Sample Code

Supporting remote interactions in tvOS

Set up your app to support remote commands and events in a variety of scenarios by using the relevant approach.

Download

tvOS 16.0+ | Xcode 14.3+

Overview

This sample code project demonstrates how to implement support for the wide variety of commands that remotes and system controls send to Apple TV.

There are four sections in this project. The first section demonstrates how to add native media playback capabilities to your app using the system player that `AVPlayerViewController` provides. It also shows how to use `MPRemoteCommandCenter` to support additional commands in a system player, and use `AVPlayerViewControllerDelegate` to customize the behavior of commands the player receives.

Using the system player is a great choice in most cases. However, if you need to implement a player with a custom UI, you also need to implement support for all the commands the system player supports. The second section shows you how, by using `MPRemoteCommandCenter` and gesture recognizers. It also demonstrates how to handle receiving remote events and update the Now Playing info display.

The third section displays a screen that visualizes remote commands as the player receives them. The fourth section shows you how to page content in a guide that displays channels and programs.

Support additional commands in the system player

You can automatically get support for most controls to play and navigate media by using `AVPlayerViewController`. To provide support for additional remote control events, such as previous and next track commands, use the shared `MPRemoteCommandCenter` object. In the following example, `additionalRemoteCommands` contains the previous and next track commands registered with the remote command center, and `handleCommand(_:)` performs the corresponding action for the command it receives.

```
private func setupAdditionalRemoteCommands() {
    additionalRemoteCommands.forEach { [weak self] remoteCommand in
        guard let self = self else { return }
        // Remove each target before you add a new one.
        remoteCommand.mediaRemoteCommand.removeTarget(nil)
        remoteCommand.mediaRemoteCommand.addTarget { _ in
            self.handleCommand(remoteCommand)
        }
    }
}
```

You can also customize the behavior of commands that `AVPlayerViewController` natively supports by conforming to `AVPlayerViewControllerDelegate`. For example, this code maps the `skipToPreviousChannel` event to the `channelDown` action:

```
func playerViewController(_ playerViewController: AVPlayerViewController,
                          skipToPreviousChannel completion: @escaping (Bool) -> Voic
    channelDown()
    completion(true)
}
```

Support all commands in a custom player and update Now Playing info

When you create a custom player, use `MPRemoteCommandCenter` to implement support for the full suite of commands and events a remote can send to it. The following example shows how to customize the behavior for changing playback rate and skipping backward or forward. It also shows how to add targets to handle all other supported commands:

```
for supportedCommand in supportedRemoteCommands {
    switch supportedCommand {
        // Define the rates and intervals for the commands that require them.
    case .changePlaybackRate:
        let allSupportedRates = rewindSupportedRates + fastForwardSupportedRates
        commandCenter
            .changePlaybackRateCommand
            .supportedPlaybackRates = allSupportedRates.map {
                $0 as NSNumber
            }

    case .skipBackward:
        commandCenter
            .skipBackwardCommand
            .preferredIntervals = [skipInterval as NSNumber]

    case .skipForward:
        commandCenter
            .skipForwardCommand
            .preferredIntervals = [skipInterval as NSNumber]

    default:
        break
    }

    // Remove each target before you add a new one.
    supportedCommand.mediaRemoteCommand.removeTarget(nil)
    supportedCommand.mediaRemoteCommand.addTarget {
        self.handle(command: supportedCommand, withCommandEvent: $0)
    }
}
```

Additionally, this section demonstrates using the default `MPNowPlayingInfoCenter` object to update the Now Playing info:

```
var nowPlayingInfo = [String: Any]()

nowPlayingInfo[MPMediaItemPropertyTitle] = currentProgram.title
nowPlayingInfo[MPMediaItemPropertyPlaybackDuration] = currentItem.duration.seconds
nowPlayingInfo[MPNowPlayingInfoPropertyIsLiveStream] = currentProgram.isLive
nowPlayingInfo[MPNowPlayingInfoPropertyAssetURL] = currentProgram.playlistURL
nowPlayingInfo[MPNowPlayingInfoPropertyPlaybackRate] = self.player.rate
nowPlayingInfo[MPNowPlayingInfoPropertyDefaultPlaybackRate] = self.defaultPlaybackRa
nowPlayingInfo[MPNowPlayingInfoPropertyElapsedPlaybackTime] = self.player.currentTim

// Set any other properties that are applicable to your application.
```

```
MPNowPlayingInfoCenter.default().nowPlayingInfo = nowPlayingInfo
```

Visualize receiving remote events

The third section of this project includes a screen that displays visual feedback of user interactions with a remote. It does this by reporting remote events that the view controller receives from a remote to the view so it can blink the corresponding cell:

```
private func reportRemoteEvent(_ remoteEvent: RemoteEvent,
                               withState state: UIGestureRecognizer.State? = nil) {
    guard let remoteEventsView = self.view as? RemoteEventsView else { return }
    remoteEventsView.remoteEventReceived(remoteEvent, withState: state)
}
```

Page content in a guide

If your app displays a guide that lists channels and their corresponding programs, the fourth section of this project shows you how to page content in response to remote commands, such as to scroll to the previous page for a channel up press:

```
@objc private func channelUpPressed() {
    // Because of the defined height of the cells, `indexPathsForVisibleItems`
    // has a maximum of 14 items on screen at any given time.
    guard let targetItemInPreviousPageSectionIdx = collectionView
        .indexPathsForVisibleItems.sorted().first?.section else { return }

    // For each page section, channel cell is always item 0 and program
    // cells start from 1.
    let firstProgramItemIdx = 1
    let targetItemIdx = currentlyFocusedIndexPath?.item ?? firstProgramItemIdx
    channelUpDownTargetIndexPath = IndexPath(
        item: targetItemIdx,
        section: targetItemInPreviousPageSectionIdx
    )
    // Scroll to the previous page for a channel up press.
    channelUpDownScrollDirection = .up

    setNeedsFocusUpdate()
    updateFocusIfNeeded()
}
```