Sample Code

Recognizing and Labeling Arbitrary Objects

Create anchors that track objects you recognize in the camera feed, using a custom optical-recognition algorithm.

Download

iOS 12.0+ | iPadOS 12.0+ | Xcode 16.0+

Overview

This sample app parses the camera feed, using the Vision framework with a Core ML model that recognizes regular desktop items. The app displays a label onscreen that indicates when it recognizes an item. You then tap the screen to place a textual annotation in the physical environment that's labeled with the name of the recognized object. Because the Core ML model used by this app doesn't tell you where the object lies within an image, label placement relative to the object depends on where you tap.

Note

ARKit requires an iOS device with an A9 or later processor. ARKit is not available in iOS Simulator.

Implement the vision/core ML image classifier

The sample code's `classificationRequest` property, `classifyCurrentImage()` `method, and` processClassifications(for:error:)` method manage:

- A Core ML image-classifier model, loaded from an `mlmodel` file bundled with the app using the Swift API that Core ML generates for the model
- `VNCoreMLRequest` and `VNImageRequestHandler` objects for passing image data to the model for evaluation

For more details on using `VNImageRequestHandler`,`VNCoreMLRequest`, and image classifier models, see the Classifying Images with Vision and Core ML sample-code project.

Run the AR session and process camera images

The sample `ViewController` class manages the AR session and displays AR overlay content in a SpriteKit view. ARKit captures video frames from the camera and provides them to the view controller in the `session(_:didUpdate:)` method, which then calls the `classifyCurrentImage()` method to run the Vision image classifier.

```
func session(_ session: ARSession, didUpdate frame: ARFrame) {
    // Do not enqueue other buffers for processing while another Vision task is stil
    // The camera stream has only a finite amount of buffers available; holding too
    guard currentBuffer == nil, case .normal = frame.camera.trackingState else {
        return
    }

    // Retain the image buffer for Vision processing.
    self.currentBuffer = frame.capturedImage
    classifyCurrentImage()
}
```

Serialize image processing for real-time performance

The `classifyCurrentImage()` method uses the view controller's `currentBuffer` property to track whether Vision is currently processing an image before starting another Vision task.

```
// Most computer vision tasks are not rotation agnostic so it is important to pass i
let orientation = CGImagePropertyOrientation(UIDevice.current.orientation)

let requestHandler = VNImageRequestHandler(cvPixelBuffer: currentBuffer!, orientatio
visionQueue.async {
    do {
        // Release the pixel buffer when done, allowing the next buffer to be proces
        defer { self.currentBuffer = nil }
        try requestHandler.perform([self.classificationRequest])
    } catch {
        print("Error: Vision request failed with error \"\(error)\"")
    }
}
```

Important

Limit your processing to one buffer at a time for performance. The camera recycles a finite pool of pixel buffers, so retaining too many buffers for processing could starve the camera and shut down the capture session. Passing multiple buffers to Vision for processing would slow down processing of each image, adding latency and reducing the amount of CPU and GPU overhead for rendering AR visualizations.

In addition, the sample app enables the `usesCPUOnly` setting for its Vision request, freeing the GPU for use in rendering.

Visualize results in AR

The `processClassifications(for:error:) method stores the best-match result label produced by the image classifier and displays it in the corner of the screen. The user can then tap in the AR scene to place that label at a real-world position. Placing a label requires two main steps.

First, a tap gesture recognizer fires the `placeLabelAtLocation(sender:)` action. This method uses the ARKit `hitTest(_:types:)` method to estimate the 3D real-world position corresponding to the tap, and adds an anchor to the AR session at that position.

```
@IBAction func placeLabelAtLocation(sender: UITapGestureRecognizer) {
    let hitLocationInView = sender.location(in: sceneView)
    let hitTestResults = sceneView.hitTest(hitLocationInView, types: [.featurePoint,
    if let result = hitTestResults.first {

        // Add a new anchor at the tap location.
        let anchor = ARAnchor(transform: result.worldTransform)
        sceneView.session.add(anchor: anchor)

        // Track anchor ID to associate text with the anchor after ARKit creates a c
        anchorLabels[anchor.identifier] = identifierString
    }
}
```

Next, after ARKit automatically creates a SpriteKit node for the newly added anchor, the `view(_:didAdd:for:)` delegate method provides content for that node. In this case, the sample `TemplateLabelNode` class creates a styled text label using the string provided by the image classifier.

```
func view(_ view: ARSKView, didAdd node: SKNode, for anchor: ARAnchor) {
    guard let labelText = anchorLabels[anchor.identifier] else {
        fatalError("missing expected associated label for anchor")
    }
    let label = TemplateLabelNode(text: labelText)
    node.addChild(label)
}
```

See Also

Text Annotations

{} Creating screen annotations for objects in an AR experience

Annotate an AR experience with virtual sticky notes that you display onscreen over real and virtual objects.