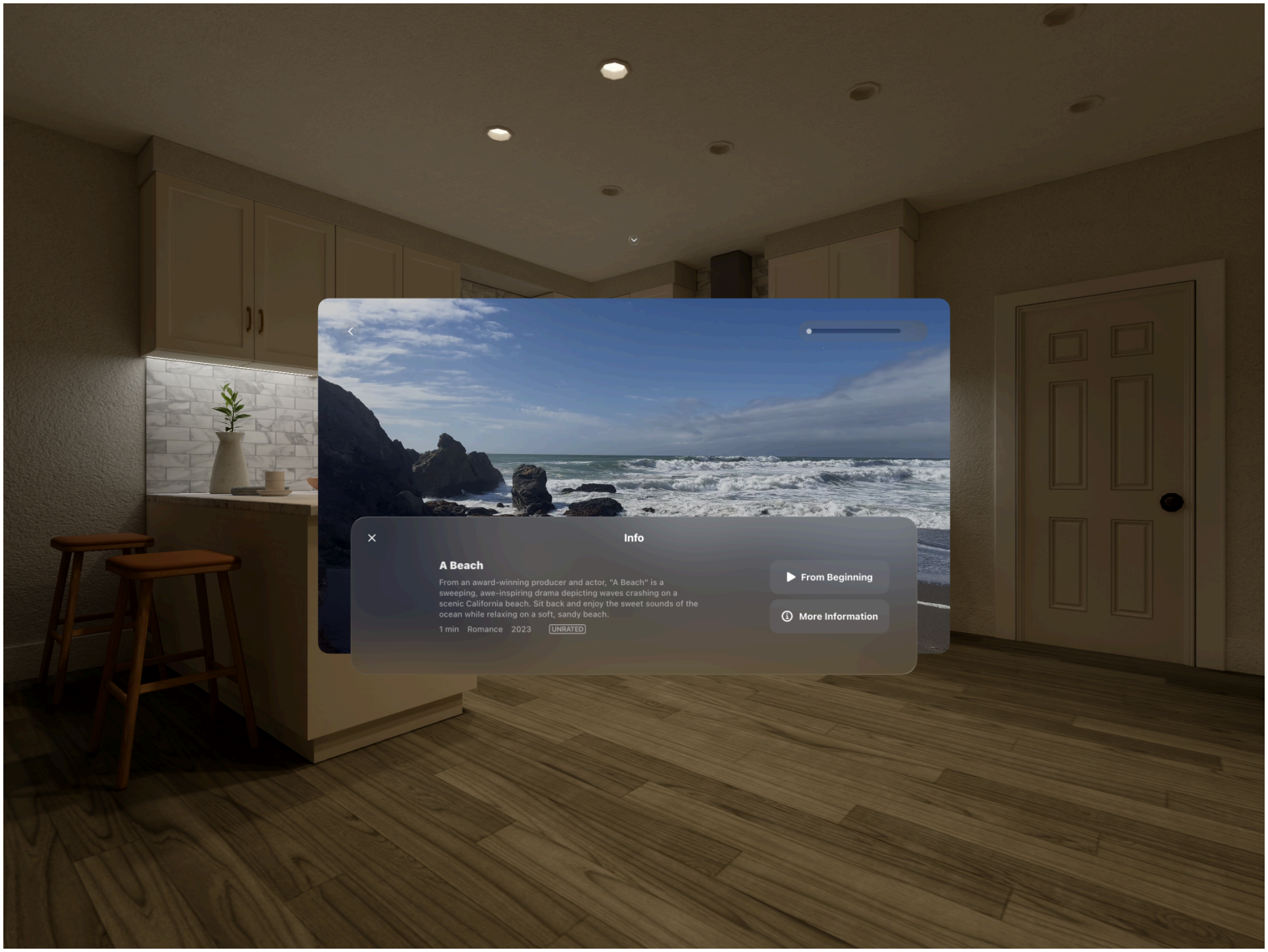
Your app can also present custom tabs to show supporting content. You define your tab content as standard SwiftUI views, wrap them in a `UIHostingController`, and set them as the `customInfoViewControllers` property. The player UI uses the `title` property of the hosting controller to display as the tab title in the interface, so set this value before setting it on the player view controller.

```
let view = UpNextView(videos: upNextVideos)
let hostingController = UIHostingController(rootView: view)

// Set custom content tabs on the player UI.
hostingController.title = String(localized: "Up Next")

hostingController.preferredContentSize = CGSize(width: 300, height: 100)
controller.customInfoViewControllers = [hostingController]
```



Present actions in the Info tab

The player UI presents an Info tab when the asset it displays provides embedded or external metadata. The tab's view displays the metadata details, and it may show up to two `UIAction` controls along its trailing edge:



Customize the actions the view presents by setting a value for the player view controller's `infoViewActions` property. When playing nonlive content, this property contains a single-element array that presents an action to play the content from the beginning. You can replace the default value (if present), add an additional action, or set this property value to an empty array to display no actions. The example below shows how to add a Add to Favorites action to the view:

```
let infoCircle = UIImage(systemName: "info.circle")
let showMoreInfo = UIAction(title: "More Information", image: infoCircle) { action i
    // Navigate to a screen to display more information.
}
// Append the action to the array.
playerViewController.infoViewActions.append(showMoreInfo)
```

Display actions contextually

You can use the visionOS player UI to present controls contextually, which your app displays for a specific range of time in the content and then dismiss. A common use for this type of control is a skip button that displays during the title sequence of a movie or TV show. People can tap the button to bypass the introduction and quickly skip to the main content.

`AVPlayerViewController` provides a `contextualActions` property you can use to specify one or more actions to present. The player displays them along the bottom-trailing side of the screen. The following code example shows a simple implementation of an action that seeks the player forward to the time of the main content:

```
// Define an action to skip the intro of a media item.
private lazy var skipActions: [UIAction] = {
    [UIAction(title: "Skip Intro") { [weak self] _ in
        guard let self else { return }
        avPlayer.seek(to: skipToTime)
    }]
}()
```

When you set a value for the `contextualActions` property, the player presents the controls immediately. To present them only during a relevant section of the content, observe the player timing by adding a periodic or boundary time observer. The following example defines a periodic time observer that fires every second during normal playback. In each invocation, it evaluates the new time to determine whether it falls within the presentation range. If it does, the example sets the skip action as the contextual actions value; otherwise, it clears the value by setting it to an empty array.

```
private func addTimeObserver() {
    // Observe the player's timing every second.
    let interval = CMTime(value: 1, timescale: 1)
    let fifteenSeconds = CMTime (value: 15, timescale: 1)
    timeObserver = avPlayer.addPeriodicTimeObserver(forInterval: interval,
                                                    queue: .main) { [weak self] time
        guard let self else { return }
        let duration = avPlayer.currentItem?.duration ?? .zero
        // Show the Skip Intro button during the first 15 seconds of the content.
        showSkipIntroAction = time <= fifteenSeconds
    }
}
```

See Also

visionOS playback

{} Playing immersive media with AVKit

Adopt the system playback interface to provide an immersive video watching experience.

{} Creating a multiview video playback experience in visionOS

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

Trimming and exporting media in visionOS

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

`class` `AVPlayerViewController`

A view controller that displays content from a player and presents a native user interface to control playback.

`protocol` `AVPlayerViewControllerDelegate`

A protocol that defines the methods to implement to respond to player view controller events.

`class` `AVExperienceController`

An object that controls video experiences.

`class` `AVMultiviewManager`

An object that manages viewing multiple videos at once.

`class` `AVGroupExperienceCoordinator`

An object that synchronizes viewing environment state across participants in a SharePlay session.