
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

13: Getting Started with ARKit

This session will get you started in making the world your view controller! You'll build two apps that cover a lot of ARKit ground: a Bob Ross tribute that takes painting into the third dimension, and a recreation of that popular catalog app from everyone's favorite semi-disposable Swedish furniture company. 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!

37 Getting Started with ARKit: 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.

38 Getting Started with ARKit: 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

You've completed your first steps into AR development. Congratulations!

Conclusion

We hope you enjoyed the RWDevCon 2018 Tutorial Video Vault!

We also hope that our team's passion for iOS, Swift, Android, and Kotlin development has spread to you, and that you can take what you've learned here and put it into practice.

And thanks for connecting with us! As Tammy said in the keynote, “we are all better together.” We hope to see you at the next RWDevCon!

— Ray, Vicki, and the entire RWDevCon Team