Sample Code

Supporting Continuity Camera in your tvOS app

Capture high-quality photos, video, and audio in your Apple TV app by connecting an iPhone or iPad as a continuity device.

Download

tvOS 17.0+ | Xcode 15.3+

Overview

Continuity Camera brings the power of the cameras and microphones from an iOS or iPadOS device to Apple TV, including advanced features like Center Stage and Portrait mode.

This sample project provides an example implementation that accesses a camera and microphone from a nearby iPhone or iPad in an Apple TV app. It builds on a similar sample, Supporting Continuity Camera in your macOS app, and shares some of its functionality, including automatic camera selection and observing the state of video effects. The following sections focus on the aspects specific to tvOS.

> **Note**
>
> This sample code project is associated with WWDC23 session 10256: Discover Continuity Camera on tvOS.

Configure the sample code project

To run this sample app, you need the following:

- An Apple TV 4k (2nd generation) or later with tvOS 17 or later.

- An iPhone or iPad with iOS 17 or iPadOS 17, respectively, or later.

You need to run this sample code project on physical devices, because Simulator doesn't include the components to support cameras.

Continuity Camera works with all iPhone and iPad models that support video effects in Control Center. You need to sign in with an Apple ID that uses two-factor authentication for the Apple TV and the device with a camera. You can use a separate Apple ID for each device or the same Apple ID for both.

The first time you run the app on an Apple TV, the system prompts you for permission to access to the camera and microphone. The app needs these permissions to function correctly.

Present the continuity device picker

When the app launches, it immediately presents a continuity device picker by calling the `continuityDevicePicker(isPresented:onDidConnect:)` modifier in its SwiftUI implementation.

```
.continuityDevicePicker(isPresented: $showContinuityDevicePicker,
                        onDidConnect: handleNewConnectionForDevice)
.task {
    // Shows the picker when app has no continuity device at launch.
    if !captureManager.activateDefaultContinuityCameraDevice() {
        showContinuityDevicePicker = true
    }
```

The picker only appears if the `isPresented` parameter — which is a Boolean `Binding` — is `true`. The picker calls the closure the app passes to the `onDidConnect` parameter when a person selects a device and the system successfully connects to it.

```
func handleNewConnectionForDevice(_ device: AVContinuityDevice?) {
    guard let device else {
        print("The Continuity Device Picker didn't connect a device.")
        return
    }

    guard let firstCamera = device.videoDevices.first else {
        print("The Continuity Device Picker doesn't have any cameras.")
        return
    }
```

```
    captureManager.setActiveVideoInput(firstCamera,
                                       isUserPreferredCamera: true)
}
```

The handling closure's `AVContinuityDevice` parameter represents the device that a person selects on their Apple TV. Each continuity device has a `videoDevices` property, which is an array of `AVCaptureDevice` instances.

The app's `handleNewConnectionForDevice(_:)` method is a minimal implementation that selects the first video device in the array. Apps typically compare all the video device elements and select one that's appropriate for their needs.

> **Note**
>
> `UIKit` based apps can create a continuity device picker by creating an `AVContinuityDevicePickerViewController` instance.

Connect a video device to a capture session

The app's `CaptureManager` class creates and maintains an `AVCaptureSession` instance for the app's lifetime. The capture manager's `setActiveVideoInput(_:)` method creates an `AVCaptureDeviceInput` instance from the video device, and then tests to see whether it's an acceptable input for the capture session.

```
let name = camera.localizedName
print("Setting video input to: \(name).")

// Creates a video input with the camera.
guard let videoInput = try? AVCaptureDeviceInput(device: camera) else {
    print("Couldn't make an input from: \(name).")
    return false
}

// Checks whether the capture session accepts the new camera as an input.
guard session.canAddInput(videoInput) else {
    print("Capture session rejected '\(name)' as an input.")
    return false
}

// Adds the new camera input to the capture session.
activeInput = videoInput
```

If the new device is an acceptable input, the method assigns it to the app's `activeInput` property. The property updates the capture session with its `willSet` and `didSet` property observers.

```
internal var activeInput: AVCaptureDeviceInput? {
    willSet {
        if let oldInput = activeInput {
            session.removeInput(oldInput)
        }
    }
    didSet {
        if let newInput = activeInput {
            session.addInput(newInput)
        }
        isActive = (activeInput != nil)
    }
}
```

The `willSet` observer removes the capture session's current input, if applicable. The `didSet` observer adds the new input to the capture session. The `didSet` observer also updates the `isActive` Boolean property, which can cause the app to change its behavior and UI.

Register for capture device updates

The app receives various updates related to its capture device by registering with NotificationCenter and with key-value observation (KVO). See Using Key-Value Observing in Swift and NSKeyValueObserving for more information.

The app specifically registers for the following events:

- A specific video effect, such as Center Stage, changes its active state.
- The system changes the capture device it prefers.
- The active capture device disconnects from the system.

> **Note**
>
> People can enable video effects in Control Center on Apple TV.

The sample's implementation that monitors the video effects and system changes is similar to the macOS equivalent of this sample, Supporting Continuity Camera in your macOS app. The sample

also monitors Notification Center events related to the camera. The app's capture manager responds when a capture device disconnects by registering with Notification Center for the `.AVCaptureDeviceWasDisconnected` event.

```
func observeCamera(_ camera: AVCaptureDevice) {
    // Tells the observer to watch the new camera's properties.
    videoEffectsObvserver.observeCamera(camera)

    // Tells the notification observer to monitor camera-related events.
    notificationObserver.observeCamera(camera,
                                       with: notification(_:for:))
}
```

The app's `CaptureDeviceNotificationObserver` structure listens for the these events on behalf of the capture manager and calls the manager's `notification(_:for:)` method for each event it gets from Notification Center.

Configure the audio engine with an audio input device

At launch, the app creates an `AudioCapturer` instance, which checks for audio inputs (microphones). It does this by inspecting the `AVAudioSession.availableInputs` property of the `AVAudioSession` type's shared instance, and then monitoring the property for updates.

The app monitors for new microphones — similar to how the app's capture manager monitors for new cameras — by observing the `isInputAvailable` property of the `AVAudioSession` type's shared instance.

```
private static let inputAvailableKeyPath = "isInputAvailable"

func registerForInputAvailabilityUpdates(on session: AVAudioSession) {
    session.addObserver(self,
                        forKeyPath: Self.inputAvailableKeyPath,
                        options: [.new],
                        context: nil)
}
```

When the app has access to a microphone, it configures an `AVAudioEngine` instance in the audio capturer's `setupAndStartAudioSession()` method.

```
func setupAndStartAudioSession() {
    configureAudioOutput()
```

```
    enableVoiceProcessing(true)
    configureAudioSessionForVoiceChat()
    startAudioEngine()
}
```

The method configures the audio engine for a conference call scenario when the app gains access to a microphone — at launch or later — with the following steps:

1. Configures the audio engine to produce sound from the system's first audio output.
2. Enables voice processing on the audio engine's input node.
3. Configures the audio engine for conversational audio.
4. Starts the audio engine.

Configure the audio engine for a call

The third step is important for conferencing apps that use Voice over IP (VoIP). The `configureAudioSessionForVoiceChat` method configures the audio session by passing the `.voiceChat` mode to the audio session's `setCategory(_:options:)` method.

```
try avAudioSession.setCategory(.playAndRecord,
                               mode: .voiceChat,
                               options: [])
```

The app gains access to additional audio features and microphone modes, including automatic gain correction, voice processing, and muting, by configuring the audio session for VoIP.

The app's audio interface has a button that lets a person temporarily disable microphone processing, including echo cancellation, by bypassing the audio engine's voice processing. Each time a person toggles the button, the app calls audio capturer's `bypassVoiceProcessing(_:)` method.

```
public func bypassVoiceProcessing(_ bypass: Bool) {
    // If true, temporarily disables echo cancelation.
    avAudioEngine.inputNode.isVoiceProcessingBypassed = bypass

    DispatchQueue.main.async {
        self.isVoiceProcessingBypassed = bypass
    }

    var message = "Audio engine's voice processing: "
```

```
    message += bypass ? "bypassed" : "normal"
    print(message)
}
```

The app can temporarily disable voice processing by setting the `isVoiceProcessingBypassed` property of the audio engine's `inputNode` to `true`. This gives the app all the incoming audio from the microphone without any adjustments from the system.

> **Note**
>
> The behavior of the audio engine's `isVoiceProcessingBypassed` property is similar to `kAUVoiceIOProperty_BypassVoiceProcessing`. For more information, see Audio Unit Voice I/O.

See Also

tvOS playback and capture

- Customizing the tvOS Playback Experience

  Adopt the latest features of the redesigned tvOS player user interface to provide a more streamlined way to watch your content.

- Presenting Navigation Markers

  Present navigation markers in the Chapters panel to help users quickly navigate your content.

- Working with Interstitial Content

  Present additional content alongside your main media presentation using HTTP Live Streaming support.

- Presenting Content Proposals in tvOS

  Display a preview of an upcoming media item at the conclusion of the currently playing media item.

- {} Working with Overlays and Parental Controls in tvOS

  Add interactive overlays, parental controls, and livestream channel flipping using a player view controller.

`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` `AVInterstitialTimeRange`

A time range in an audiovisual presentation for content with an interstitial designation, such as advertisements or legal notices.

`class` `AVNavigationMarkersGroup`

A set of markers for navigating playback of an audiovisual presentation.

`class` `AVContentProposalViewController`

A view controller that proposes content to watch next.

`class` `AVDisplayManager`

A tvOS management object that controls whether a TV switches modes to match the video's native mode.

`class` `AVContinuityDevicePickerViewController`

A view controller that provides an interface to a person so they can select and connect a continuity device to the system.

`protocol` `AVContinuityDevicePickerViewControllerDelegate`

An interface that responds to events from a continuity device picker view controller.