

# AVCam: Building a Camera App

---

Capture photos with depth data and record video using the front and rear iPhone and iPad cameras.

Sample code taken from:

[https://developer.apple.com/documentation/avfoundation/cameras\\_and\\_media\\_capture/avcam\\_building\\_a\\_camera\\_app](https://developer.apple.com/documentation/avfoundation/cameras_and_media_capture/avcam_building_a_camera_app)

## Overview

The iOS Camera app allows you to capture photos and movies from both the front and rear cameras. Depending on your device, the Camera app also supports the still capture of depth data, portrait effects matte, and Live Photos.

This sample code project, AVCam, shows you how to implement these capture features in your own camera app. It leverages basic functionality of the built-in front and rear iPhone and iPad cameras.

- Note: To use AVCam, you need an iOS device running iOS 13 or later. Because Xcode doesn't have access to the device camera, this sample won't work in Simulator. AVCam hides buttons for modes that the current device doesn't support, such as portrait effects matte delivery on an iPhone 7 Plus.

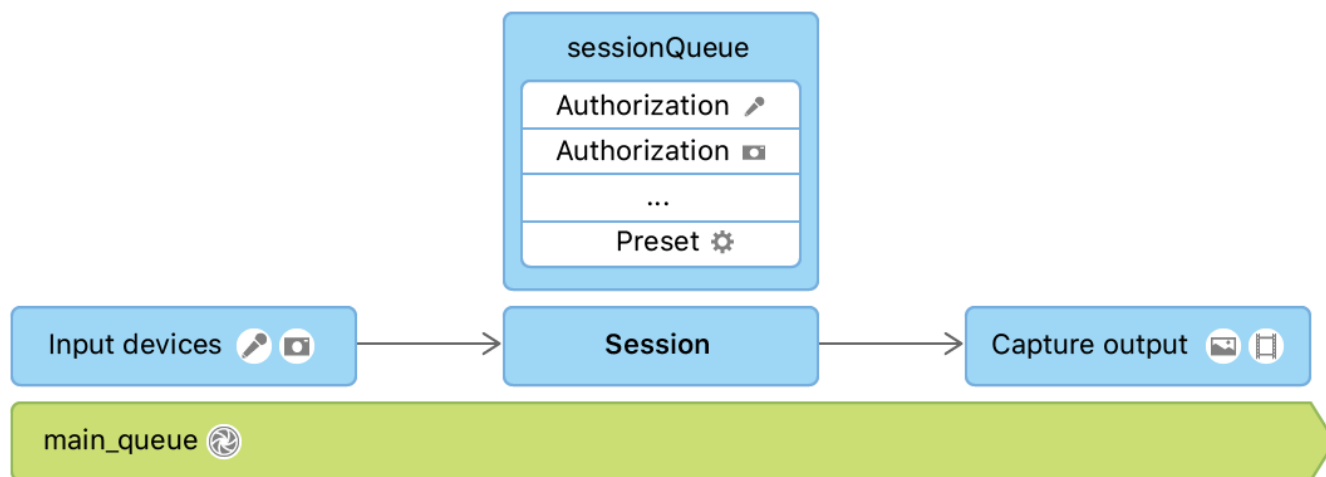
## Configure a Capture Session

**AVCaptureSession** accepts input data from capture devices like the camera and microphone. After receiving the input, **AVCaptureSession** marshals that data to appropriate outputs for processing, eventually resulting in a movie file or still photo. After configuring the capture session's inputs and outputs, you tell it to start—and later stop—capture.

```
private let session = AVCaptureSession()
```

AVCam selects the rear camera by default and configures a camera capture session to stream content to a video preview view. **PreviewView** is a custom **UIView** subclass backed by an **AVCaptureVideoPreviewLayer**. AVFoundation doesn't have a **PreviewView** class, but the sample code creates one to facilitate session management.

The following diagram shows how the session manages input devices and capture output:



Delegate any interaction with the `AVCaptureSession`—including its inputs and outputs—to a dedicated serial dispatch queue (`sessionQueue`), so that the interaction doesn't block the main queue. Perform any configuration involving changes to a session's topology or disruptions to its running video stream on a separate dispatch queue, since session configuration always blocks execution of other tasks until the queue processes the change. Similarly, the sample code dispatches other tasks—such as resuming an interrupted session, toggling capture modes, switching cameras, and writing media to a file—to the session queue, so that their processing doesn't block or delay user interaction with the app.

In contrast, the code dispatches tasks that affect the UI (such as updating the preview view) to the main queue, because `AVCaptureVideoPreviewLayer`, a subclass of `CALayer`, is the backing layer for the sample's preview view. You must manipulate `UIView` subclasses on the main thread for them to show up in a timely, interactive fashion.

In `viewDidLoad`, AVCam creates a session and assigns it to the preview view:

```
previewView.session = session
```

For more information about configuring image capture sessions, see [Setting Up a Capture Session](#).

## Request Authorization for Access to Input Devices

Once you configure the session, it is ready to accept input. Each `AVCaptureDevice`—whether a camera or a mic—requires the user to authorize access. AVFoundation enumerates the authorization state using `AVAuthorizationStatus`, which informs the app whether the user has restricted or denied access to a capture device.

For more information about preparing your app's `Info.plist` for custom authorization requests, see [Requesting Authorization for Media Capture](#).

## Switch Between the Rear- and Front-Facing Cameras

The `changeCamera` method handles switching between cameras when the user taps a button in the UI. It uses a discovery session, which lists available device types in order of preference, and accepts the first device in its `devices` array. For example, the `videoDeviceDiscoverySession` in AVCam queries the device on which

the app is running for available input devices. Furthermore, if a user's device has a broken camera, it won't be available in the `devices` array.

```
switch currentPosition {
case .unspecified, .front:
    preferredPosition = .back
    preferredDeviceType = .builtInDualCamera

case .back:
    preferredPosition = .front
    preferredDeviceType = .builtInTrueDepthCamera

@unknown default:
    print("Unknown capture position. Defaulting to back, dual-camera.")
    preferredPosition = .back
    preferredDeviceType = .builtInDualCamera
}
```

If the discovery session finds a camera in the proper position, it removes the previous input from the capture session and adds the new camera as an input.

```
// Remove the existing device input first, because AVCaptureSession
// doesn't support
// simultaneous use of the rear and front cameras.
self.session.removeInput(self.videoDeviceInput)

if self.session.canAddInput(videoDeviceInput) {
    NotificationCenter.default.removeObserver(self, name:
    .AVCaptureDeviceSubjectAreaDidChange, object: currentVideoDevice)
    NotificationCenter.default.addObserver(self, selector:
    #selector(self.subjectAreaDidChange), name:
    .AVCaptureDeviceSubjectAreaDidChange, object: videoDeviceInput.device)

    self.session.addInput(videoDeviceInput)
    self.videoDeviceInput = videoDeviceInput
} else {
    self.session.addInput(self.videoDeviceInput)
}
```

[View in Source](#)

## Handle Interruptions and Errors

Interruptions such as phone calls, notifications from other apps, and music playback may occur during a capture session. Handle these interruptions by adding observers to listen for

`AVCaptureSessionWasInterruptedNotification`:

```
NotificationCenter.default.addObserver(self,
                                      selector:
#selector(sessionWasInterrupted),
                                      name:
    .AVCaptureSessionWasInterrupted,
                                      object: session)
NotificationCenter.default.addObserver(self,
                                      selector:
#selector(sessionInterruptionEnded),
                                      name:
    .AVCaptureSessionInterruptionEnded,
                                      object: session)
```

[View in Source](#)

When AVCam receives an interruption notification, it can pause or suspend the session with an option to resume activity when the interruption ends. AVCam registers `sessionWasInterrupted` as a handler for receiving notifications, to inform the user when there's an interruption to the capture session:

```
if reason == .audioDeviceInUseByAnotherClient || reason ==
.videoDeviceInUseByAnotherClient {
    showResumeButton = true
} else if reason == .videoDeviceNotAvailableWithMultipleForegroundApps {
    // Fade-in a label to inform the user that the camera is unavailable.
    cameraUnavailableLabel.alpha = 0
    cameraUnavailableLabel.isHidden = false
    UIView.animate(withDuration: 0.25) {
        self.cameraUnavailableLabel.alpha = 1
    }
} else if reason == .videoDeviceNotAvailableDueToSystemPressure {
    print("Session stopped running due to shutdown system pressure
level.")
}
```

[View in Source](#)

The camera view controller observes `AVCaptureSessionRuntimeError` to receive a notification when an error occurs:

```
NotificationCenter.default.addObserver(self,
                                      selector:
#selector(sessionRuntimeError),
                                      name:
    .AVCaptureSessionRuntimeError,
                                      object: session)
```

When a runtime error occurs, restart the capture session:

```
// If media services were reset, and the last start succeeded, restart the session.
if error.code == .mediaServicesWereReset {
    sessionQueue.async {
        if self.isSessionRunning {
            self.session.startRunning()
            self.isSessionRunning = self.session.isRunning
        } else {
            DispatchQueue.main.async {
                self.resumeButton.isHidden = false
            }
        }
    }
} else {
    resumeButton.isHidden = false
}
```

[View in Source](#)

The capture session may also stop if the device sustains system pressure, such as overheating. The camera won't degrade capture quality or drop frames on its own; if it reaches a critical point, the camera stops working, or the device shuts off. To avoid surprising your users, you may want your app to manually lower the frame rate, turn off depth, or modulate performance based on feedback from `AVCaptureSystemPressureState`:

```
let pressureLevel = systemPressureState.level
if pressureLevel == .serious || pressureLevel == .critical {
    if self.movieFileOutput == nil || self.movieFileOutput?.isRecording == false {
        do {
            try self.videoDeviceInput.device.lockForConfiguration()
            print("WARNING: Reached elevated system pressure level: \(pressureLevel). Throttling frame rate.")
            self.videoDeviceInput.device.activeVideoMinFrameDuration = CMTime(value: 1, timescale: 20)
            self.videoDeviceInput.device.activeVideoMaxFrameDuration = CMTime(value: 1, timescale: 15)
            self.videoDeviceInput.device.unlockForConfiguration()
        } catch {
            print("Could not lock device for configuration: \(error)")
        }
    }
} else if pressureLevel == .shutdown {
    print("Session stopped running due to shutdown system pressure level.")
}
```

[View in Source](#)

## Capture a Photo

Taking a photo happens on the session queue. The process begins by updating the `AVCapturePhotoOutput` connection to match the video orientation of the video preview layer. This enables the camera to accurately capture what the user sees onscreen:

```
if let photoOutputConnection = self.photoOutput.connection(with: .video) {
    photoOutputConnection.videoOrientation = videoPreviewLayerOrientation!
}
```

After aligning the outputs, AVCam proceeds to create `AVCapturePhotoSettings` to configure capture parameters such as focus, flash, and resolution:

```
var photoSettings = AVCapturePhotoSettings()

// Capture HEIF photos when supported. Enable auto-flash and high-
// resolution photos.
if self.photoOutput.availablePhotoCodecTypes.contains(.hevc) {
    photoSettings = AVCapturePhotoSettings(format: [AVVideoCodecKey:
AVVideoCodecType.hevc])
}

if self.videoDeviceInput.device.isFlashAvailable {
    photoSettings.flashMode = .auto
}

photoSettings.isHighResolutionPhotoEnabled = true
if !photoSettings.__availablePreviewPhotoPixelFormatTypes.isEmpty {
    photoSettings.previewPhotoFormat = [kCVPixelBufferPixelFormatTypeKey
as String: photoSettings.__availablePreviewPhotoPixelFormatTypes.first!]
}
// Live Photo capture is not supported in movie mode.
if self.livePhotoMode == .on &&
self.photoOutput.isLivePhotoCaptureSupported {
    let livePhotoMovieFileName = NSUUID().uuidString
    let livePhotoMovieFilePath = (NSTemporaryDirectory() as
NSString).appendingPathComponent((livePhotoMovieFileName as
NSString).appendingPathExtension("mov")!)
    photoSettings.livePhotoMovieFileURL = URL(fileURLWithPath:
livePhotoMovieFilePath)
}

photoSettings.isDepthDataDeliveryEnabled = (self.depthDataDeliveryMode ==
.on
    && self.photoOutput.isDepthDataDeliveryEnabled)

photoSettings.isPortraitEffectsMatteDeliveryEnabled =
(self.portraitEffectsMatteDeliveryMode == .on
    && self.photoOutput.isPortraitEffectsMatteDeliveryEnabled)

if photoSettings.isDepthDataDeliveryEnabled {
    if !self.photoOutput.enabledSemanticSegmentationMatteTypes.isEmpty {
```

```
        photoSettings.enabledSemanticSegmentationMatteTypes =
    self.photoOutput.enabledSemanticSegmentationMatteTypes
    }
}

photoSettings.photoQualityPrioritization =
    self.photoQualityPrioritizationMode
```

The sample uses a separate object, the **PhotoCaptureProcessor**, for the photo capture delegate to isolate each capture life cycle. This clear separation of capture cycles is necessary for Live Photos, where a single capture cycle may involve the capture of several frames.

Each time the user presses the central shutter button, AVCam captures a photo with the previously configured settings by calling **capturePhotoWithSettings**:

```
self.photoOutput.capturePhoto(with: photoSettings, delegate:
    photoCaptureProcessor)
```

[View in Source](#)

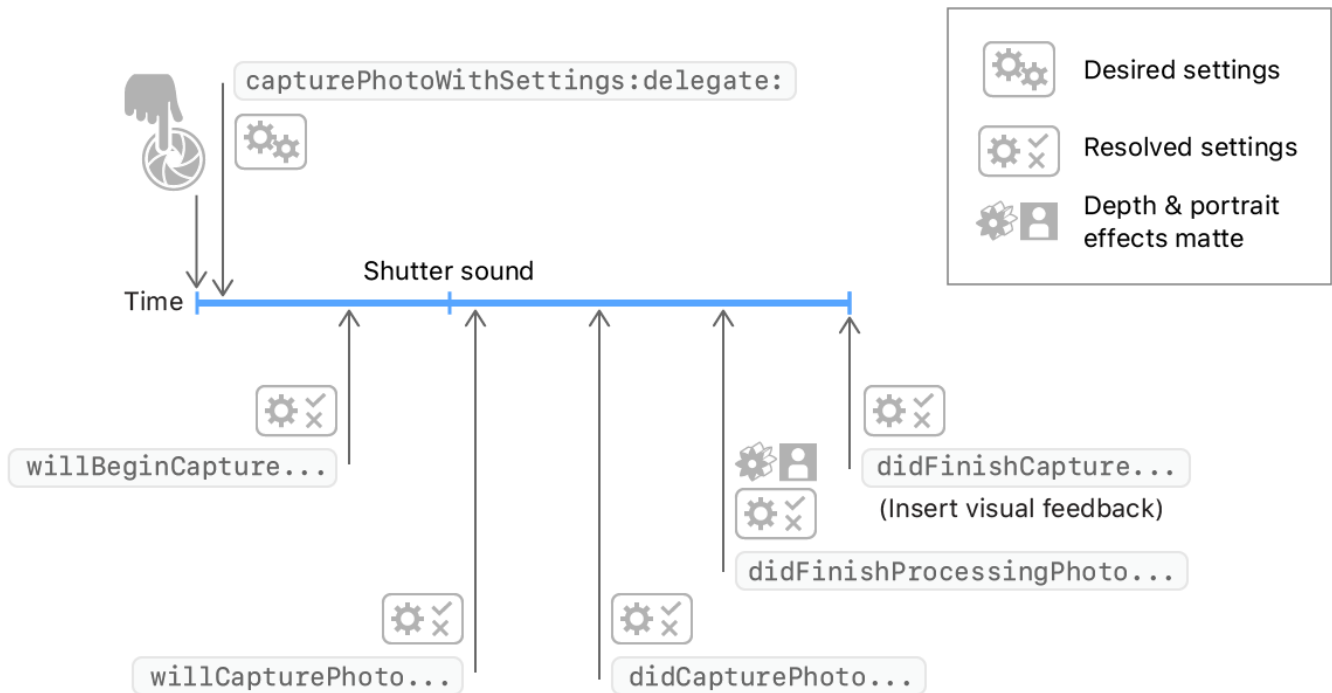
The **capturePhoto** method accepts two parameters:

- An **AVCapturePhotoSettings** object that encapsulates the settings your user configures through the app, such as exposure, flash, focus, and torch.
- A delegate that conforms to the **AVCapturePhotoCaptureDelegate** protocol, to respond to subsequent callbacks that the system delivers during photo capture.

Once the app calls **capturePhoto**, the process for starting photography is over. From that point forward, operations on that individual photo capture happens in delegate callbacks.

## Track Results Through a Photo Capture Delegate

The method **capturePhoto** only begins the process of taking a photo. The rest of the process happens in delegate methods that the app implements.



- `photoOutput(_:willBeginCaptureFor:)` arrives first, as soon as you call `capturePhoto`. The resolved settings represent the actual settings that the camera will apply for the upcoming photo. AVCam uses this method only for behavior specific to Live Photos. AVCam tries to tell if the photo is a Live Photo by checking its `livePhotoMovieDimensions` size; if the photo is a Live Photo, AVCam increments a count to track Live Photos in progress:

```
self.sessionQueue.async {
    if capturing {
        self.inProgressLivePhotoCapturesCount += 1
    } else {
        self.inProgressLivePhotoCapturesCount -= 1
    }

    let inProgressLivePhotoCapturesCount =
self.inProgressLivePhotoCapturesCount
    DispatchQueue.main.async {
        if inProgressLivePhotoCapturesCount > 0 {
            self.capturingLivePhotoLabel.isHidden = false
        } else if inProgressLivePhotoCapturesCount == 0 {
            self.capturingLivePhotoLabel.isHidden = true
        } else {
            print("Error: In progress Live Photo capture count is less
than 0.")
        }
    }
}
```

[View in Source](#)

- `photoOutput(_:willCapturePhotoFor:)` arrives right after the system plays the shutter sound. AVCam uses this opportunity to flash the screen, alerting to the user that the camera captured a photo.



The sample code implements this flash by animating the preview view layer's `opacity` from `0` to `1`.

```
// Flash the screen to signal that AVCam took a photo.
DispatchQueue.main.async {
    self.previewView.videoPreviewLayer.opacity = 0
    UIView.animate(withDuration: 0.25) {
        self.previewView.videoPreviewLayer.opacity = 1
    }
}
```

[View in Source](#)

- `photoOutput(_:didFinishProcessingPhoto:error:)` arrives when the system finishes processing depth data and a portrait effects matte. AVCam checks for a portrait effects matte and depth metadata at this stage:

```
// A portrait effects matte gets generated only if AVFoundation detects a face.
if var portraitEffectsMatte = photo.portraitEffectsMatte {
    if let orientation = photo.metadata[
String(kCGImagePropertyOrientation)] as? UInt32 {
        portraitEffectsMatte =
portraitEffectsMatte.applyingExifOrientation(CGImagePropertyOrientation(rawValue: orientation!))
    }
    let portraitEffectsMattePixelBuffer =
portraitEffectsMatte.mattingImage
    let portraitEffectsMatteImage = CIImage( cvImageBuffer:
portraitEffectsMattePixelBuffer, options: [
.auxiliaryPortraitEffectsMatte: true ] )
}
```

[View in Source](#)

- `photoOutput(_:didFinishCaptureFor:error:)` is the final callback, marking the end of capture for a single photo. AVCam cleans up its delegate and settings so they don't remain for subsequent photo captures:

```
self.sessionQueue.async {

    self.inProgressPhotoCaptureDelegates[photoCaptureProcessor.requestedPhotoSettings.uniqueID] = nil
}
```

[View in Source](#)

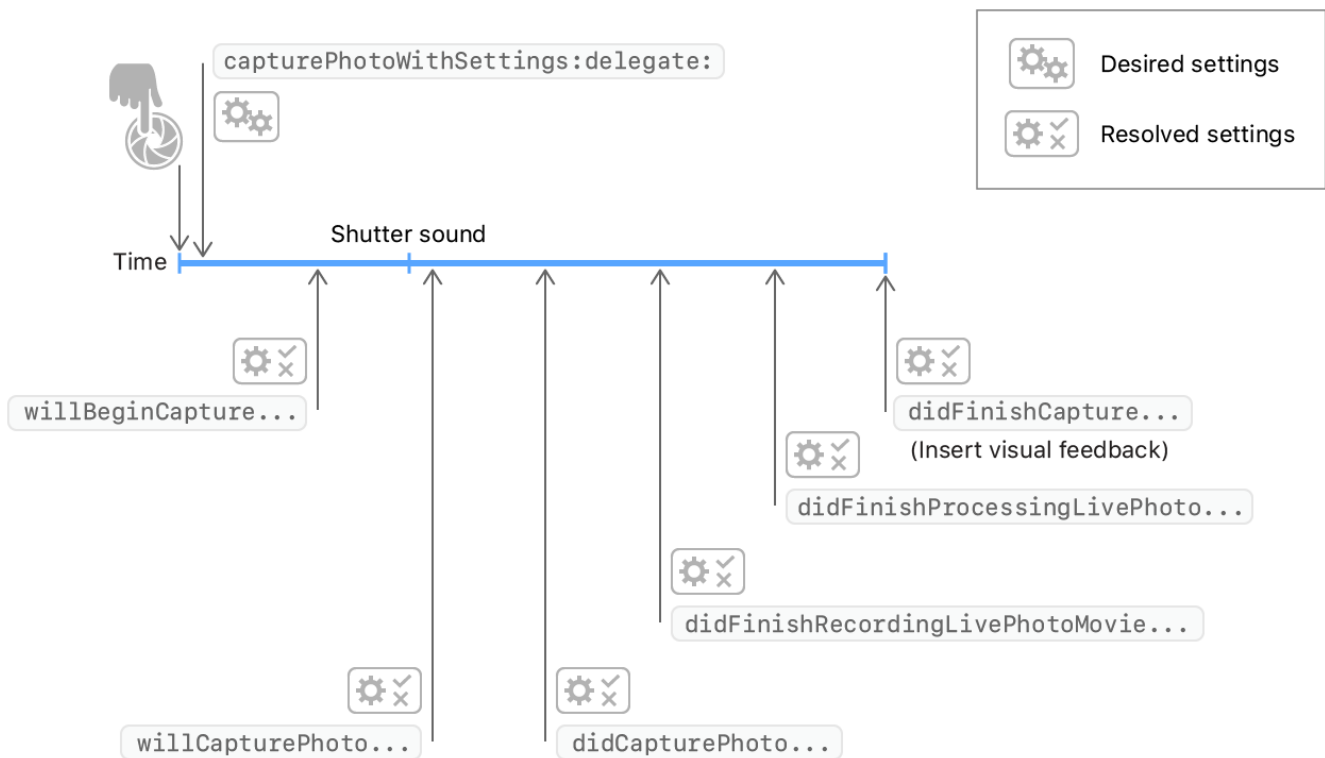
You can apply other visual effects in this delegate method, such as animating a preview thumbnail of the captured photo.

For more information about tracking photo progress through delegate callbacks, see [Tracking Photo Capture Progress](#).

## Capture Live Photos

When you enable capture of Live Photos, the camera takes one still image and a short movie around the moment of capture. The app triggers Live Photo capture the same way as still photo capture: through a single call to `capturePhotoWithSettings`, where you pass the URL for the Live Photos short video through the `livePhotoMovieFileURL` property. You can enable Live Photos at the `AVCapturePhotoOutput` level, or you can configure Live Photos at the `AVCapturePhotoSettings` level on a per-capture basis.

Since Live Photo capture creates a short movie file, AVCam must express where to save the movie file as a URL. Also, because Live Photo captures can overlap, the code must keep track of the number of in-progress Live Photo captures to ensure that the Live Photo label stays visible during these captures. The `photoOutput(_:willBeginCaptureFor:)` delegate method in the previous section implements this tracking counter.



- `photoOutput(_:didFinishRecordingLivePhotoMovieForEventualFileAt:resolvedSettings:)` fires when recording of the short movie ends. AVCam dismisses the Live badge here. Because the camera has finished recording the short movie, AVCam executes the Live Photo handler decrementing the completion counter:

```
livePhotoCaptureHandler(false)
```

[View in Source](#)

- `photoOutput(_:didFinishProcessingLivePhotoToMovieFileAt:duration:photoDisplayTime:resolvedSettings:error:)` fires last, indicating that the movie is fully written to disk and is ready for consumption. AVCam uses this opportunity to display any capture errors and redirect the saved file URL to its final output location:

```
if error != nil {
    print("Error processing Live Photo companion movie: \
(String(describing: error))")
    return
}
livePhotoCompanionMovieURL = outputFileURL
```

[View in Source](#)

For more information about incorporating Live Photo capture into your app, see [Capturing Still and Live Photos](#).

## Capture Depth Data and Portrait Effects Matte

Using `AVCapturePhotoOutput`, AVCam queries the capture device to see whether its configuration can deliver depth data and a portrait effects matte to still images. If the input device supports either of these modes, and you enable them in the capture settings, the camera attaches depth and portrait effects matte as auxiliary metadata on a per-photo request basis. If the device supports delivery of depth data, portrait effects matte, or Live Photos, the app shows a button, used to toggle the settings for enabling or disabling the feature.

```
if self.photoOutput.isDepthDataDeliverySupported {
    self.photoOutput.isDepthDataDeliveryEnabled = true

    DispatchQueue.main.async {
        self.depthDataDeliveryButton.isEnabled = true
    }
}

if self.photoOutput.isPortraitEffectsMatteDeliverySupported {
    self.photoOutput.isPortraitEffectsMatteDeliveryEnabled = true

    DispatchQueue.main.async {
        self.portraitEffectsMatteDeliveryButton.isEnabled = true
    }
}

if !self.photoOutput.availableSemanticSegmentationMatteTypes.isEmpty {
    self.photoOutput.enabledSemanticSegmentationMatteTypes =
self.photoOutput.availableSemanticSegmentationMatteTypes

    DispatchQueue.main.async {
        self.semanticSegmentationMatteDeliveryButton.isEnabled =
(self.depthDataDeliveryMode == .on) ? true : false
    }
}
```

```
DispatchQueue.main.async {
    self.livePhotoModeButton.isHidden = false
    self.depthDataDeliveryButton.isHidden = false
    self.portraitEffectsMatteDeliveryButton.isHidden = false
    self.semanticSegmentationMatteDeliveryButton.isHidden = false
    self.photoQualityPrioritizationSegControl.isHidden = false
    self.photoQualityPrioritizationSegControl.isEnabled = true
}
```

[View in Source](#)

The camera stores depth and portrait effects matte metadata as auxiliary images, discoverable and addressable through the **Image IO** API. AVCam accesses this metadata by searching for an auxiliary image of type **auxiliaryPortraitEffectsMatte**:

```
if var portraitEffectsMatte = photo.portraitEffectsMatte {
    if let orientation =
photo.metadata[String(kCGImagePropertyOrientation)] as? UInt32 {
        portraitEffectsMatte =
portraitEffectsMatte.applyingExifOrientation(CGImagePropertyOrientation(rawValue: orientation!))
    }
    let portraitEffectsMattePixelBuffer =
portraitEffectsMatte.mattingImage
```

For more information about depth data capture, see [Capturing Photos with Depth](#).

## Capture Semantic Segmentation Mattes

Using **AVCapturePhotoOutput**, AVCam also lets you capture semantic segmentation mattes, which segment a person's hair, skin, and teeth into distinct matte images. The ability to capture these auxiliary images along with your primary photo simplifies applying photo effects, such as changing a person's hair color or brightening their smile.

You enable the capture of these auxiliary images by setting the photo output's **enabledSemanticSegmentationMatteTypes** property to your preferred values (**hair**, **skin**, and **teeth**). To capture all supported types, set this property to match the photo output's **availableSemanticSegmentationMatteTypes** property.

```
// Capture all available semantic segmentation matte types.
photoOutput.enabledSemanticSegmentationMatteTypes =
    photoOutput.availableSemanticSegmentationMatteTypes
```

When the photo output finishes capturing a photo, you retrieve the associated segmentation matte images by querying the photo's **semanticSegmentationMatte(forType:)** method. This method returns an **AVSemanticSegmentationMatte** that contains the matte image and additional metadata that you can use when

processing the image. The sample app adds the semantic segmentation matte image's data to an array so you can write it to the user's photo library.

```
// Find the semantic segmentation matte image for the specified type.
guard var segmentationMatte = photo.semanticSegmentationMatte(for:
ssmType) else { return }

// Retrieve the photo orientation and apply it to the matte image.
if let orientation = photo.metadata[String(kCGImagePropertyOrientation)]
as? UInt32,
    let exifOrientation = CGImagePropertyOrientation(rawValue:
orientation) {
    // Apply the Exif orientation to the matte image.
    segmentationMatte =
segmentationMatte.applyingExifOrientation(exifOrientation)
}

var imageOption: CIImageOption!

// Switch on the AVSemanticSegmentationMatteType value.
switch ssmType {
case .hair:
    imageOption = .auxiliarySemanticSegmentationHairMatte
case .skin:
    imageOption = .auxiliarySemanticSegmentationSkinMatte
case .teeth:
    imageOption = .auxiliarySemanticSegmentationTeethMatte
default:
    print("This semantic segmentation type is not supported!")
    return
}

// Create a new CIImage from the matte's underlying CVPixelBuffer.
let ciImage = CIImage(cvImageBuffer: segmentationMatte.mattingImage,
options: [imageOption: true])

// Get the HEIF representation of this image.
guard let linearColorSpace = CGColorSpace(name: CGColorSpace.linearSRGB),
    let imageData = context.heifRepresentation(of: ciImage,
                                                format: .RGBA8,
                                                colorSpace:
linearColorSpace,
                                                options: [.depthImage:
ciImage]) else { return }

// Add the image data to the SSM data array for writing to the photo
library.
semanticSegmentationMattedataArray.append(imageData)
```

## Save Photos to the User's Photo Library

Before you can save an image or movie to the user's photo library, you must first request access to that library. The process for requesting write authorization mirrors capture device authorization: show an alert with text that you provide in the `Info.plist`.

AVCam checks for authorization in the `'captureOutput:didFinishRecordingToOutputFileAtURL:fromConnections'` callback method, which is where the `AVCaptureOutput` provides media data to save as output.

```
PHPhotoLibrary.requestAuthorization { status in
```

For more information about requesting access to the user's photo library, see [Requesting Authorization to Access Photos](#).

## Record Movie Files

AVCam supports video capture by querying and adding input devices with the `.video` qualifier. The app defaults to the rear dual camera, but, if the device doesn't have a dual camera, the app defaults to the wide-angle camera.

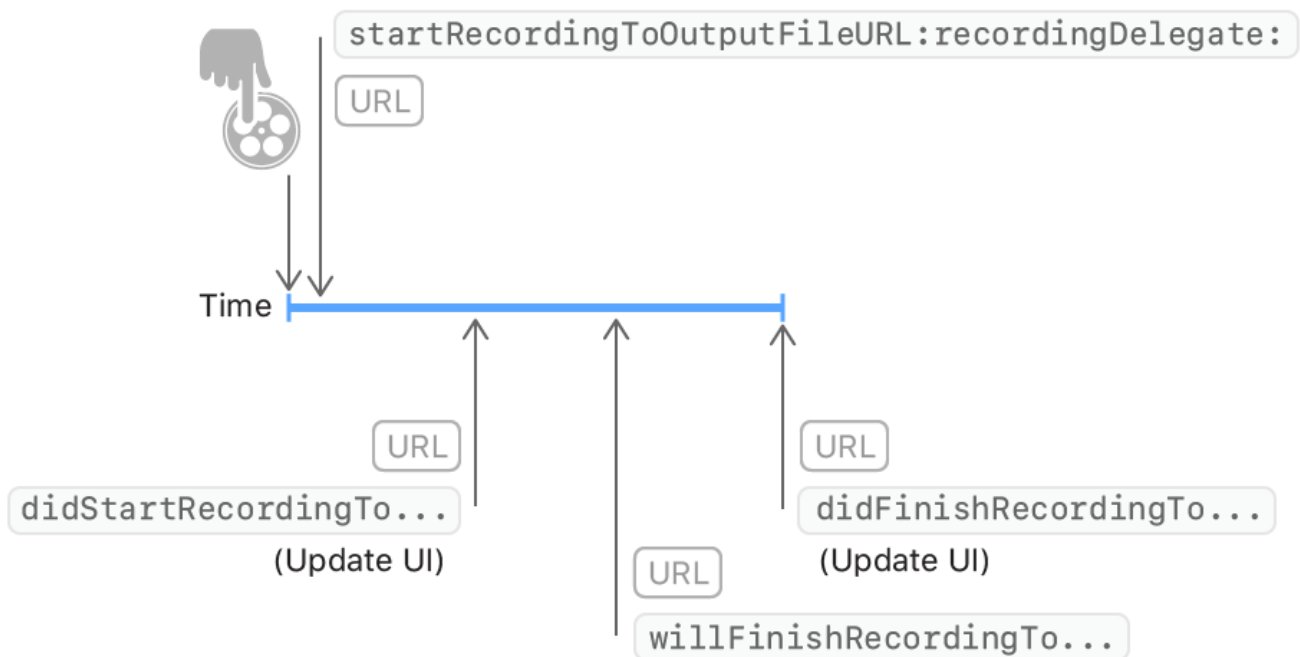
```
if let dualCameraDevice = AVCaptureDevice.default(.builtInDualCamera, for:
.video, position: .back) {
    defaultVideoDevice = dualCameraDevice
} else if let backCameraDevice =
AVCaptureDevice.default(.builtInWideAngleCamera, for: .video, position:
.back) {
    // If a rear dual camera is not available, default to the rear wide
angle camera.
    defaultVideoDevice = backCameraDevice
} else if let frontCameraDevice =
AVCaptureDevice.default(.builtInWideAngleCamera, for: .video, position:
.front) {
    // If the rear wide angle camera isn't available, default to the front
wide angle camera.
    defaultVideoDevice = frontCameraDevice
}
```

Instead of passing settings to the system as with still photography, pass an output URL like in Live Photos. The delegate callbacks provide the same URL, so your app doesn't need to store it in an intermediate variable.

Once the user taps Record to begin capture, AVCam calls `startRecording`:

```
movieFileOutput.startRecording(to: URL(fileURLWithPath: outputFilePath),
recordingDelegate: self)
```

Just like `capturePhoto` triggered delegate callbacks for still capture, `startRecording` triggers a series of delegate callbacks for movie recording.



Track the progress of the movie recording through the delegate callback chain. Instead of implementing `AVCapturePhotoCaptureDelegate`, implement `AVCaptureFileOutputRecordingDelegate`. Since the movie-recording delegate callbacks require interaction with the capture session, AVCam makes `CameraViewController` the delegate instead of creating a separate delegate object.

- `fileOutput(_:didStartRecordingTo:from:)` fires when the file output starts writing data to a file. AVCam uses this opportunity to change the Record button to a Stop button:

```
DispatchQueue.main.async {
    self.recordButton.isEnabled = true
    self.recordButton.setImage(#imageLiteral(resourceName: "CaptureStop"),
    for: [])
}
```

[View in Source](#)

- `fileOutput(_:didFinishRecordingTo:from:error:)` fires last, indicating that the movie is fully written to disk and is ready for consumption. AVCam takes this chance to move the temporarily saved movie from the given URL to the user's photo library or the app's documents folder:

```
PHPhotoLibrary.shared().performChanges({
    let options = PHAssetResourceCreationOptions()
    options.shouldMoveFile = true
    let creationRequest = PHAssetCreationRequest.forAsset()
    creationRequest.addResource(with: .video, fileURL: outputFileURL,
    options: options)
}, completionHandler: { success, error in
    if !success {
        print("AVCam couldn't save the movie to your photo library: \(
    (String(describing: error))")
    }
})
```

```
    }  
    cleanup()  
  }  
}
```

[View in Source](#)

In the event that AVCam goes into the background—such as when the user accepts an incoming phone call—the app must ask permission from the user to continue recording. AVCam requests time from the system to perform this saving through a background task. This background task ensures that there is enough time to write the file to the photo library, even when AVCam recedes to the background. To conclude background execution, AVCam calls `endBackgroundTask` in `didFinishRecordingTo` after saving the recorded file.

```
self.backgroundRecordingID =  
UIApplication.shared.beginBackgroundTask(expirationHandler: nil)
```

## Take Photos While Recording a Movie

Like the iOS Camera app, AVCam can take photos while also capturing a movie. AVCam captures such photos at the same resolution as the video.

```
let movieFileOutput = AVCaptureMovieFileOutput()  
  
if self.session.canAddOutput(movieFileOutput) {  
    self.session.beginConfiguration()  
    self.session.addOutput(movieFileOutput)  
    self.session.sessionPreset = .high  
    if let connection = movieFileOutput.connection(with: .video) {  
        if connection.isVideoStabilizationSupported {  
            connection.preferredVideoStabilizationMode = .auto  
        }  
    }  
    self.session.commitConfiguration()  
  
    DispatchQueue.main.async {  
        captureModeControl.isEnabled = true  
    }  
  
    self.movieFileOutput = movieFileOutput  
  
    DispatchQueue.main.async {  
        self.recordButton.isEnabled = true  
  
        /*  
         For photo captures during movie recording, Speed quality photo  
         processing is prioritized  
         to avoid frame drops during recording.  
        */  
    }
```



```
        self.photoQualityPrioritizationSegControl.selectedSegmentIndex = 0
        self.photoQualityPrioritizationSegControl.sendActions(for:
    UIControl.Event.valueChanged)
    }
}
```