Sample Code

Playing video content in a standard user interface

Play media full screen, embedded inline, or in a floating Picture in Picture (PiP) window using a player view controller.

Download

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | Xcode 13.4+

Overview

AVKit is a cross-platform media playback UI framework built on top of AVFoundation and Core Media. It makes it easy to play `AVPlayer`-based media content using the same user interface as Apple's own apps. For UIKit apps, AVKit provides `AVPlayerViewController`, a view controller that displays content from a player and presents a native user interface to control playback.

This sample app demonstrates three display options for media playback using `AVPlayerViewController`: full screen, embedded inline, or in a floating PiP window.

The sample uses `AVPlayerViewController` in full-screen playback mode to scale the video to fill the display, enabling a distraction-free environment that hides the system and app controls until people take action to reveal them. To demonstrate video inline playback, the sample embeds the `AVPlayerViewController` view in the app's user interface. The sample also uses `AVPlayerViewController` to play video in PiP mode, where the video remains in view in a floating video overlay while the user interacts with other apps. The user manages the player using the standard player interface.

Getting started with `AVPlayerViewController` is straightforward. You create an `AVPlayer`, and then create an `AVPlayerViewController` and assign the player to it. And finally, you present the `AVPlayerViewController`. When playing full screen, embedded inline, or in a floating PiP window, you implement callback methods to respond to the various `AVPlayerViewController` events.

> **Note**
>
> This sample code project is associated with WWDC 2019 session 503: Delivering Intuitive Media Playback with AVKit.

Create and configure the player view controller

The sample's `loadPlayerViewControllerIfNeeded` function creates an `AVPlayerViewController` that it uses to play the videos in the various playback modes.

```
private func loadPlayerViewControllerIfNeeded() {
    if playerViewControllerIfLoaded == nil {
        playerViewControllerIfLoaded = AVPlayerViewController()
    }
}
```

The sample implements the `AVPlayerViewControllerDelegate` methods to respond to player view controller events. This allows the sample to handle the app's user interface based on the player view controller state, along with observing for potential errors. To receive notifications of the player view controller events, the project's `PlayerViewControllerCoordinator` assigns itself as the player view controller delegate.

```
playerViewController.delegate = self
```

A player view controller requires an `AVPlayer` object to provide the media content to display. The `AVPlayer` plays media assets that AVFoundation models using the `AVAsset` class, which represent the media to play. However, an `AVAsset` only models the static aspects of the media, such as its duration or creation date, and on its own, is unsuitable for playback with an `AVPlayer`. To play an asset, the sample creates an instance of its dynamic counterpart, `AVPlayerItem`. This object models the timing and presentation state of an asset that an instance of `AVPlayer` plays. The sample creates an `AVPlayer` from the `AVPlayerItem`, and assigns the `AVPlayer` to the `AVPlayerViewController`.

```
if !playerViewController.hasContent(fromVideo: video) {
    let playerItem = AVPlayerItem(url: video.hlsUrl)
    playerViewController.player = AVPlayer(playerItem: playerItem)
}
```

Play media full screen

When the user taps on one of the app's views to play video full screen, the sample calls the `present(_:animated:completion:)` method to present the video full screen modally, not as a subview controller of some other view controller. The sample uses the default modal presentation style `UIModalPresentationStyle.automatic`, which resolves to a full-screen presentation. To begin playback, the sample calls the `AVPlayerViewController` player's `play()` method.

```
guard let playerViewController = playerViewControllerIfLoaded else { return }
presentingViewController.present(playerViewController, animated: true) {
    playerViewController.player?.play()
}
```

Handle player view controller full-screen events

The sample implements the `playerViewController(_:willBeginFullScreenPresentationWithAnimationCoordinator:)` delegate method to receive notifications when the `AVPlayerViewController` is about to start displaying its contents full screen. This delegate method passes the player view controller and transition coordinator to use for coordinating animations. When the sample presents or dismisses a view controller, UIKit creates a transition coordinator object automatically and assigns it to the view controller's `transitionCoordinator` property. The transition coordinator object only lasts for the duration of the transition animation.

The sample calls the transition coordinator's `animate(alongsideTransition:completion:)` method to run the animations at the same time as the view controller transition animations. The sample also implements the `animate(alongsideTransition:completion:)` method's completion handler that executes after the transition finishes. In the completion handler, the sample updates the playback state string that displays in the content overlay view on top of the player view controller. The sample also checks whether the transition succeeds or the user cancels it. If it succeeds, the sample saves a strong reference to the player view controller. The sample uses this reference to dismiss any active player view controllers before restoring the app's interface when PiP stops.

```
func playerViewController(
    _ playerViewController: AVPlayerViewController,
    willBeginFullScreenPresentationWithAnimationCoordinator coordinator: UIViewContr
    ) {
    status.insert([.fullScreenActive, .beingPresented])

    coordinator.animate(alongsideTransition: nil) { context in
```

```
        self.status.remove(.beingPresented)
        // Check context.isCancelled to determine whether the transition is successf
        if context.isCancelled {
            self.status.remove(.fullScreenActive)
        } else {
            // Keep note of the view controller that the system uses to present full
            self.fullScreenViewController = context.viewController(forKey: .to)
        }
    }
}
```

The sample implements the `playerViewController(_:willEndFullScreenPresentationWithAnimationCoordinator:)` delegate method to receive notifications when the `AVPlayerViewController` is about to stop displaying its contents full screen. In this method, the sample also calls the transition coordinator's `animate(alongsideTransition:completion:)` method to run the animations at the same time as the view controller transition animations. The sample implements the `animate(alongsideTransition:completion:)` method's completion handler to update the debug string that displays in the content overlay view on top of the player view controller.

```
func playerViewController(
    _ playerViewController: AVPlayerViewController,
    willEndFullScreenPresentationWithAnimationCoordinator coordinator: UIViewControl
    ) {
    status.insert([.beingDismissed])
    delegate?.playerViewControllerCoordinatorWillDismiss(self)

    coordinator.animate(alongsideTransition: nil) { context in
        self.status.remove(.beingDismissed)
        if !context.isCancelled {
            self.status.remove(.fullScreenActive)
        }
    }
}
```

Display custom overlays in the player view controller

`AVPlayerViewController` provides a `contentOverlayView` property for adding noninteractive custom views, such as a logo or watermark, between the video content and the controls.

The sample creates a custom view DebugHUD for displaying the current playback state (embedded inline, full-screen active, and so on) of a video playback item. The sample's addDebugHUDToPlayerViewControllerIfNeeded function adds this custom view to the contentOverlayView.

```
private func addDebugHUDToPlayerViewControllerIfNeeded() {
    if status.contains(.embeddedInline) || status.contains(.fullScreenActive) {
        if let playerViewController = playerViewControllerIfLoaded,
            let contentOverlayView = playerViewController.contentOverlayView,
            !debugHud.isDescendant(of: contentOverlayView) {
            playerViewController.contentOverlayView?.addSubview(debugHud)
```

The sample's PlayerViewControllerCoordinator declares the status variable that maintains the current playback state.

```
private(set) var status: Status = [] {
    didSet {
        debugHud.status = status
        externalDebugHud.status = status
        if oldValue.isBeingShown && !status.isBeingShown {
            playerViewControllerIfLoaded = nil
        }
        addDebugHUDToPlayerViewControllerIfNeeded()
    }
```

The PlayerViewControllerCoordinator updates the playback state in the DebugHUD view in response to player view controller events and other state changes. For example, to receive notifications when the player view controller video frames are ready for display, the sample observes the player view controller's isReadyForDisplay property. When the property changes, the PlayerViewControllerCoordinator updates the status variable to reflect the current playback state.

```
readyForDisplayObservation = playerViewController.observe(\.isReadyForDisplay) { [we
    if observed.isReadyForDisplay {
        self?.status.insert(.readyForDisplay)
    } else {
        self?.status.remove(.readyForDisplay)
    }
}
```

Play media inline

The sample's `embedInline` function incorporates the player view controller's view into the app's view hierarchy for inline playback. To do this, the function first checks whether an `AVPlayerViewController` object already exists in the view hierarchy, and if so, removes it. Next, the function adds the `AVPlayerViewController` as a subview of the current view controller. After that, it adds the `AVPlayerViewController` `view` to the specified containing view so that it resides on top of any subviews. Lastly, the function calls the view controller `didMove(toParent:)` function. Container view controller subclasses need to call `didMove(toParent:)` after a transition to the new subview completes or, in the case of no transition, immediately after the call to `addChild(_:)`.

The user manages inline playback using the standard player interface.

```
func embedInline(in parent: UIViewController, container: UIView) {
    loadPlayerViewControllerIfNeeded()
    guard let playerViewController = playerViewControllerIfLoaded, playerViewContro
    removeFromParentIfNeeded()
    status.insert(.embeddedInline)
    parent.addChild(playerViewController)
    container.addSubview(playerViewController.view)
    playerViewController.view.translatesAutoresizingMaskIntoConstraints = false
    NSLayoutConstraint.activate([
        playerViewController.view.centerXAnchor.constraint(equalTo: container.cente
        playerViewController.view.centerYAnchor.constraint(equalTo: container.cente
        playerViewController.view.widthAnchor.constraint(equalTo: container.widthAn
        playerViewController.view.heightAnchor.constraint(equalTo: container.height
    ])
    playerViewController.didMove(toParent: parent)
}
```

Configure audio session and background modes for PiP

To use PiP, the sample configures its audio session and background modes. For more information, see Configuring your app for media playback. After this configuration, the player view controller automatically supports PiP playback.

Handle PiP player view controller events

To receive notifications when PiP is about to start, or fails to start, the sample implements the delegate methods `playerViewControllerWillStartPictureInPicture(_:)` and `playerView(_:failedToStartPictureInPictureWithError:)`, respectively. To receive

notifications when PiP stops, the sample implements the `playerViewControllerDidStopPictureInPicture(_:)` method.

Each of the sample's `AVPlayerViewControllerDelegate` method implementations updates the `DebugHUD` custom view to reflect the current playback state.

```
func playerViewControllerWillStartPictureInPicture(_ playerViewController: AVPlayerV
    status.insert(.pictureInPictureActive)
}

func playerViewControllerDidStopPictureInPicture(_ playerViewController: AVPlayerVie
    status.remove(.pictureInPictureActive)
}

func playerViewController(_ playerViewController: AVPlayerViewController, failedToSt
    status.remove(.pictureInPictureActive)
}
```

Restore the video playback interface when PiP stops

To handle the restore process when PiP stops, the sample implements the `playerView(_:restoreUserInterfaceForPictureInPictureStopWithCompletionHandler:)` method. The framework calls this method when control returns to the app, giving the app the opportunity to determine how to properly restore its video playback interface. The sample sends the callback up to its own delegate to handle the restore operation.

```
func playerViewController(
    _ playerViewController: AVPlayerViewController,
    restoreUserInterfaceForPictureInPictureStopWithCompletionHandler completionHandl
    ) {
    if let delegate = delegate {
        delegate.playerViewControllerCoordinator(self, restoreUIForPIPStop: completi
    } else {
        completionHandler(false)
    }
}
```

See Also

iOS playback and capture

`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` `AVCaptureEventInteraction`

An object that registers handlers to respond to capture events from system hardware buttons.

`class` `AVCaptureEvent`

An object that describes a user interaction with a system hardware button.

`class` `AVCaptureEventSound`

A sound object for a capture event.

`class` `AVInputPickerInteraction`

Use `AVInputPickerInteraction` to present an input picker.