Sample Code

Controlling a DockKit accessory using your camera app

Follow subjects in real time using an iPhone that you mount on a DockKit accessory.

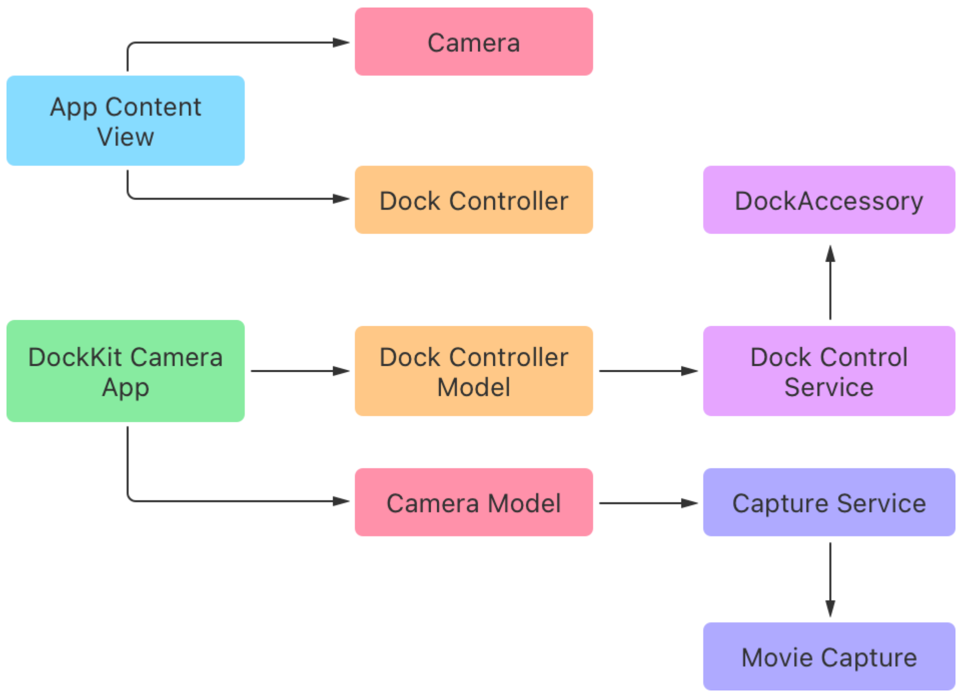
Download

iOS 18.0+ | Xcode 16.1+

Overview

This sample code project shows you how to use your camera app with a DockKit accessory to frame and track subjects in real time. It demonstrates how DockKit system tracking works for your camera app, and how you can override system tracking to frame and track specific subjects using custom machine learning signals. It also shows you how to integrate physical buttons on your DockKit device with camera controls.

The sample uses SwiftUI and the features of Swift concurrency to build a responsive camera app with DockKit control. See AVCam: Building a camera app for more details about the camera implementation design. This sample code project uses the sample app from that project as a starting point to write a basic camera app. The following diagram depicts the app's design:



The sample app defines two key services:

- `CaptureService` is an actor that manages the interactions with the AVFoundation capture APIs. This object configures the capture pipeline and manages its life cycle, and it defines an asynchronous interface to capture videos. It also delegates handling of those operatons to the app's `MovieCapture` object.
- `DockControlService` is an actor that manages interactions with a `DockAccessory` using DockKit APIs. This object listens to `DockAccessory` connection/disconnection events, manages subscriptions to the connected `DockAccessory`, and controls its movements using an asynchronous interface. It also delegates camera control in response to `DockAccessory` events to the `CameraModel` object.

> **Note**
>
> Configuring and starting a capture session are blocking operations that can take time to complete. To keep the user interface responsive, the app defines `CaptureService` as an actor type to ensure that AVFoundation capture API calls don't occur on the main thread.

Configure the sample code project

Because Simulator doesn't have access to device cameras and can't connect to a DockKit device, it isn't suitable for running the sample app. To run the app, you need an iPhone with iOS 18 or later.

Write a basic camera app to take photos

See AVCam: Building a camera app to learn how to write a basic camera app to capture videos using an iPhone's front and rear cameras.

Configure the DockKit accessory manager

AVCaptureSession is a singleton class that provides connection and disconnection notifications with a DockKit accessory by subscribing to the accessoryStateChanges API.

The dock control service subscribes to `accessoryStateChanges` in its `setUp(features: DockAccessoryFeatures)` method.

```
// Subscribe to accessory state changes.
for await stateEvent in try DockAccessoryManager.shared.accessoryStateChanges {
    // Save the DockKit accessory when docked (connected).
    if let newAccessory = stateEvent.accessory, stateEvent.state == .docked {
        dockkitAccessory = newAccessory
        await setupAccessorySubscriptions(for: newAccessory)
    }
}
```

When an accessory connects, DockKit sets it up to use system tracking, and to listen to accessory events and battery states in `setupAccessorySubscriptions(for accesory: DockAccessory)`.

```
func setupAccessorySubscriptions(for accesory: DockAccessory) async {
    // Enable system tracking on the first connection.
    try await DockAccessoryManager.shared.setSystemTrackingEnabled(true)
    // Start the necessary subscriptions to accessory events and battery states.
    subscribeToAccessoryEvents(for: accesory)
    toggleBatterySummary(to: true, for: accesory)
}
```

Change the tracking mode

The app provides a tracking mode menu to switch between system tracking, custom tracking, and manual tracking. The default is system tracking, which the app sets by calling `setSystemTrackingEnabled(_:)` to `true`.

```
func updateTrackingMode(to trackingMode: TrackingMode) async {
    self.trackingMode = trackingMode
    // Call `systemTrackingEnabled` with `true` to enable the system tracking mode.
    try await DockAccessoryManager.shared.setSystemTrackingEnabled(trackingMode == 
}
```

The app provides various menus and options to configure the selected subjects, selected frame, region to track, and more. All menus and buttons primarily live in the main DockKit menu.

Tap to track the subject

The app provides a tap-to-track toggle to enable or disable selecting a specific subject to track by tapping the camera view. When the tap-to-track toggle is enabled, the `selectSubject(at point: CGPoint?)` method allows people to select tapped subjects.

```
func selectSubject(at point: CGPoint?) async -> Bool {
    if let point = point {
        // Select a specific subject at the point.
        try await accessory.selectSubject(at: point)
    } else {
        // Clear the selected subjects.
        try await accessory.selectSubjects([])
    }
}
```

Set the region of interest

The app provides a region-of-interest toggle to enable or disable setting a region of interest to frame the selected subjects by holding and dragging the camera view. When toggling the region of interest, the `setRegionOfInterest(to region: CGRect)` method allows setting a region `CGRect` in the camera view. The dock accessory keeps the subjects framed in the selected region.

```
func setRegionOfInterest(to region: CGRect) async {
    try await accessory.setRegionOfInterest(region)
}
```

Set the framing mode

The app provides a framing mode menu to select a `FramingMode`.

```
func updateFraming(to framing: FramingMode) async -> Bool {
    try await accessory.setFramingMode(dockKitFramingMode(from: framing))
}
```

The app uses the helper function `dockKitFramingMode(from: framing)` to map a local `FramingMode` enumeration to `DockAccessory.FramingMode`.

```
func dockKitFramingMode(from framingMode: FramingMode) -> DockAccessory.FramingMode
    switch framingMode {
    case .auto:
        return DockAccessory.FramingMode.automatic
    case .center:
        return DockAccessory.FramingMode.center
    case .left:
        return DockAccessory.FramingMode.left
    case .right:
        return DockAccessory.FramingMode.right
    }
}
```

Implement manual control using actuator velocities

When someone sets the `TrackingMode` to `TrackingMode.manual`, the app provides chevrons to move `DockAccessory` up, left, right, and down by using the `setAngularVelocity(_:)` API.

```
func handleChevronTapped(chevronType: ChevronType, speed: Double = 0.2) async {
    var velocity = Vector3D()
    switch chevronType {
    case .tiltUp:
        velocity.x = -speed
        break
    case .tiltDown:
        velocity.x = speed
        break
    case .panLeft:
        velocity.y = -speed
        break
    case .panRight:
        velocity.y = speed
        break
    }
    try await dockkitAccessory.setAngularVelocity(velocity)
```

Run the default animations

The app provides buttons to run the four default animations that `DockAccessory` provides. Before running the animation, the app disables system tracking. When the animation is complete, the app restores system tracking to its prior value.

```
// Disable the system tracking before running the animation.
try await DockAccessoryManager.shared.setSystemTrackingEnabled(false)

// Run the animation and wait for it to finish.
let progress = try await dockkitAccessory.animate(motion: dockKitAnimation(from: ani
while(!progress.isCancelled && !progress.isFinished) {
    try await Task.sleep(nanoseconds: NSEC_PER_SEC/10) // 0.1 sec
}

// Restore the system tracking after running the animation.
try await DockAccessoryManager.shared.setSystemTrackingEnabled(trackingMode == .syst
```

The app uses the helper function `dockKitAnimation(from animation: Animation)` to map a local animation enumeration to `DockAccessory.Animation`.

```
func dockKitAnimation(from animation: Animation) -> DockAccessory.Animation {
    switch animation {
    case .yes:
        return DockAccessory.Animation.yes
    case .nope:
        return DockAccessory.Animation.nope
    case .wakeup:
        return DockAccessory.Animation.wakeup
    case .kapow:
        return DockAccessory.Animation.kapow
    }
}
```

The DockKit menu provides toggles to subscribe to various states, like battery and tracking, and displays them in the app's UI.

Implement the battery state

The dock control service subscribes to batteryStates to acquire the current battery state of the accessory. The current battery state includes the battery level, charging indicator, and so forth.

```
for await batterySummaryState in try dockkitAccessory.batteryStates {
    battery = .available(percentage: batterySummaryState.batteryLevel, charging: bat
}
```

Implement the tracking states

The dock control service subscribes to trackingStates to get a list of tracked subjects with attributes like saliency and speaking confidence. The dock control service delegates the handling of the conversion from a normalized subject rectangle to camera view space coordinates to the `CameraModel`, which uses the capture service for the operation. The app uses these states, along with the transformed subject rectangle, to show an overlay on the faces of the subjects.

```
for await trackingSummaryState in try dockkitAccessory.trackingStates {
    for subject in trackingSummaryState.trackedSubjects {
        switch subject {
        case .person(let person):
            if let rect = await cameraCaptureDelegate?.convertToViewSpace(from: pers
                // Create a `DockAccessoryTrackedPerson` object from `TrackingState`
                trackedPersons.append(DockAccessoryTrackedPerson(saliency: person.sa
                                                                 speaking: person.sp
            }
        default:
            // Do nothing.
            break
        }
    }
}
```

Implement camera control using accessory events

The dock control service subscribes to an `async` stream of AccessoryEvents. A physical input on the DockAccessory triggers an accessory event. When the app receives an accessory event, it delegates handling of the event to the `CameraModel`, which uses the capture service to perform camera operations.

```swift
for await event in try accesory.accessoryEvents {
    switch (event) {
    case let .button(id, pressed):
        break
    case .cameraZoom(factor: let factor):
        let zoomType = factor > 0 ? CameraZoomType.increase : CameraZoomType.decreas
        // Implement the camera zoom.
        cameraCaptureDelegate?.zoom(type: zoomType, factor: 0.2)
        break
    case .cameraShutter:
        if (Date.now.timeIntervalSince(lastShutterEventTime) > 0.2) {
            // Implement the camera start capture or stop capture.
            cameraCaptureDelegate?.startOrStartCapture()
            lastShutterEventTime = .now
        }
        break
    case .cameraFlip:
        // Implement the camera flip.
        cameraCaptureDelegate?.switchCamera()
        break
    default: break
    }
}
```

`CameraModel` implements the sample's `CameraCaptureDelegate` protocol and provides the helper methods to control the camera.

Implement the camera zoom

The capture service implements the `updateMagnification(for zoomType: CameraZoomType, by scale: Double = 0.2)` method in response to a zoom event from the accessory.

```swift
func updateMagnification(for zoomType: CameraZoomType, by scale: Double = 0.2) {
    try? currentDevice.lockForConfiguration()
    let magnification = (zoomType == .increase ? 1.0 : -1.0) * scale
    var newZoomFactor = currentDevice.videoZoomFactor + magnification
    newZoomFactor = max(min(newZoomFactor, self.maxZoomFactor), self.minZoomFactor)
    newZoomFactor = Double(round(10 * newZoomFactor) / 10)
    currentDevice.videoZoomFactor = newZoomFactor
    currentDevice.unlockForConfiguration()
    self.zoomFactor = newZoomFactor
}
```

Implement the camera shutter

The capture service implements the `startRecording()` method in response to a start-capture shutter event, and a `stopRecording()` method in response to a stop-capture shutter event.

```
func startRecording() {
    movieCapture.startRecording()
}

func stopRecording() async throws -> Movie {
    try await movieCapture.stopRecording()
}
```

Implement the camera flip

The capture service implements the `selectNextVideoDevice()` method in reponse to the camera flip event.

```
func selectNextVideoDevice() {
    // Change the session's active capture device.
    changeCaptureDevice(to: nextDevice)
}
```

See Also

Controlling the dock accessory

`class DockAccessoryManager`

Observe the state of dock accessories and enable or disable system tracking.

`class DockAccessory`

Obtain accessory information and control tracking behavior.

`enum DockKitError`

A list of errors that DockKit sends.