
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Copyright ©2018 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

W4: ARKit Workshop

- This workshop will make use of ARKit 1.5.
- You'll need latest version of Xcode 9.3 (preferably beta 3 or later) on your Mac
- You'll need iOS 11.3 (preferably beta 5 or later) on your ARKit-ready iDevice
- You'll need an ARKit-ready iDevice, which is one that has an A9 processor or later. iPhone: 6S / 6S Plus / 7S / 7S Plus / 8 / 8 Plus / X. iPad: 2017 model. iPad Pro: 9.7" / 10.5" / 12.9".
- You should bring a lightning cable (preferably a longer one) to the workshop. Hotel wifi is notoriously bad, and you should expect wifi syncing to be painfully slow.

W4: ARKit Workshop

In this workshop, you'll learn how to use ARKit to make the world your view controller! You'll first make a paint program that takes Bob Ross into the third dimension, and then you'll recreate that popular catalog app from everyone's favorite semi-disposable Swedish furniture company.

You'll follow those up with a museum app that has a touch of "Black Mirror", and finally, bring us one step closer to SkyNet by combining augmented reality and machine learning. Along the way, we'll sneak in a lot of augmented reality programming principles, techniques, and tricks, but you'll have so much fun that you won't even realize that you're learning new things!

73 ARKit Workshop: Demo 1

By Joey deVilla

Welcome to the "Hello, world" demo! Don't worry — this one will be fun. You'll pay tribute to the legendary painting instructor Bob Ross by building an AR painting app, and along the way you'll...

- Set up an ARKit Scene View
- Draw basic shapes into an augmented reality scene
- Animate shapes in an augmented reality scene
- Get the device's location, orientation, and position and turn your ARKit-ready iPhone or iPad into a "brush" that paints various shapes, both static and animated, into the place you're in.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo1/starter**. To keep things simple, we've set things up so that you'll do all your work in just one place: **CanvasViewController.swift**.

1) Set up the AR SceneKit view

At the top of **CanvasViewController.swift**, add this code to the bottom of the list of properties, below the *"Define AR configuration"* comment:

```
let configuration = ARWorldTrackingConfiguration()
```

In `viewDidLoad()`, implement *"Set up the AR SceneKit view":*

```
canvas.delegate = self
```

```
canvas.debugOptions = [ARSCNDebugOptions.showWorldOrigin,
                       ARSCNDebugOptions.showFeaturePoints]
canvas.showsStatistics = true
canvas.autoenablesDefaultLighting = true
canvas.session.run(configuration)
```

Run the app. You should see a number of big changes:

- Most of the screen now shows what the rear camera sees.
- There's now an ARKit statistics bar between the **Paint** button and the tabs.
- You should see three intersecting lines — red, green, and blue — located around the point where your device was when the app launched. This marks what ARKit considers to be the coordinates of the origin, and the red, green, and blue lines mark the X-, Y-, and Z-axes respectively.

2) Add a happy lil' orange sphere to the AR scene

In `drawTestShapes()`, implement *"Draw happy lil' orange sphere":*

```
// Draw happy lil' orange sphere
let sphere = SCNNode(geometry: SCNSphere(radius: 0.05))
sphere.position = SCNVector3(0, 0, -0.3)
sphere.geometry?.firstMaterial?.diffuse.contents = UIColor.orange
canvas.scene.rootNode.addChildNode(sphere)
```

In `viewDidLoad()`, add this to the end of the method:

```
drawTestShapes()
```

Run the app. You should now see an orange ball floating about a foot away from you.

3) Add a shiny happy lil' blue box, tilted at a jaunty angle, to the AR scene

In `drawTestShapes()`, implement *"Draw happy lil' blue box, tilted at a jaunty angle":*

```
// Draw happy lil' blue box, tilted at a jaunty angle
let box = SCNNode(geometry: SCNBox(width: 0.1, height: 0.1, length: 0.1,
chamferRadius: 0))
box.position = SCNVector3(0, 0.3, -0.2)
let degrees45 = Double.pi / 8
box.eulerAngles = SCNVector3(degrees45, degrees45, degrees45)
```

```
box.geometry?.firstMaterial?.diffuse.contents = UIColor.blue
box.geometry?.firstMaterial?.specular.contents = UIColor.white
canvas.scene.rootNode.addChildNode(box)
```

Run the app. You'll see the orange ball, and if you tilt your device upwards from where the orange ball is located, you'll see a shiny blue box tilted at a jaunty angle.

4) Animate the blue box

In `drawTestShapes()`, implement *"Animate the blue box":*

```
// Animate the blue box
let rotateAction = SCNAction.rotate(by: 2 * .pi,
                                    around: SCNVector3(0, 1, 0),
                                    duration: 2)
let rotateForeverAction = SCNAction.repeatForever(rotateAction)
box.runAction(rotateForeverAction)
```

Run the app and look at the blue box. You'll see that it's now rotating about its Y-axis.

At this point, you've made the ARKit version of "Hello, world!". It's now time to make something more like a real app.

5) Get the device's location, orientation, and position

First, remove the call to `drawTestShapes()` from `viewDidLoad()`. You don't need it anymore.

Then, in `render(_:willRenderScene:atTime)` (in the **ARSCNViewDelegate methods** section), implement *"Get the device's location, orientation, and position":*

```
// Get the device's location, orientation, and position
guard let pointOfView = canvas.pointOfView else { return }
let transform = pointOfView.transform
let orientation = SCNVector3(-transform.m31,
                             -transform.m32,
                             -transform.m33)
let location = SCNVector3(transform.m41,
                          transform.m42,
                          transform.m43)
let position = orientation + location
print("location: \(location)\norientation: \(orientation)")
```

Run the app and look at the debug console. Tilt your device in various positions and see how the numbers change.

6) Create a brush

In `render(_:willRenderScene:atTime)`, implement *"Create the brush and erase any old cursor shapes":*

```
// Create the brush and erase any old cursor shapes
let brush = self.createBrush(brushShape: self.brushSettings.shape,
                             brushSize: self.brushSettings.size,
                             position: position)
self.eraseNodes(named: "cursor")
```

Next, go inside the `if self.paintButton.isHighlighted` statement and handle the case where the user is pressing the **Paint** button by implementing *"Give the shape a shine and set it to the selected color":*

```
// Give the shape a shine and set it to the selected color
brush.geometry?.firstMaterial?.diffuse.contents =
self.brushSettings.color
brush.geometry?.firstMaterial?.specular.contents = UIColor.white
```

Then, in the `else` clause, handle the case where the user is *not* pressing the **Paint** button by implementing *"Set the shape to the cursor color and name":*

```
// Set the shape to the cursor color and name
brush.geometry?.firstMaterial?.diffuse.contents = UIColor.lightGray
brush.name = "cursor"
```

Finally, go past the end of the `if` statement and implement *"Paint the shape to the screen":*

```
// Paint the shape to the screen
self.canvas.scene.rootNode.addChildNode(brush)
```

Run the app. You can now paint in AR space! You can even go to the **Brush settings** tab and change the color, type, and size of the brush. The only control on that tab that *doesn't* work is the **Paint spinning shapes?** switch, and we're going to make it work in the next step.

7) Animate the shape if it's supposed to be animated

In `render(_:willRenderScene:atTime)`, implement *"Spin the shape continuously around the y-axis":*

```
if self.brushSettings.isSpinning {
  // Spin the shape continuously around the y-axis
  let rotateAction = SCNAction.rotate(by: 2 * .pi,
```

```
                                        around: SCNVector3(0, 1, 0),
                                        duration: 2)
    let rotateForeverAction = SCNAction.repeatForever(rotateAction)
    brush.runAction(rotateForeverAction)
  }
```

Run the app. Switch to the **Brush settings** tab, turn the **Paint spinning shapes?** on, switch back to the **Paint** tab, and start painting. The shapes you paint now rotate around their y-axes in a mesmerizing way. Bob Ross would be pleased!

8) That's it!

Congrats, at this time you should have both a fully functional AR painting app *and* a good understanding of ARKit basics, including:

- Setting up an ARKit Scene View
- Drawing basic shapes into an augmented reality scene
- Animating shapes in an augmented reality scene
- Getting the device's location, orientation, and position

It's time to move on to the next topic: plane detection and how it can help you choose your next piece of Swedish semi-disposable furniture.

74 ARKit Workshop: Demo 2

By Joey deVilla

In this demo, you'll make **Raykea**, the scaled-down RWDevCon answer to the *IKEA Place* app. If you're not familiar with the app, it's an AR app that helps you answer the question "What would items from the IKEA catalog look like if they were in this room?"

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo2/starter**. To keep things simple, we've set things up so that you'll do all your work in just one place: **RoomViewController.swift**.

1) Set up the AR configuration

Go to **Intializers** section and find the `createARConfiguration()` method. Enter the following between the `let config` and `return config` lines:

```
config.worldAlignment = .gravity
config.planeDetection = [.horizontal, .vertical]
config.isLightEstimationEnabled = true
config.providesAudioData = false
```

2) Implement the method that draws AR planes over any detected surfaces

When the app detects a horizontal surface in the real world, it draws a grid over that surface, and the user can tap the grid to place AR furniture on it. When the

app detects a vertical surface in the real world, it covers it with an AR poster of Ray (because a good Ray poster really pulls the room together).

Go to the **Plane detection** section and find the `drawPlaneNode(on:for:)` method. Implement *"Create a plane node with the same position and size as the detected plane":*

```
let planeNode = SCNNode(geometry: SCNPlane(
  width: CGFloat(planeAnchor.extent.x),
  height: CGFloat(planeAnchor.extent.z)
))
planeNode.position = SCNVector3(planeAnchor.center.x,
                                planeAnchor.center.y,
                                planeAnchor.center.z)
planeNode.geometry?.firstMaterial?.isDoubleSided = true
```

`SCNPlanes` are perpendicular to their anchor by default, so we need to rotate the plane by 90 degrees clockwise around the x-axis. Do this by implementing *"Align the plane with the anchor":*

```
// Align the plane with the anchor.
planeNode.eulerAngles = SCNVector3(-Double.pi / 2, 0, 0)
```

We have now created an AR plane node, and we've given that node a position and orientation. It's time to apply an image to the node's surface, which will depend on whether it's horizontal or vertical. Implement *"Give the plane node the appropriate surface"* with the following:

```
// Give the plane node the appropriate surface.
if planeAnchor.alignment == .horizontal {
  planeNode.geometry?.firstMaterial?.diffuse.contents = UIImage(named:
"grid")
  planeNode.name = "horizontal"
} else {
  planeNode.geometry?.firstMaterial?.diffuse.contents = UIImage(named:
"ray")
  planeNode.name = "vertical"
}
```

And finally, implement *"Add the plane node to the scene":*

```
// Add the plane node to the scene.
node.addChildNode(planeNode)
appState = .readyToFurnish
```

3) Handle newly-detected surfaces

Not far above the `drawPlaneNode(on:for:)` method that you just implemented is the `renderer(_:didAdd:for:)` method. It's called whenever a SceneKit node for a new AR anchor is added to the scene.

Go to `renderer(_:didAdd:for:)` and implement the section that begins with *"We only want to deal with plane anchors":*

```
// We only want to deal with plane anchors, which encapsulate
// the position, orientation, and size, of a detected surface.
guard let planeAnchor = anchor as? ARPlaneAnchor else { return }
```

Now that we've ensured that we're dealing with a plane anchor and nothing else, we can add a plane to the AR scene. Do this by implementing *"Draw the appropriate plane over the detected surface":*

```
// Draw the appropriate plane over the detected surface.
drawPlaneNode(on: node, for: planeAnchor)
```

Run the app and point your device about the room until the yellow feature dots appear. Then point it at the floor, a nearby wall, or even your computer's monitor. The following should happen:

- When it detects a horizontal surface, such as the floor or a table, it will draw a grid with the text "Place furniture here" over that surface.
- When it detects a vertical surface with a border, such as a picture frame or a computer monitor, it will draw a poster of Ray overt it.

The pictures drawn over the detected planes will be positioned with their centers at the location of their corresponding plane anchors, and their lengths and widths will match their plane anchors' extents.

4) Handle changes to the position or size of a previously detected horizontal surface

The `renderer(_:didUpdate:for:)` method is just below the method you were implementing. It's called whenever the properties for an AR anchor in the scene are adjusted. This happens when ARKIt revises its estimation of the position or size of a previously detected surface.

Start with a `guard` to ensure that the method responds only when the updated anchor is a plane anchor. Do this by implementing the *"Once again, we only want to deal with plane anchors"* method:

```
// Once again, we only want to deal with plane anchors.
guard let planeAnchor = anchor as? ARPlaneAnchor else { return }
```

Now that you've ensured that you're dealing only with a plane anchor, remove any child nodes its corresponding node may have. Implement *"Remove any children this node may have":*

```
// Remove any children this node may have.
node.enumerateChildNodes { (childNode, _) in
  childNode.removeFromParentNode()
}
```

Now implement *"Update the plane over this surface":*

```
// Update the plane over this surface.
drawPlaneNode(on: node, for: planeAnchor)
```

5) Handle surfaces that have been deleted

Let's implement the `renderer(_:didRemove:for:)` method. It's called whenever the SceneKit node for an AR anchor has been removed from the scene. This happens when ARKit determines that a previously detected surface isn't there anymore.

Start with a `guard` to ensure that the method responds only to the removal of a plane from the scene. Implement *"We only want to deal with plane anchors":*

```
// We only want to deal with plane anchors.
guard anchor is ARPlaneAnchor else { return }
```

Now that you've ensured that you're dealing with a plane anchor, remove any child nodes its corresponding node may have. Do this implementing *"Remove any children this node may have":*

```
// Remove any children this node may have.
node.enumerateChildNodes { (childNode, _) in
  childNode.removeFromParentNode()
}
```

Run the app again. This time, not only do grids and Ray appear over horizontal and vertical surfaces respectively, but they also adjust in position, size, and orientation as ARKit gets more information about the surfaces, and disappear as ARKit decides that they're no longer in the scene.

6) Add the ability to determine if a detected horizontal surface is currently on-screen

Complete the logic for the `isAnyPlaneInView()` method (in the **App status** section) by implementing *"Perform hit test for planes":*

```
// Perform hit test for planes.
let hitTest = sceneView.hitTest(point, types: .existingPlaneUsingExtent)
if !hitTest.isEmpty {
  return true
```

```
}
```

Run the app, find a surface, then point your device towards the ceiling or away from any detected surface. The status area near the top of the screen will display the message **Point your device towards one of the detected surfaces.**

7) Handle taps on the screen

Go to the `handleScreenTap(sender:)` method in the **Adding and removing furniture** section. Implement *"Find out where the user tapped on the screen":*

```
// Find out where the user tapped on the screen.
let tappedSceneView = sender.view as! ARSCNView
let tapLocation = sender.location(in: tappedSceneView)
```

Then implement the section whose name begins with *"Find all the detected planes that would intersect":*

```
// Find all the detected planes that would intersect with
// a line extending from where the user tapped the screen.
let planeIntersections = tappedSceneView.hitTest(tapLocation, types:
[.existingPlaneUsingGeometry])
```

And finally, implement the section whose name begins with *"If the closest of those planes is horizontal":*

```
// If the closest of those planes is horizontal,
// put the current furniture item on it.
if !planeIntersections.isEmpty {
  let firstHitTestResult = planeIntersections.first!
  guard let planeAnchor = firstHitTestResult.anchor as? ARPlaneAnchor
else { return }
  if planeAnchor.alignment == .horizontal {
    addFurniture(hitTestResult: firstHitTestResult)
  }
}
```

8) Draw the currently selected piece of furniture at the location where the user tapped

This is handled by the `addFurniture(hitTestResult:)` method, which comes immediately after `handleScreenTap(sender:)`.

First, determine where in the scene the furniture should be drawn. Implement *"Get*

the real-world position corresponding to where the user tapped on the screen":

```
// Get the real-world position corresponding to
// where the user tapped on the screen.
// Get the real-world position corresponding to
// where the user tapped on the screen.
let transform = hitTestResult.worldTransform
let positionColumn = transform.columns.3 // 4th column; column index
starts at 0
let initialPosition = SCNVector3(positionColumn.x,
                                 positionColumn.y,
                                 positionColumn.z)
```

Then add the furniture to the scene by implementing *"Get the current furniture item, correct its position if necessary, and add it to the scene":*

```
// Get the current furniture item, correct its position if necessary,
// and add it to the scene.
let node = furnitureSettings.currentFurniturePiece()
node.position = initialPosition +
furnitureSettings.currentFurnitureOffset()
sceneView.scene.rootNode.addChildNode(node)
```

Run the app. You should now be able to place furniture on detected horizontal surfaces by tapping on any "Place furniture here" grid. The default furniture is the bookcase, but you can select which furniture to place in the **Furniture catalog** tab.

9) That's it!

Bravo! You've just built a furniture layout app, and along the way, you've also developed a good understanding of ARKit surface detection, including:

- Responding to ARKit's detection of horizontal and vertical surfaces
- Responding to ARKit's revision of horizontal and vertical surfaces that it has detected
- Responding to when ARKit decides that previously detected horizontal and vertical surfaces are no longer there
- Taking the 2D coordinates of a user's tap on an AR object displayed on the screen and translating them into real-world 3D coordinates
- Drawing rendered AR objects

It's time to move on to the next topic: recognizing known 2D images with the power of ARKit.

ARKit Workshop: Demo 3

By Joey deVilla

In this demo, you'll code up **BaedekAR** (the name is derived from *Baedeker*, an old term for "tourist guidebook"), a basic version of an actual app that'll be used in museums and art galleries. With the app, you can point your device at artwork that it "knows", and it'll draw its name in AR text and display a hotspot that you can tap to find out more about the artwork. As a protective measure, it can also block certain disturbing works of art (such as a pictures of a creepy clown) with a more soothing image.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo3/starter**. To keep things simple, we've set things up so that you'll do all your coding in just one place: **ViewController.swift**.

1) Import the reference images

First, we'll need a place to store the reference images. Open **Assets.xcassets** and add a new AR resource group. Do this by clicking on the **+** button in the list of assets, and then selecting **New AR Resource Group** from the menu that appears:

You now have a new AR resource group. You should see something like this:

Switch to Finder. In the starter project folder (**Demo3/starter/BaedekAR/BaedekAR**), find and open the folder named **Reference images**. You should see something like this:

Drag the files from the **Reference images** folder into the AR resource group you just created and drop them into the area marked "No AR images — Drag and drop images to add new AR images."

You should now see something like this:

2) Specify the size of the real-world images

ARKit uses the reference images to know what real-world images to look for. However, knowing what to look for isn't enough; ARKit also needs to know the size of each reference image's real-world counterpart. Knowing the size of the real-world image makes it easier for ARKit to tell how far the image is from the camera.

In this exercise, you'll use the app to detect images on your MacBook monitor, so for simplicity's sake, we'll set the width of all the corresponding real-world images to 30 cenitmeters (about 12 inches).

While still looking at the **AR Resources** group in **Assets.xcassets**, make sure that the Attributes inspector is open. Select each reference image and set its width to 0.3 meters. Xcode will automatically set its height based on the image's width-to-height ratio.

3) Import the reference images

At the start of the `startARSession()` method (located in the **2D image detection** section), implement *"Make sure that we have an AR resource group":*

```
// Make sure that we have an AR resource group.
guard let referenceImages =
ARReferenceImage.referenceImages(inGroupNamed: "AR Resources",
                                                                bundle: nil)
else {
  fatalError("This app doesn't have an AR resource group named "AR
Resources"!")
}
```

4) Let's detect those reference images

In order to detect those imported reference images, we need to set up the AR configuration object to do just that. Add these lines to the `startARSession()` method under the *"Set up the AR configuration"* comment:

```
// Set up the AR configuration.
config.worldAlignment = .gravityAndHeading
config.detectionImages = referenceImages
```

Scroll down to the `renderer(_:didAdd:for:)` method and find this code near the beginning...

```
let isArtImage = true
var statusMessage = ""
```

...and replace it with this:

```
guard let imageAnchor = anchor as? ARImageAnchor else { return }
let referenceImage = imageAnchor.referenceImage
let imageName = referenceImage.name ?? ""
let isArtImage = self.isArtImage(imageName)
var statusMessage = ""
if isArtImage {
  statusMessage = "Found “\(artworkDisplayNames[imageName] ??
"artwork")”."
}
```

Run the app, open any one of the reference images on your computer in Preview or your favorite image editor, and point your device at it. You'll see the name of the image appear in the status indicator.

5) Make tappable hotspots for the detected images

Go back to the `renderer(_:didAdd:for:)` method (in the **2D image detection** section) and find the `if isArtImage` statement and implement *"If the detected artwork is one that we'd like to highlight":*

```
// If the detected artwork is one that we'd like to highlight (and one
which we'd
// like the user to tap to find out more), draw an “artwork” plane and
// the name of the artwork over the image.
planeNode = self.createArtworkPlaneNode(withReferenceImage:
referenceImage,
                                        andImageName: imageName)
let nameNode = self.createArtworkNameNode(withImageName: imageName)
node.addChildNode(nameNode)
```

Then implement *"Rotate the newly-created plane":*

```
// Rotate the newly-created plane by 90 degrees clockwise around the x-
axis
// so that it's vertical.
planeNode.eulerAngles.x = -.pi / 2
```

And just below that, implement *"Add the plane node to the scene":*

```
// Add the plane node to the scene.
node.addChildNode(planeNode)
```

Scroll down to the `createArtworkPlaneNode(withReferenceImage:andImageName:)`

method. Uncomment the code after the comment that reads *"*"Flash" the plane so the user is aware that it's now available"* so that it looks like this:

```
// "Flash" the plane so the user is aware that it's now available.
// Called when the plane first appears.
let flashPlaneAction = SCNAction.sequence([
  .wait(duration: 0.25),
  .fadeOpacity(to: 0.85, duration: 0.25),
  .fadeOpacity(to: 0.25, duration: 0.25),
  .fadeOpacity(to: 0.85, duration: 0.25),
  .fadeOpacity(to: 0.25, duration: 0.25),
  .fadeOpacity(to: 0.85, duration: 0.25),
  .fadeOpacity(to: 0.25, duration: 0.25),
])
```

Then find `Draw the plane` comment and replace the `return SCNNode()` line so that the code looks like this:

```
// Draw the plane.
let plane = SCNPlane(width: referenceImage.physicalSize.width * 1.5,
                     height: referenceImage.physicalSize.height * 1.5)
let planeNode = SCNNode(geometry: plane)
planeNode.opacity = 0.25
planeNode.name = imageName
planeNode.runAction(flashPlaneAction)
return planeNode
```

Once again, run the app, and open any one of the reference images on your computer in Preview or your favorite image editor. This time, when you point your device at one of the images, a translucent plane which will flash into existence over that image. If you tap the plane, you'll be taken to a screen that shows you details about the image.

6) Annotate the detected images with text

Now that we have those tappable hotspots, let's annotate them with the names of the corresponding works of art with 3D AR text. We'll do this in the `createArtworkNameNode(withImageName:)` method in the **2D image detection** section.

Uncomment the code after the *"Create the text node"* comment so that you have this:

```
// Create the text node.
let artworkNameText = SCNText(string: artworkDisplayNames[imageName],
                              extrusionDepth: CGFloat(0.02))
artworkNameText.font = UIFont(name: textFont, size:
textSize)?.withTraits(traits: .traitBold)
artworkNameText.alignmentMode = kCAAlignmentCenter
artworkNameText.firstMaterial?.diffuse.contents = UIColor.orange
artworkNameText.firstMaterial?.specular.contents = UIColor.white
```

```
artworkNameText.firstMaterial?.isDoubleSided = true
artworkNameText.chamferRadius = CGFloat(textDepth)
let artworkNameTextNode = SCNNode(geometry: artworkNameText)
artworkNameTextNode.scale = SCNVector3Make(textScaleFactor,
textScaleFactor, textScaleFactor)
artworkNameTextNode.name = imageName
```

To make these annotations easier to read, you'll set them up so that they're always turned to face the user by using a *Billboard constraint*. First, you must set up the annotation so that it pivots around the y-axis in its horizontal center. Implement *"Make the text node pivot in its middle around the y-axis":*

```
// Make the text node pivot in its middle around the y-axis.
let (minBound, maxBound) = artworkNameText.boundingBox
artworkNameTextNode.pivot = SCNMatrix4MakeTranslation((maxBound.x -
minBound.x) / 2,
                                                      minBound.y,
                                                      0)
```

Then add this code after the comment that begins with *"Ensure that the marker text is always readable"* so that it looks like this:

```
// Ensure that the artwork name is always readable with a Billboard
constraint,
// which constraints a SceneKit node to always point towards the camera.
let billboardConstraint = SCNBillboardConstraint()
billboardConstraint.freeAxes = SCNBillboardAxis.Y
artworkNameTextNode.constraints = [billboardConstraint]
```

Finally, change the last line of the method so that it returns `artworkNameTextNode` instead of a new `SCNNode`:

```
return artworkNameTextNode
```

7) Now, let's block the creepy clown image with a soothing picture of Ray

Let's go back to the `renderer(_:didAdd:for:)` method (it's in the **2D image detection** section). Find the *first* `if isArtImage` statement...

```
if isArtImage {
  statusMessage = "Found “\(artworkDisplayNames[imageName] ??
"artwork")”."
}
```

...and expand upon it with an `else` statment:

```
if isArtImage {
  statusMessage = "Found “\(artworkDisplayNames[imageName] ??
```

```
"artwork")"."
} else {
  statusMessage = "Unpleasant image blocked."
}
```

Then, in the *second* `if isArtImage` statement, in the `else` statement, implement *"If the detected artwork is one that we'd like to obscure":*

```
// If the detected artwork is one that we'd like to obscure,
// draw a "blocker" plane over the image.
planeNode = self.createBlockerPlaneNode(withReferenceImage:
referenceImage,
                                        andImageName: imageName)
planeNode.name = self.blockedName
```

Scroll down to the `createBlockerPlaneNode(withReferenceImage:andImageName:)` method. Then find the *"Draw the plane"* comment and replace the `return SCNNode()` line that the code becomes:

```
// Draw the plane.
let plane = SCNPlane(width: referenceImage.physicalSize.width * 2.0,
                     height: referenceImage.physicalSize.height * 2.0)
let planeNode = SCNNode(geometry: plane)
planeNode.geometry?.firstMaterial?.diffuse.contents = UIImage(named:
"ray")
planeNode.name = imageName
return planeNode
```

Open the creepy clown image on your computer in Preview or your favorite image editor, run the app and point your device at it. You won't see the clown for long, as its scary image will quickly be replaced by Ray's comforting face.

8) That's it!

Nicely done! You've just built a basic version of museum app that identifies known drawings, paintings, photographs, and other two-dimensional images. It tags certain recognized images with AR text and a translucent overlay that can be tapped so that the user can get more information about them. It obscures unwanted images with the soothing picture of Mr. Ray Wenderlich.

In the process, you covered:

- Creating an AR resource group for known 2D images and importing those images
- Responding to the detection of known images
- Creating a tappable hotspot for detected images
- Drawing AR text that is always readable to the user
- Blocking known unwanted images

You've now graduated from the section of the workshop that covers solely the features built into ARKit. We'll now move into augmenting ARKit by combining it with Core ML.

76 ARKit Workshop: Demo 4

By Joey deVilla

In this demo, you'll build **Vision Quest**, an app that combines ARKit with CoreML to identify (or more accurately, *attempt* to identify) real-world objects and specify what and where they are with AR markers. It's satisfying when the app properly identifies objects, and *hilarious* when it doesn't.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Begin work with the starter project in **Demo4/starter**. To keep things simple, we've set things up so that you'll do all your work in just one place: **ViewController.swift**.

1) Respond to the user's taps on the screen

When the user taps on a feature point, we want the app to respond by drawing AR text over that point.

Enter the following code into the `handleTap(gestureRecognize:)` method:

```
let screenCenter = CGPoint(x: sceneView.bounds.midX, y:
sceneView.bounds.midY)
let hitTestPoints = sceneView.hitTest(screenCenter, types:
[.featurePoint])

if let closestResult = hitTestPoints.first {
  // Get the coordinates of the real-world point that corresponds to
  // where the user tapped on the screen.
  let transform = closestResult.worldTransform
  let positionColumn = transform.columns.3
  let worldCoord = SCNVector3(positionColumn.x, positionColumn.y,
positionColumn.z)
```

```
    // Draw label at that point.
    let node = createMarkerNode(text: "Got one!",
                                confidence: 1)
    node.position = worldCoord
    sceneView.scene.rootNode.addChildNode(node)
  }
```

Run the app. You should now be able to tap on feature points, and the will tag the that feature point with the text **Got one! [100%]**.

We're now done with the ARKit of the project. Let's now make some machine learning magic by incorporating Core ML.

2) Initialize the vision model

Go to the **Intializers** section and find the `initVision()` method. Enter the following below the comment that begins with *"Load the vision model"* so that the code looks like this:

```
  // Load the vision model. You can find more vision models at Apple's Core
  ML page:
  // https://developer.apple.com/machine-learning/
  guard let model = try? VNCoreMLModel(for: MobileNet().model) else {
    return false
  }
```

This loads the **MobileNet** model, which detects the dominant object in an image and tries to see determine which of its 1,000 categories (which include people, animals, vehicles, and so on) that object fits into.

Then implement *"Set up a request for Core ML to classify images using the vision model":*

```
  // Set up a request for Core ML to classify images using the vision
  model.
  let classificationRequest = VNCoreMLRequest(model: model,
                                              completionHandler:
  classificationCompleteHandler)
  classificationRequest.imageCropAndScaleOption =
  VNImageCropAndScaleOption.centerCrop
  visionRequests = [classificationRequest]
```

And finally, change the final line of `initVision()` to return `true`:

```
  return true
```

Run the app. You should see one change: the "Couldn't load the ML model..." message no longer appears in the status area at the top of the screen.

3) Unleash SkyNet! (or: Harness Core ML's vision capability)

Scroll down to the section ominously named **SkyNet starts here!!!** and implement the `renderer(_:willRenderScene:atTime:)` method by putting the following into it:

```
evaluateCurrentFrame()
```

Let's now implement `evaluateCurrentFrame()`. Implement *"Get the image from the current frame":*

```
// Get the image from the current frame.
guard let arFramePixelBuffer =
(sceneView.session.currentFrame?.capturedImage) else { return }
let image = CIImage(cvPixelBuffer: arFramePixelBuffer)
```

Then add this code after the comment that begins with "Use the image to make an image request handler" so that you have the following:

```
// Use the image to make an image request handler
// and then pass it to Core ML to see if can guess
// what it is.
let imageRequestHandler = VNImageRequestHandler(ciImage: image, options:
[:])
do {
  try imageRequestHandler.perform(self.visionRequests)
} catch {
  print(error)
}
```

Finally, let's complete the `classificationCompleteHandler(request:error:)` method. First, let's capture the name and confidence of what Core ML thinks the object currently in the camera's view is. Do this by implementing *"Capture the best guess":*

```
// Capture the best guess.
let topObservation = observations.first as! VNClassificationObservation
currentBestGuess = topObservation.identifier
currentBestGuessConfidence = topObservation.confidence
```

We'll use these values later for the AR text markers. In the meantime, we want to update the status area at the top of the screen with the Core ML's top 3 guesses of what's currently in the camera's view. Do this by uncommenting the *"Display the top 3 guesses"* code so that you have the following:

```
// Display the top 3 guesses in both the debug console and the
observations label.
let top3Guesses = observations[0...2]
  .compactMap { $0 as? VNClassificationObservation }
  .map { "\($0.identifier) \(String(format: " [%1.0f%%]", $0.confidence *
```

```
100))" }
  .joined(separator: "\n")
print(top3Guesses)
print("--")
DispatchQueue.main.async {
  self.observationsLabel.text = top3Guesses
}
```

Run the app and point it at various objects. Core ML's top 3 guesses of what's in the camera's view, based on the MobileNet model, will be displayed in the status area at the top of the screen. Some of those guesses will be right, but many of them will be wrong (and often quite funny).

4) Enable "tap to tag"

Let's make it so that when the user taps on an object, it gets tagged with Core ML's best guess using AR text.

Go back to the `handleTap(gestureRecognize:)` method (in the **UI** section) and look for the *"Draw label at that point"* comment. The line immediately after that comment should read as follows:

```
let node = createMarkerNode(text: "Got one!",
                            confidence: 1)
```

Change it to this:

```
let node = createMarkerNode(text: currentBestGuess,
                            confidence: currentBestGuessConfidence)
```

Run the app and tap on objects. It'll now tag objects with Core ML's best guesses. Prepare for hilarity to ensue, as many of its guesses will be quite wrong!

While you're at it, make a note of the frame rate in the statistics bar at the bottom of the screen. On an iPhone X, the currently fastest ARKit-capable phone, it should be in the range of 30 fps (frames per second).

5) Speed up the frame rate

Let's see if we can't make the frame rate a little faster by harnessing dispatch queues.

First, **delete** the `renderer(_:willRenderScene:atTime:)` method. We won't be using it anymore. Instead, we'll make use of a dispatch queue that we'll set up in `startCoreMLVisionLoop()`, which should now be at the top of the **SkyNet starts here!** section.

Add the following code to `startCoreMLVisionLoop()`:

```
dispatchQueueML.async {
  while true {
    self.evaluateCurrentFrame()
  }
}
```

We'll call `startCoreMLVisionLoop()` when the app starts. Scroll up to `viewDidLoad()` and find this `if` statement — it's in the block of code that calls a bunch of methods whose name begins with `init`:

```
if initVision() {

} else {
  observationsLabel.text = "Couldn't load the ML model. Better check the
code..."
}
```

Add a call to `startCoreMLVisionLoop()` in the blank line after `if initVision()` so that the `if` statement looks like this:

```
if initVision() {
  startCoreMLVisionLoop()
} else {
  observationsLabel.text = "Couldn't load the ML model. Better check the
code..."
}
```

Run the app again. Even on the (relatively) pokey iPhone 6S, you should be faster framerates.

6) That's it!

Bravo! You've just combined machine learning, computer vision, and augmented reality in a single project, and in fewer than 250 lines of code (a lot of which are white space and comments)!

You've also learned about:

- Loading a vision model and using it to build a request to classify its contents
- Taking the image from the current frame in an AR scene and feeding it to an image request handler
- Getting the results of a Core ML classification and displaying them
- Creating AR text and ensuring that it always faces the user so they don't have to read backwards or upside-down text
- How terribly, hilariously wrong machine learning-powered vision models can be

7) Challenges

MobileNet isn't the only vision model available. Apple's Core ML page has a number of ready-to-plug-in models available for download in the section marked *Working with Models*. They also show you how to roll your own.

The models listed on the CoreML page all return information in the same format: a list of observations, each containing a string containing the name or names of identified objects, and a number representing Core ML's confidence in a match.

What kind of place is this?

One of the models available for download from Apple's Core ML is **Places205-GoogLeNet**. Instead of identifying the domninant object in an image, it looks at the entire image and determines what kind of room or place is depicted.

We've already included the **Places205-GoogLeNet** model in the project. It's the **GoogLeNetPlaces.mlmodel** in your project's **ML models** folder, and it's already been made available to your project.

Go back to **ViewController.swift** and find this line...

```
guard let model = try? VNCoreMLModel(for: Inceptionv3().model) else {
```

...and switch to the **Places205-GoogLeNet** model by changing the line to:

```
guard let model = try? VNCoreMLModel(for: GoogLeNetPlaces().model) else {
```

Run the app and look at the status area at the top of the screen. You'll see that it no longer lists guesses for objects, but guesses about what kind of room is pictured in the camera's view. Try it in different rooms, and even outdoors (the office where I work is on a boardwalk, and this model knows what a boardwalk is). You'll find that it's pretty accurate.

Improving on MobileNet

Like MobileNet, the **Inception v3** model accepts an image and attempts to determine where it belongs in a set of 1000 categories (some of which are oddly specific). You can read more about this model in Rethinking the Inception Architecture for Computer Vision, a paper published by researchers at Cornell University in December 2015.

To make use of this model, go to Apple's Core ML page, find the **Inception v3** model, and download it.

Once it's downloaded, drag the downloaded file, **Inceptionv3.mlmodel** into your project's **ML models** folder. Select it in the Project Navigator and open the File Inspector in Xcode's right column. In the **Target Membership** list at the bottom of

the File Inspector, make sure that **VisionQuest** is checked. This makes sure that the model is available to your project.

Go back to **ViewController.swift** and go to the `initVision()` method (in case you've forgotten, it's in the **Initializers** section. Find this line...

```
guard let model = try? VNCoreMLModel(for: MobileNet().model) else {
```

and replace `MobileNet` with `Inceptionv3` so that the line reads like this:

```
guard let model = try? VNCoreMLModel(for: Inceptionv3().model) else {
```

Run the app. You should find that it's much better at identifying things.

Try working with other object-classification models

If you'd like to see how a large model performs, download the largest model on Apple's Core ML page, **VGG16**, which is just over 550MB.

Roll your own!

Feeling bold? Go to GitHub and check out Apple's Turi Create, an open source project of theirs that makes it easier to create your own custom machine leaning models.