Sample Code

Tracking and altering images

Create images from rectangular shapes found in the user's environment, and augment their appearance.

Download

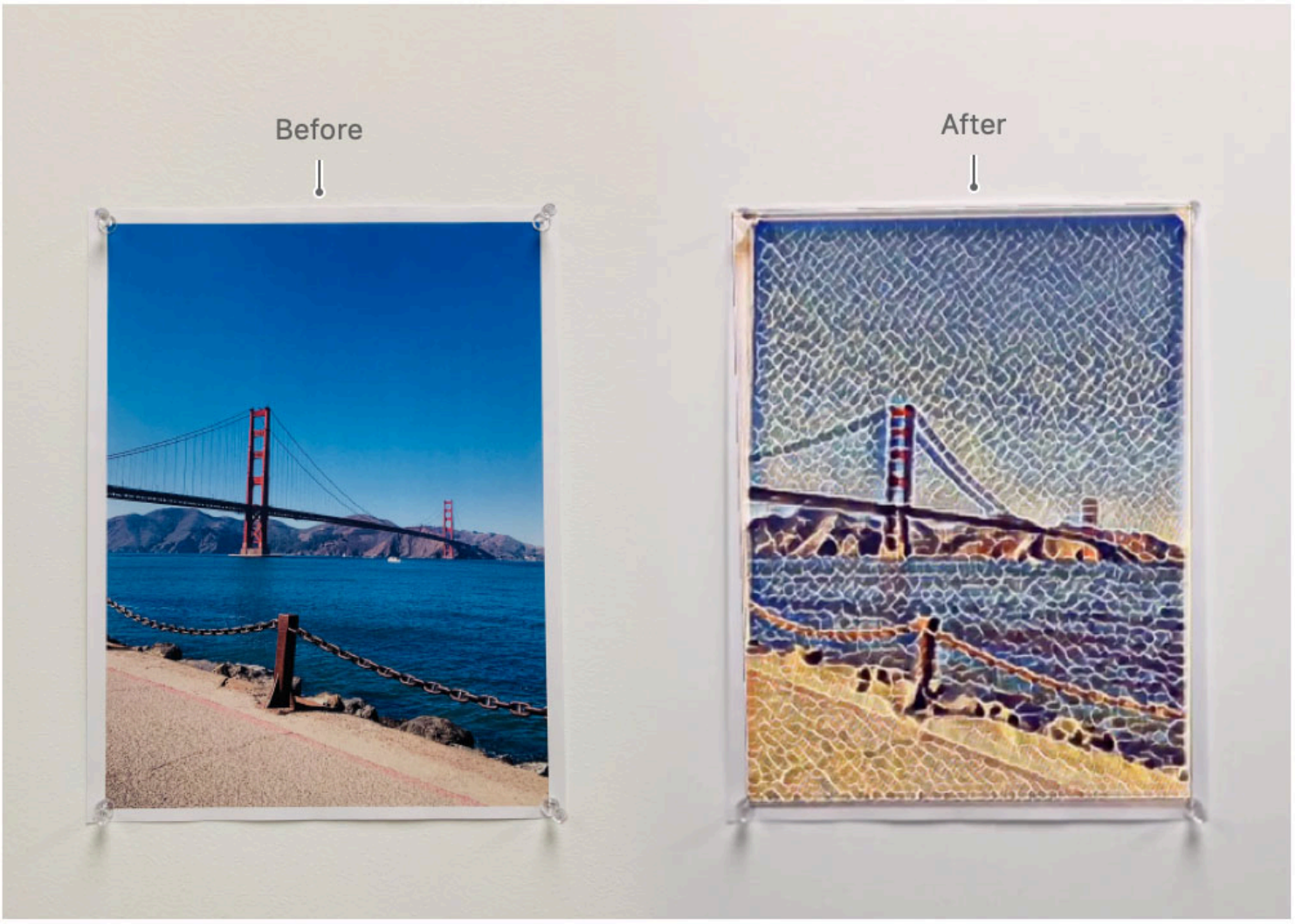
iOS 13.0+ | iPadOS 13.0+ | Xcode 16.0+

Overview

To demonstrate general image recognition, this sample app uses Vision to detect rectangular shapes in the user's environment that are most likely artwork or photos. Run the app on an iPhone or iPad, and point the device's camera at a movie poster or wall-mounted picture frame. When the app detects a rectangular shape, you extract the pixel data defined by that shape from the camera feed to create an image.

The sample app changes the appearance of the image by applying a Core ML model that performs a stylistic alteration. By repeating this action in succession, you achieve real-time image processing using a trained neural network.

To complete the effect of augmenting an image in the user's environment, you use ARKit's image tracking feature. ARKit can hold an altered image steady over the original image as the user moves the device in their environment. ARKit also tracks the image if it moves on its own, as when the app recognizes a banner on the side of a bus, and the bus begins to drive away.



This sample app uses SceneKit to render its graphics.

Detect Rectangular Shapes in the User's Environment

As shown below, you can use Vision in real-time to check the camera feed for rectangles. You perform this check up to 10 times a second by using `RectangleDetector` to schedule a repeating timer with an `updateInterval` of `0.1` seconds.

```
init() {
    self.updateTimer = Timer.scheduledTimer(withTimeInterval: updateInterval, repeat
        if let capturedImage = ViewController.instance?.sceneView.session.currentFra
            self?.search(in: capturedImage)
        }
    }
}
```

Because Vision requests can be taxing on the processor, check the camera feed no more than 10 times a second. Checking for rectangles more frequently may cause the app's frame rate to decrease, without noticeably improving the app's results.

When you make Vision requests in real-time with an ARKit–based app, you should do so serially. By waiting for one request to finish before invoking another, you ensure that the AR experience remains smooth and free of interruptions. In the `search` function, you use the `isBusy` flag to ensure you're only checking for one rectangle at a time:

```
private func search(in pixelBuffer: CVPixelBuffer) {
    guard !isBusy else { return }
    isBusy = true

    // ...
}
```

The sample sets the `isBusy` flag to `false` when a Vision request completes or fails.

Crop the Camera Feed to an Observed Rectangle

When Vision finds a rectangle in the camera feed, it provides you with the rectangle's precise coordinates through a `VNRectangleObservation`. You apply those coordinates to a Core Image perspective correction filter to crop it, leaving you with just the image data inside the rectangular shape.

```
private func completedVisionRequest(_ request: VNRequest?, error: Error?) {
    defer {
        isBusy = false
    }
    // Only proceed if a rectangular image was detected.
    guard let rectangle = request?.results?.first as? VNRectangleObservation else {
        guard let error = error else { return }
        print("Error: Rectangle detection failed - Vision request returned an error.
        return
    }
    guard let filter = CIFilter(name: "CIPerspectiveCorrection") else {
        print("Error: Rectangle detection failed - Could not create perspective corr
        return
    }
    let width = CGFloat(CVPixelBufferGetWidth(currentCameraImage))
    let height = CGFloat(CVPixelBufferGetHeight(currentCameraImage))
```

```
    let topLeft = CGPoint(x: rectangle.topLeft.x * width, y: rectangle.topLeft.y * h
    let topRight = CGPoint(x: rectangle.topRight.x * width, y: rectangle.topRight.y
    let bottomLeft = CGPoint(x: rectangle.bottomLeft.x * width, y: rectangle.bottoml
    let bottomRight = CGPoint(x: rectangle.bottomRight.x * width, y: rectangle.botto

    filter.setValue(CIVector(cgPoint: topLeft), forKey: "inputTopLeft")
    filter.setValue(CIVector(cgPoint: topRight), forKey: "inputTopRight")
    filter.setValue(CIVector(cgPoint: bottomLeft), forKey: "inputBottomLeft")
    filter.setValue(CIVector(cgPoint: bottomRight), forKey: "inputBottomRight")

    let ciImage = CIImage(cvPixelBuffer: currentCameraImage).oriented(.up)
    filter.setValue(ciImage, forKey: kCIInputImageKey)

    guard let perspectiveImage: CIImage = filter.value(forKey: kCIOutputImageKey) as
        print("Error: Rectangle detection failed - perspective correction filter has
        return
    }
    delegate?.rectangleFound(rectangleContent: perspectiveImage)
}
```

Using the first image in the Overview, the camera image is:

The cropped result is:

Create a Reference Image

To prepare to track the cropped image, you create an `ARReferenceImage`, which provides ARKit with everything it needs, like its look and physical size, to locate that image in the physical environment.

```
let possibleReferenceImage = ARReferenceImage(referenceImagePixelBuffer,
                                              orientation: .up,
                                              physicalWidth: CGFloat(0.5))
```

ARKit requires that reference images contain sufficient detail to be recognizable; for example, a plain white image cannot be tracked. To prevent ARKit from failing to track a reference image, you validate it first before attempting to use it.

```
possibleReferenceImage.validate { [weak self] (error) in
    if let error = error {
        print("Reference image validation failed: \(error.localizedDescription)")
        return
    }
    // ...
```

Track the Image Using ARKit

Provide the reference image to ARKit to get updates on where the image lies in the camera feed when the user moves their device. Do that by creating an image tracking session and passing the reference image in to the configuration's `trackingImages`.

```
let configuration = ARImageTrackingConfiguration()
configuration.maximumNumberOfTrackedImages = 1
configuration.trackingImages = trackingImages
sceneView.session.run(configuration, options: runOptions)
```

Vision made the initial observation about where the image lies in 2D space in the camera feed, but ARKit resolves its location in 3D space, in the physical environment. When ARKit succeeds in recognizing the image, it creates an `ARImageAnchor` and a SceneKit node at the right position. You save the anchor and node that ARKit gives you by passing them to an `AlteredImage` object.

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAnch
    alteredImage?.add(anchor, node: node)
    setMessageHidden(true)
}
```

Alter the Image's Appearance Using Core ML

This sample app is bundled with a Core ML model that performs image processing. Given an input image and an integer index, the model outputs a visually modified version of that image in one of eight different styles. The particular style of the output depends on the value of the index you pass in. The first style resembles burned paper, the second style resembles a mosaic, and there are six other styles as shown in the following image.

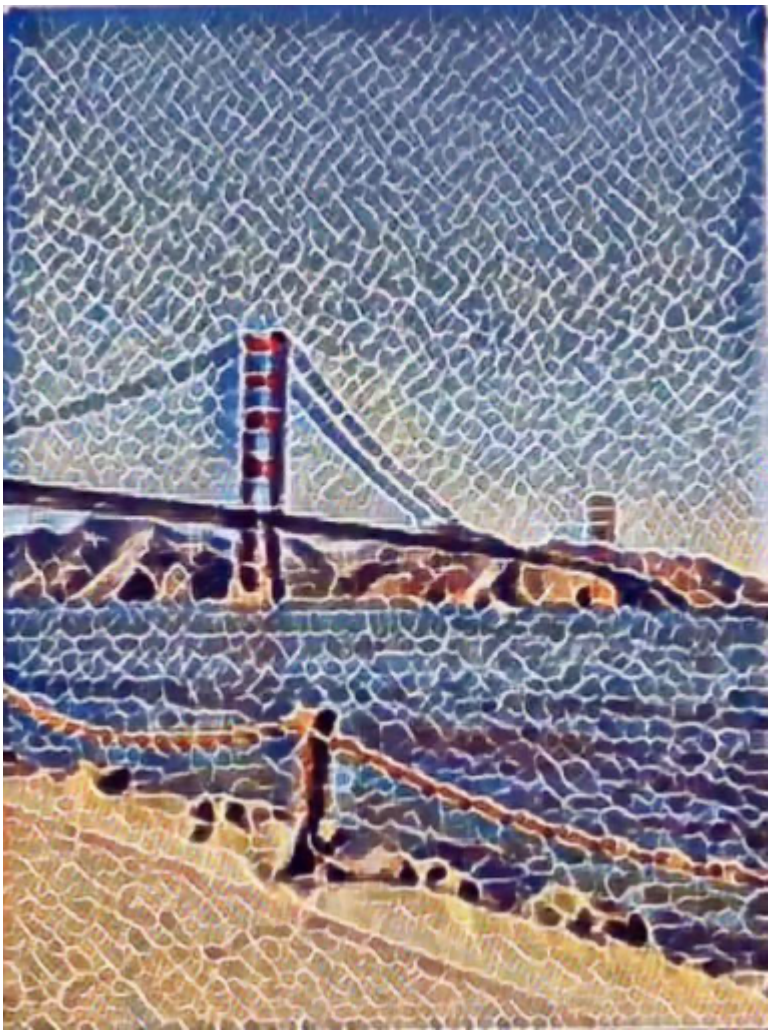
When Vision finds a rectangular shape in the user's environment, you pass the camera's image data defined by that rectangle into a new `AlteredImage`.

```
guard let newAlteredImage = AlteredImage(rectangleContent, referenceImage: possibleF
```

The following code shows how you choose the artistic style to apply to the image by inputting the integer index to the Core ML model. Then, you process the image by calling the Core ML model's `predictions(from:options:)`.

```
let input = StyleTransferModelInput(image: self.modelInputImage, index: self.styleIr
let output = try AlteredImage.styleTransferModel.prediction(input: input, options: c
```

The following figure shows the result when you process the input image with a style index of 2.



Display the Altered Image in Augmented Reality

To complete the augmented reality effect, you cover the original image with the altered image. First, add a visualization node to hold the altered image as a child of the node provided by ARKit.

```
node.addChildNode(visualizationNode)
```

When Core ML produces the output image, you call `imageAlteringComplete(_:)` to pass the model's output image into the visualization node's `display` function, where you set the image as the visualization node's contents.

```
func imageAlteringComplete(_ createdImage: CVPixelBuffer) {
    guard fadeBetweenStyles else { return }
    modelOutputImage = createdImage
    visualizationNode.display(createdImage)
}
```

The visualization node's contents overlap the original image when SceneKit displays it. In the case of the image above, the following screenshot shows the end result as seen through a user's

device:

Continually Update the Image’s Appearance

This sample demonstrates real-time image processing by switching artistic styles over time. By calling `selectNextStyle`, you can make successive alterations of the original image. `styleIndex` is the integer input to the Core ML model that determines the style of the output.

```
func selectNextStyle() {
    styleIndex = (styleIndex + 1) % numberOfStyles
}
```

The sample's `VisualizationNode` fades between two images of differing style, which creates the effect that the tracked image is constantly transforming into a new look. You accomplish this effect by defining two SceneKit nodes. One node displays the current altered image, and the other displays the previous altered image.

```
private let currentImage: SCNNode
private let previousImage: SCNNode
```

You fade between these two nodes by running an opacity animation:

```
SCNTransaction.begin()
SCNTransaction.animationDuration = fadeDuration
currentImage.opacity = 1.0
previousImage.opacity = 0.0
SCNTransaction.completionBlock = {
    self.delegate?.visualizationNodeDidFinishFade(self)
}
SCNTransaction.commit()
```

When the animation finishes, you begin altering the original image with the next artistic style by calling `createAlteredImage` again:

```
func visualizationNodeDidFinishFade(_ visualizationNode: VisualizationNode) {
    guard fadeBetweenStyles, anchor != nil else { return }
    selectNextStyle()
    createAlteredImage()
}
```

Respond to Image Tracking Updates

As part of the image tracking feature, ARKit continues to look for the image throughout the AR session. If the image itself moves, ARKit updates the `ARImageAnchor` with its corresponding image's new location in the physical environment, and calls your delegate's `renderer(_:didUpdate:for:)` to notify your app of the change.

```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode, for anchor: ARA
    alteredImage?.update(anchor)
}
```

The sample app tracks a single image at a time. To do that, you invalidate the current image tracking session if an image the app was tracking is no longer visible. This, in turn, enables Vision to start looking for a new rectangular shape in the camera feed.

```
func update(_ anchor: ARAnchor) {
    if let imageAnchor = anchor as? ARImageAnchor, self.anchor == anchor {
        self.anchor = imageAnchor
        // Reset the timeout if the app is still tracking an image.
        if imageAnchor.isTracked {
            resetImageTrackingTimeout()
        }
    }
}
```

See Also

Image tracking

{} Detecting Images in an AR Experience

React to known 2D images in the user's environment, and use their positions to place AR content.

Tracking preregistered images in 3D space

Place content based on the current position of a known image in a person's surroundings.

`class ImageTrackingProvider`

A source of live data about a 2D image's position in a person's surroundings.

`struct ImageAnchor`

A 2D image's position in a person's surroundings.

`struct` `ReferenceImage`

A 2D image the system uses as a reference to find the same image in a person's surroundings.