optional

Unit 3A

Building AR Apps with Xcode

Augmented reality, or AR, is quickly becoming one of the most interesting areas of exploration in app development. Unlike virtual reality, where users are immersed in a 3D digital world, augmented reality allows them to position 3D digital objects in the real world, viewable through their device's camera.

ARKit is a cutting-edge platform for developing augmented reality (AR) apps for iPhone and iPad. In this unit, you'll learn how ARKit can help you create amazing experiences.

Ready to get started? In order to build apps that take advantage of ARKit, you must be connected to a physical device that runs iOS 11 or newer. Use the Device Compatibility chart to learn more.

SDK Lessons

- Building AR Apps with Xcode
- Introduction to SceneKit
- Finding Flat Surfaces
- Interacting with Augmented Reality
- Image Recognition in ARKit

What You'll Build

The AR Drawing app lets the user place 3D shapes and models anywhere in the world to create a custom scene. Not only can the objects be placed directly in front of the camera, they can be attached to walls, tables, and other flat surfaces.

Lesson 3A.1

The Augmented Reality Template

Welcome to your first lesson on ARKit. Whether you're looking to build a new app centered around augmented reality or you want to add augmented reality to an existing app, you'll need to learn some additional Xcode project configurations. Then you'll be set to dig into the basics of the augmented reality framework.

What You'll Learn

- How to create a new augmented reality-based project using ARKit
- How to add ARKit into an existing Xcode project
- How SceneKit works alongside ARKit
- How to enable different debugging options for AR apps

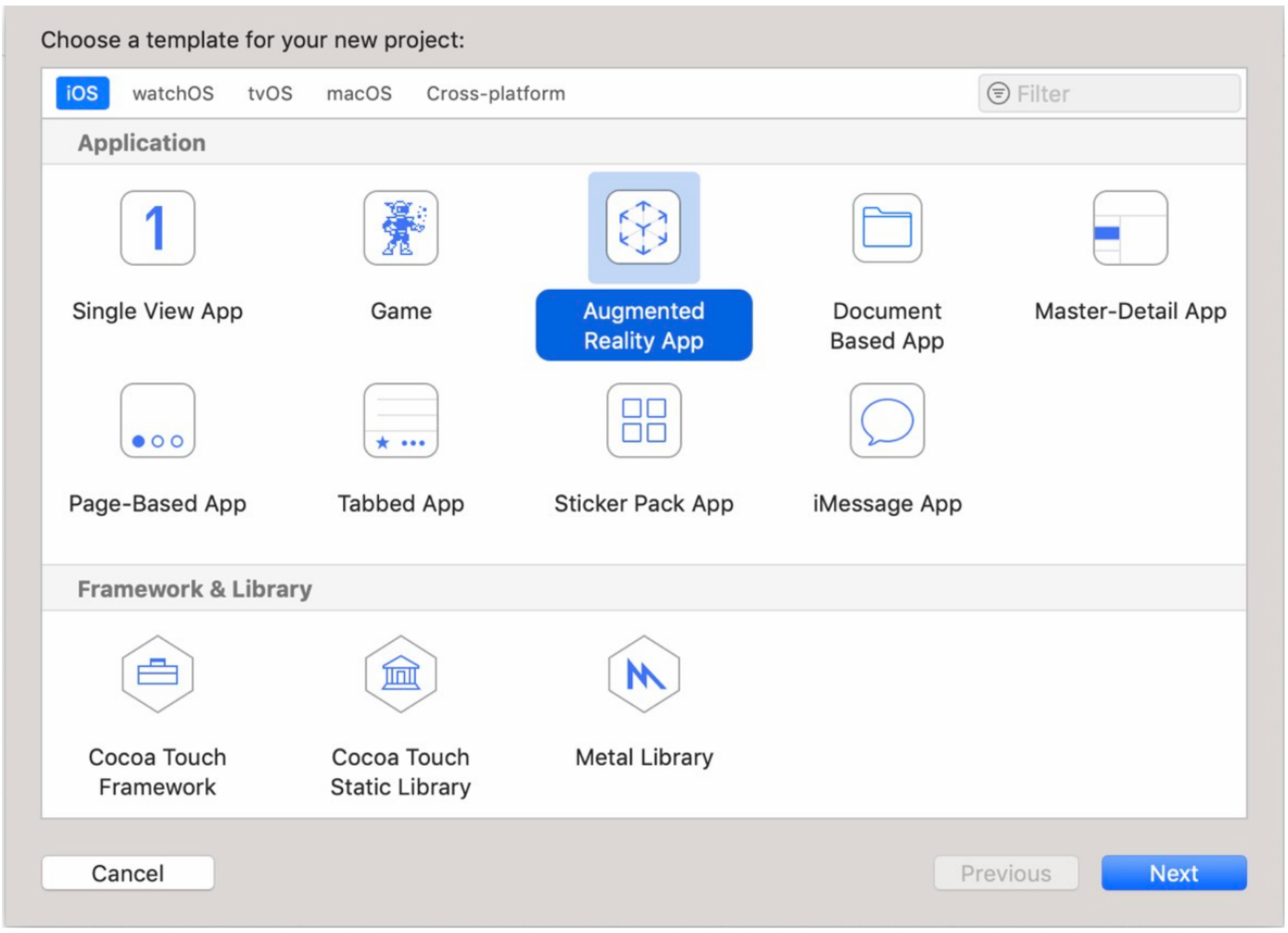
Vocabulary

- `ARConfiguration`
- ARKit
- `ARSCNView`
- `ARSession`
- origin
- scene
- SceneKit

Related Resources

- WWDC 2017 Introducing ARKit: Augmented Reality for iOS

Open Xcode and create a new project. To get started quickly with ARKit, you can use the new Xcode template called Augmented Reality App. This template conveniently includes the configuration files and code required for your AR projects. Select this template over the standard Single View App template that you've used in previous lessons.



On the next screen, you'll need to specify a new option, Content Technology. Your choice —SceneKit, SpriteKit, or Metal—determines which framework Xcode will use in conjunction with ARKit to add objects into your augmented world. The SpriteKit framework is focused on working with 2D assets, while SceneKit and Metal are focused on 3D. Throughout this guide, you'll only be using SceneKit. Select it now.

Name your project "ARKit-Template." Connect your physical device to your computer, then set it as the target device, rather than Simulator. Before you can proceed, you may need to update iOS on your device. Testing your app will only work if your device is running an iOS version that supports ARKit; otherwise the device can’t be selected as the target.

When your app launches for the first time, it will ask for permission to use the device's camera. Without permission, ARKit can't begin a new session. After you've confirmed access, the rear-facing camera is enabled, and you'll see a 3D ship floating in the scene as you move your device. You'll also see a bar near the bottom of the screen, where you can select the plus sign (+) on the left to reveal statistics regarding memory and performance.

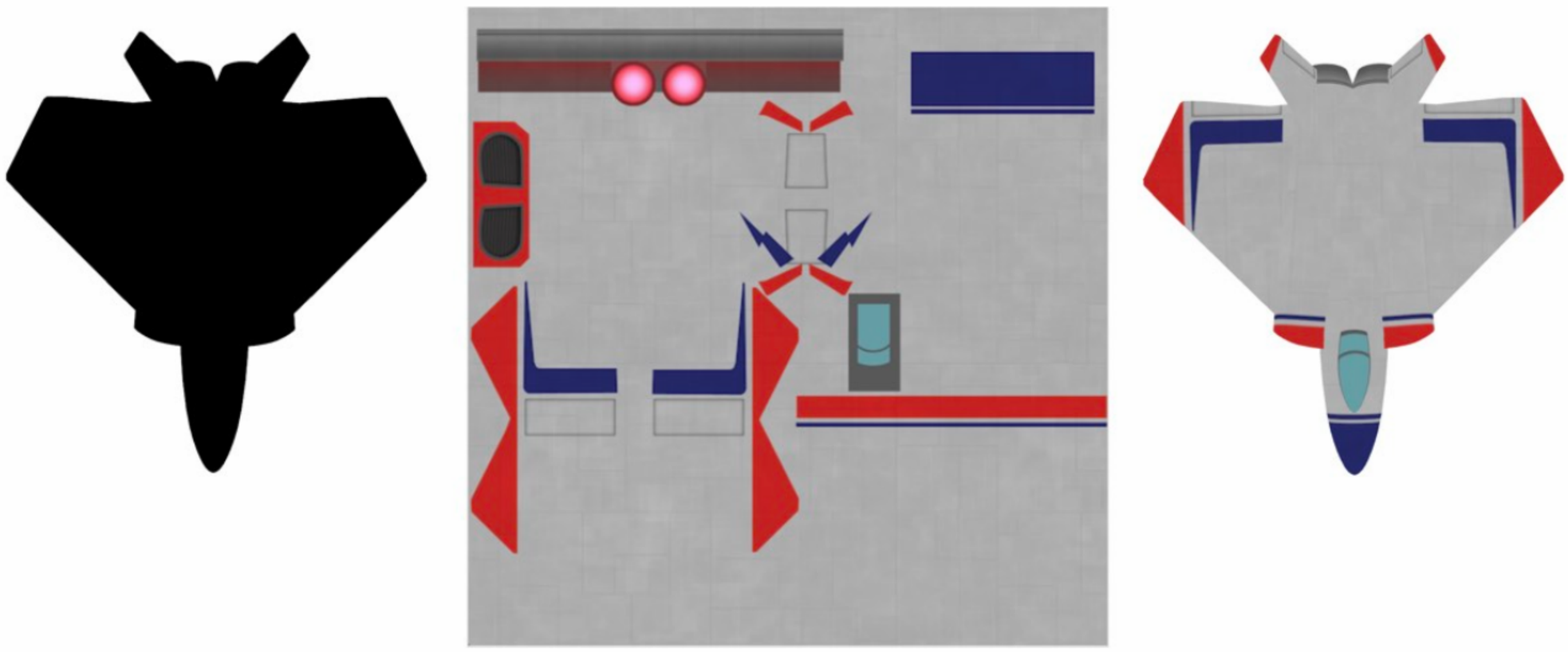
Take a few minutes to experiment with how ARKit tracks your device's position and orientation in the real world. You may notice that the ship doesn't appear fixed in space as you move around. Are there certain conditions that impact the fidelity of the experience? The framework is doing a lot of heavy lifting in the background to enable AR, and it relies on having an environment rich with visual cues. Try changing your environment to observe how it affects the app.

Dissecting the Project Template

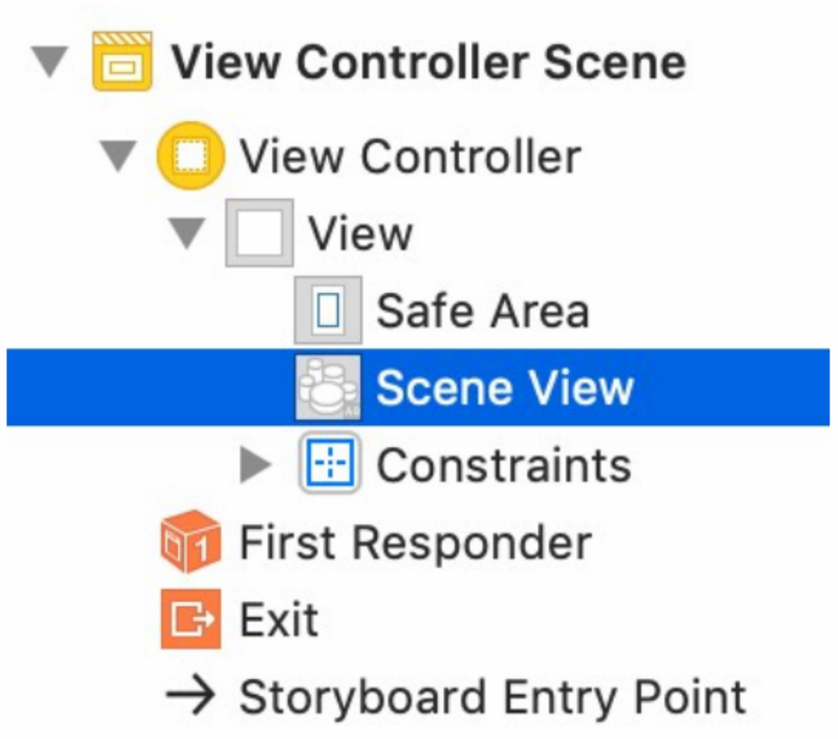
So what's the difference between this Xcode project and ones you've created in previous lessons? Before jumping into the Swift code, use the project navigator to examine the other files for your ARKit-Template app. Compare them to the files you're accustomed to seeing with the Single View App template.

When you think you've finished comparing, see if your findings match the following list of differences:

- The `art.scnassets` catalog contains all the SceneKit assets included in an app. The first asset, `ship.scn`, is a SceneKit scene file, which contains details about the 3D model, including its size, colors, and positioning. Typically, you'd have a 3D artist create these files, but you can create your own inside the interface. The second asset, `texture.png`, is the texture that will be applied to the 3D object. For example, you can imagine this flat image being perfectly bent and glued all the way around the ship model.



- In `Main.storyboard`, a subview has been automatically added to the view controller's view. This subview is of type `ARSCNView`, and it has been configured to fill the entire view controller's view. This special subclass displays the user's camera and allows for any 3D objects to be added to the scene.



- To inspect the `Info.plist` file, select it from the project navigator, then right-click inside its contents and select Show Raw Keys/Values. There are two additional entries in this file when compared to the Single View App template. The first is a key, `NSCameraUsageDescription`, whose value is the text the user will see when prompted for camera access. If you deselect Show Raw Keys/Values, this key will show as "Privacy - Camera Usage Description". The second entry is the additional `arkit` value under the `UIRequiredDeviceCapabilities` key. With Show Raw Keys/Values deselected, this key shows as "Required device capabilities". This informs the App Store that your app uses ARKit, and ensures your app is only available for supported devices.

Privacy - Camera Usage Description	String	This application will use the camera for Augmented Reality.
Launch screen interface file b...	String	LaunchScreen
Main storyboard file base name	String	Main
▼ Required device capabilities	Array	(2 items)
Item 0	String	armv7
Item 1	String	arkit
Status bar is initially hidden	Boolean	YES

BASICS OF ARKIT AND SCENEKIT

Now that you understand the project structure, open `ViewController.swift` to see what code you'll need to get a basic AR app up and running. At the top of the file are two new `import` lines. Since the `ViewController` class needs to work with classes that are available within ARKit and SceneKit, those frameworks have been imported for you.

```
import UIKit
import ARKit
import SceneKit
```

ARSCNView

The `sceneView` outlet is connected to the `ARSCNView` that you saw in Interface Builder. It has a `scene` property, which is the scene to be displayed in the view. In `viewDidLoad()`, the property is set to the scene that includes the 3D ship. The view's `showsStatistics` Boolean value is set to `true`, which enables the bottom bar that you saw earlier.

```
    @IBOutlet var sceneView: ARSCNView!

    override func viewDidLoad() {
        super.viewDidLoad()

        // Set the view's delegate
        sceneView.delegate = self

        // Show statistics such as fps and timing information
```

```
        sceneView.showsStatistics = true

        // Create a new scene
        let scene = SCNScene(named: "art.scnassets/ship.scn")!

        // Set the scene to the view
        sceneView.scene = scene
    }
```

ARSession and ARConfiguration

When the `ARSCNView` is loaded, it initializes its own session property of type `ARSession`, which manages motion-tracking and processes the camera image data. But before a session can do all that, you must call the `run(_:)` function and supply an `ARConfiguration` as the argument. Since you'll call `run(_:)` whenever the `ARSCNView` is displayed, it's a good idea to call this function in `viewWillAppear(_:)`. To track your device's position, you'll need to initialize and run the `ARWorldTrackingConfiguration`. That way, ARKit can maintain the correct size and position of the ship.

```
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)

        // Create a session configuration
        let configuration = ARWorldTrackingConfiguration()

        // Run the view's session
        sceneView.session.run(configuration)
    }
```

What about when the `ARSCNView` *isn't* displayed? If the camera view is no longer the focus, there's no reason to continue running the session. To prevent unnecessary work, you can `pause()` the session whenever the view is no longer displayed. By calling the

function in `viewWillDisappear(_:)`, you'll trigger a `pause()` when you dismiss this view controller or when you present a new one.

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    // Pause the view's session
    sceneView.session.pause()
}
```

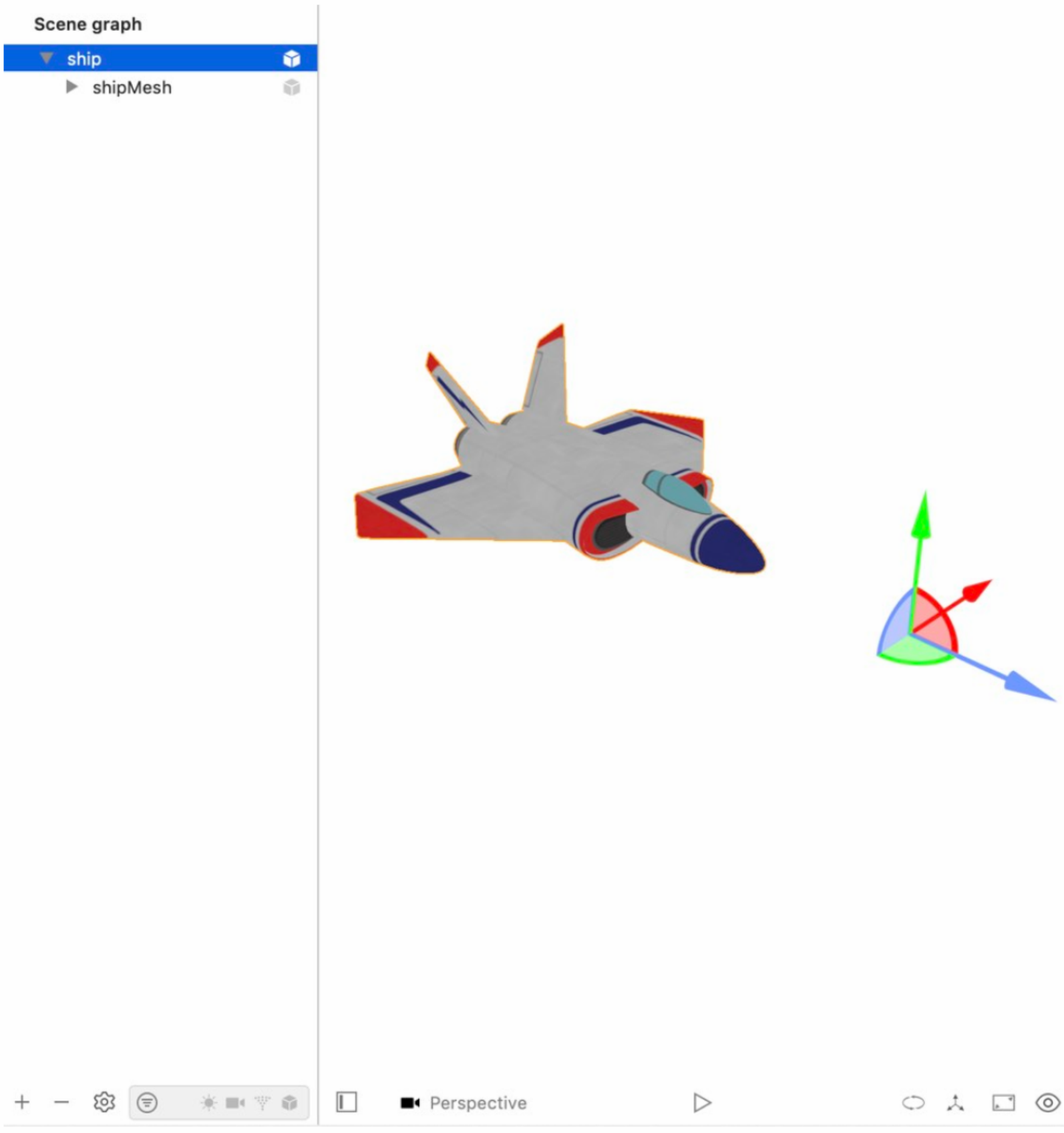
ARSCNViewDelegate

There are a few additional pieces of code inside the Augmented Reality App template. The view controller declares itself an `ARSCNViewDelegate`, assigns itself as the delegate of sceneView inside `viewDidLoad()`, and has some empty methods at the bottom of the class definition. This allows the `ARSCNView` to send information about events related to the scene—when objects are added, when the session is interrupted—to the view controller. It's as if the view controller tells the scene view, "Let me know when those events take place."

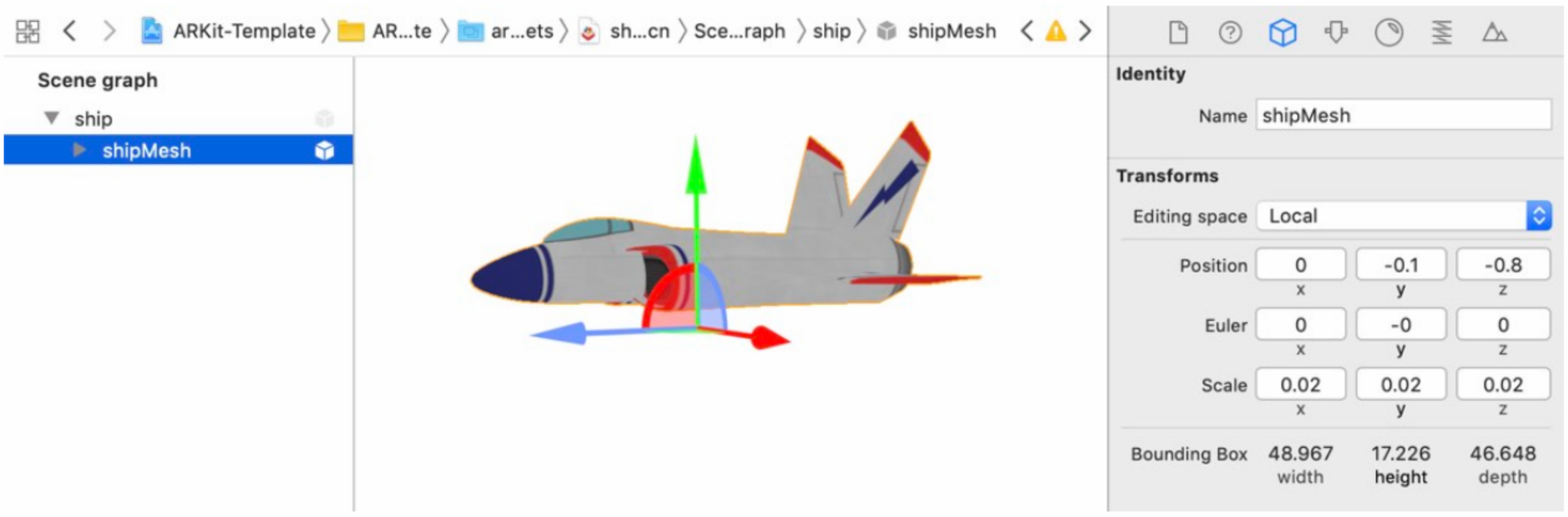
For more information on delegates, refer to Unit 4.

SCENE POSITIONING

As you move your device around, the ship uses the `ARWorldTrackingConfiguration` to maintain its position in 3D space. But how does the `ARSCNView` know where to place the ship initially? Open the `ship.scn` file and select the top-level node, "ship," in the scene graph view. (You can toggle this display on/off using the button in the lower left of the editor.)



The ship is positioned on a grid that represents the world coordinate system. Notice the set of three arrows? That's the origin of the node, which is aligned with the origin of the world: `(0,0,0)` in 3D space. The x-axis is represented by the red line, y-axis in green, and z-axis in blue. If you were to travel in the direction of the red, green, or blue arrow, you would increase the corresponding value. Since the ship is positioned along the z-axis in the *opposite* direction of the arrow, it has a negative z-value. The values themselves can be found in the Node inspector in the utilities panel. Select "shipMesh" from the scene graph view, and observe that the z-value is 0.8.



If you've never done 3D computing before, you'll have to get used to how 3D coordinate systems work as you complete this unit.

Would it be helpful to see the origin in your app? An `ARSCNView` has a `debugOptions` property that can draw additional overlays, which might include the origin or 3D boxes around objects in the scene. Add the following line to `viewDidLoad()` to show the world origin.

```
sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin]
```

Build and run the app again. Notice how the ship is positioned in the correct location compared to the origin. It may seem weird that you're initially pointed in the *negative* z-direction. That's because the world origin is determined by wherever you are when you launch the app. It's also helpful to note that, unlike UIKit, the x- and y-axes are laid out according to the standard Cartesian coordinate system (which you may remember from math class) with y-coordinates increasing in value as you go up.

REVIEW QUESTIONS

- What unit of measurement does ARKit use to determine sizes and distances?

EXERCISES

- Start with a Single View App template, and adjust the project to use ARKit and SceneKit. When you run the app, the camera should be enabled and the world origin displayed.
- Modify the ARKit-Template app by adjusting the position of the ship to float directly above the world origin.
- Investigate the utilities panel in your ARKit-Template app to determine where the ship's texture is being applied. Modify the property so that the ship is solid orange.
- Use the Xcode documentation to make a list of all the available `ARSCNDebugOptions`. Experiment to see which options affect the Augmented Reality App template and what they do.

Lesson 3A.2

Introduction to SceneKit

In the previous lesson, you learned that the Augmented Reality App template provides you with a ship scene. Of course, as you create your own AR apps, you'll want to use other 3D models. In this lesson, you'll learn about the basic 3D objects available in SceneKit, how to add them to an augmented reality scene, and how to position them in relation to each other.

What You'll Learn

- How SceneKit objects are added and positioned within your scene
- How to create and load your own SceneKit files
- How to create SceneKit objects programmatically and visually
- How to apply basic lighting to any scene

Vocabulary

- depth fighting
- scene hierarchy
- `SCNLight`
- `SCNNode`
- `SCNVector`

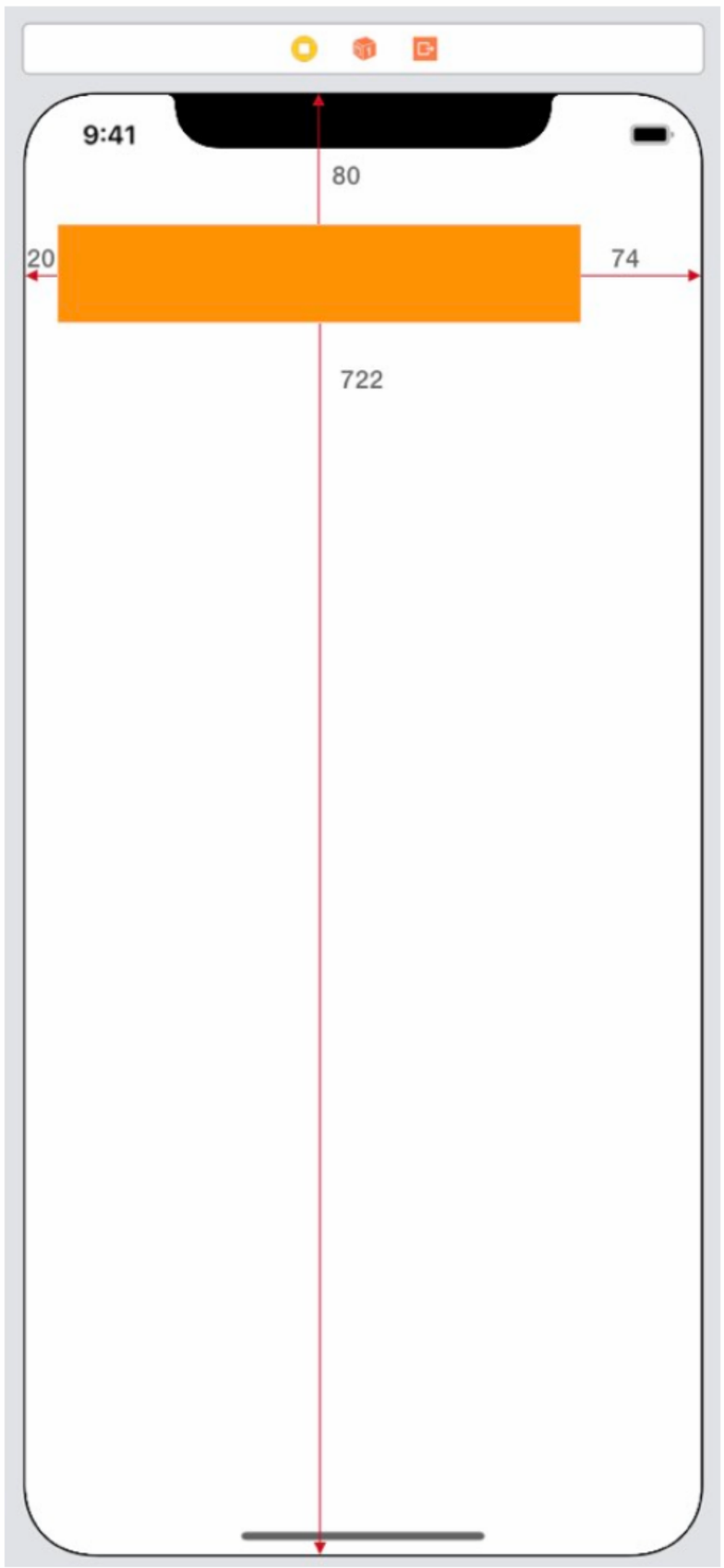
Related Resources

- WWDC 2012: Introducing SceneKit
- WWDC 2017: SceneKit What's New
- SceneKit Documentation

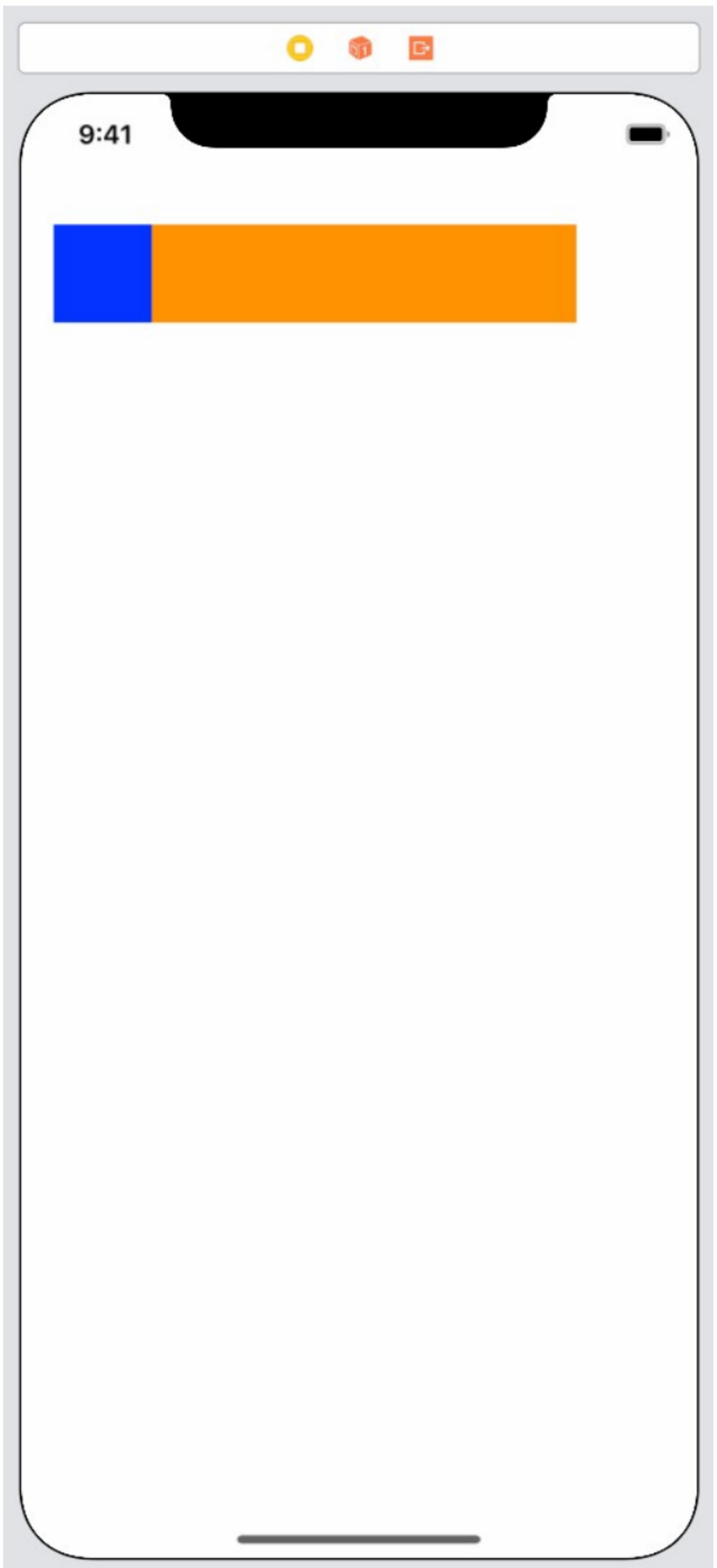
SCENEKIT OBJECTS

Working in 3D involves positioning objects in space. SceneKit organizes the world as a set of nodes in a scene hierarchy, and each node represents an object in 3D space. When you write code, you'll be using instances of the `SCNScene` and `SCNNode` classes to manage your scene. Each `SCNNode` has a position in 3D space, along with other properties, such as its shape (or geometry), lighting information, and relationship to other nodes.

As you may recall from previous lessons that use UIKit, every visual element (or view) that your user sees inherits originally from `UIView`. For a view to display, it requires three pieces of information: an x/y position, a width, and a height. Together, these pieces of data form the `frame` of the view. For example, in the following image, the orange rectangle is placed 20 pixels to the right and 40 pixels down, and has a width of 320 and a height of 60. Therefore, the `frame` property has a value of `(20, 40, 320, 60)`.



If you were to add a blue square with a frame value of `(0, 0, 60, 60)` as a subview to the orange rectangle, where would it appear? At first, you might think the blue square will appear in the top-left corner of the screen, but instead it's in the top-left corner of the original orange rectangle. This is because the `(0,0)` x/y position is *relative* to its parent view.



SceneKit positions nodes in a similar way—only in 3D instead of 2D. Within an ARKit app, there's a world origin at position `(0,0,0)`, which you saw in Lesson 1 by enabling the `showWorldOrigin` debug option. You can position nodes relative to the world origin

or, as with views and subviews, relative to the position of other nodes. With relative positioning, if the position of the top-level node changes, the child nodes will follow suit. In the Augmented Reality App template's `ship.scn` file, the `ship` node is the top-level node, and `shipMesh` is its only child node.

In SceneKit, rather than adding subclasses of `UIView` to the screen, you add nodes to the scene hierarchy, represented by `SCNNode` objects. A `SCNNode` describes an element in 3D space, including the position of the element, its shape (or geometry), and lighting information.

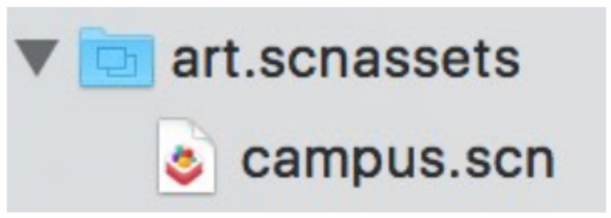
But a node doesn't necessarily have to be something the user sees. It could simply represent a position, acting as a "camera" from which to view the 3D world. For example, by switching the camera to an `SCNNode` positioned above the 3D object, you could switch from a front view of an object to a bird's eye view. When used in conjunction with ARKit, SceneKit's camera nodes are largely unused, since your device *is* the camera.

CREATING SCENES

As you learn how to create your own scenes, you'll put together the scene of a school campus. You'll build parts of the campus visually and you'll build others programmatically. As you gain experience, you'll decide which way you prefer to create scenes, and you'll discover reasons to use one method over the other.

Create a new project using the Augmented Reality App template, and name the project "SceneKit-Primitives." (If you like, you can delete the `ship.scn` and `texture.png` files, since you won't be using them in this lesson.) To create your own scene, right-click the `art.scnassets` folder in the project navigator, then select New File and rename the `.scn` file to `campus.scn`. The scene contains one camera node inside the scene graph. To start from scratch, go ahead and delete the node.

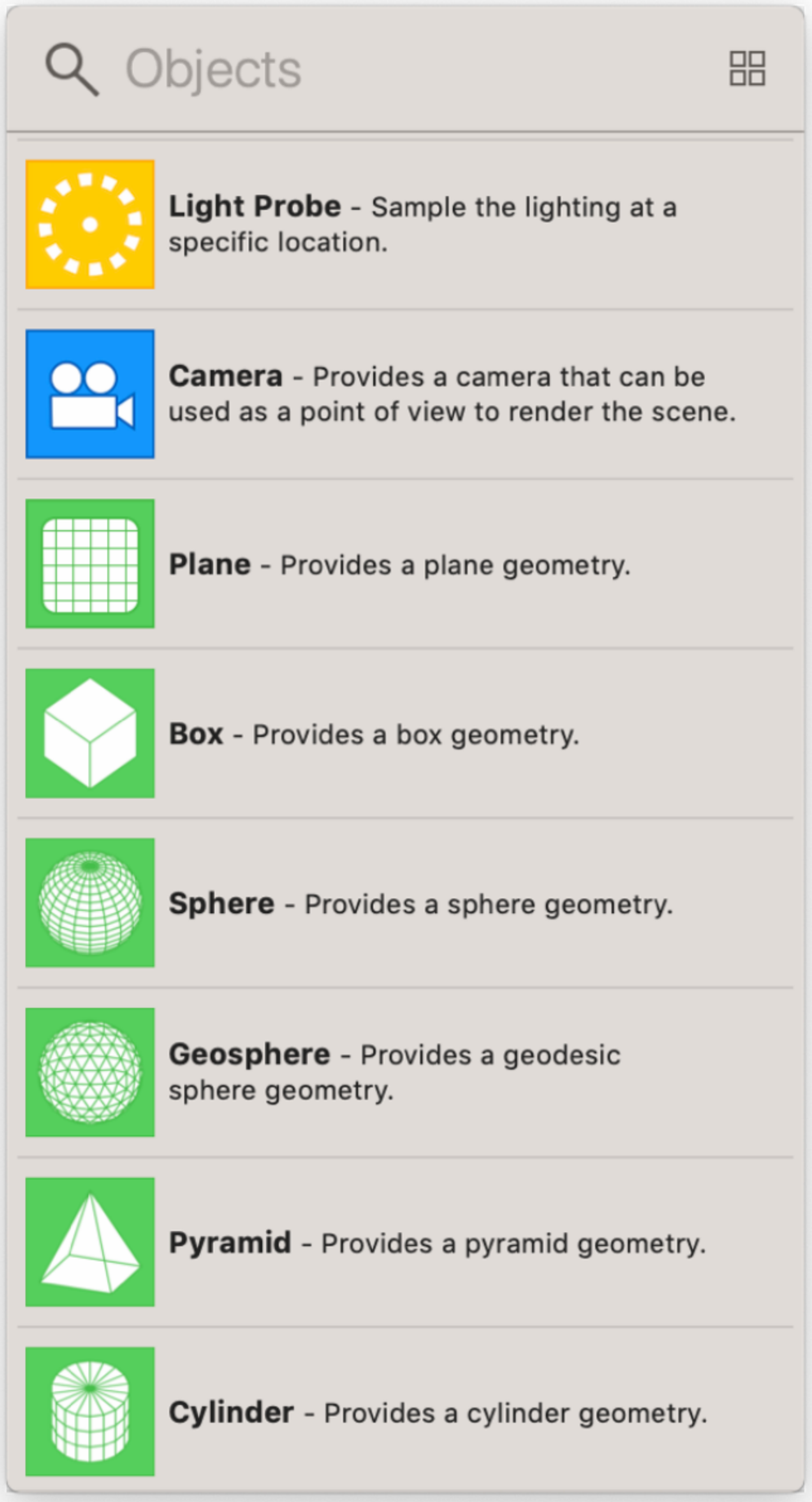
Navigate to the `viewDidLoad()` method in the `ViewController` definition, and change the code to load your new scene file, while enabling the world origin.



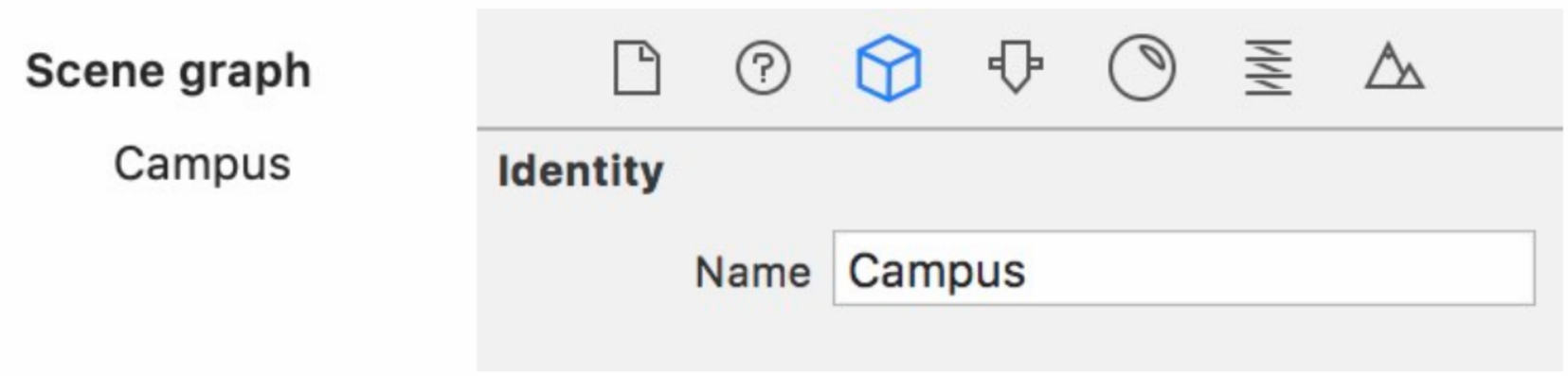
```
override func viewDidLoad() {
    super.viewDidLoad()

    // Set the view's delegate
    sceneView.delegate = self

    // Show statistics such as fps and timing information
    sceneView.showsStatistics = true

    // Show the world origin
    sceneView.debugOptions = [ARSCNDebugOptions.showWorldOrigin]

    // Load campus
    loadCampus()
}

func loadCampus() {
    // Create a new scene
    let scene = SCNScene(named: "art.scnassets/campus.scn")!

    // Set the scene to the view
    sceneView.scene = scene
}
```

Adding Nodes

Navigate to `campus.scn` again. Open the Object library to find a list of objects—including nodes, lighting, and geometric shapes—which you can add to the scene graph.



Locate the "Empty Node" in the Object library and drag it to the scene graph view. Under the Node inspector , name the node "Campus." This is the top-level node that you'll use to position many of the pieces of the scene.



How do you use code to add a `SCNNode` to a scene? In fact, it's a lot like adding a `UIView` to an existing view. You would start by initializing the object with the default initializer, `SCNNode()`. You would then give the node a position. Of course, in 3D, an x/y position isn't sufficient. The node's `position` property is of type `SCNVector3`, and the initializer uses three parameters: x, y, and z. By default, the values are `(0.0, 0.0, 0.0)`. You could change these values programmatically to suit your needs. For example, here's how you would create a node that's positioned 0.2 meters above the world origin:

```
let node = SCNNode()
node.position = SCNVector3(0.0, 0.2, 0.0)
```

Unlike with views, nodes aren't added as subviews; they're added as child nodes to an existing node. The scene contains a `rootNode` property that begins at `(0,0,0)`, the origin. With the `addChildNode` method, you can use this node to begin attaching new nodes.

```
sceneView.scene.rootNode.addChildNode(node)
```

For now, you've added the necessary node in the `campus.scn` file, so there's no need to add nodes in code. However, it's important to understand both methods, just as it's important to know how to work with views programmatically and with Interface Builder.

Now you have a node, but how do you make it visible? To add shape and color, you can assign the `geometry` value to an `SCNGeometry` object. But rather than modifying an empty node, an easier approach is to add shapes from the Object library using Interface Builder.

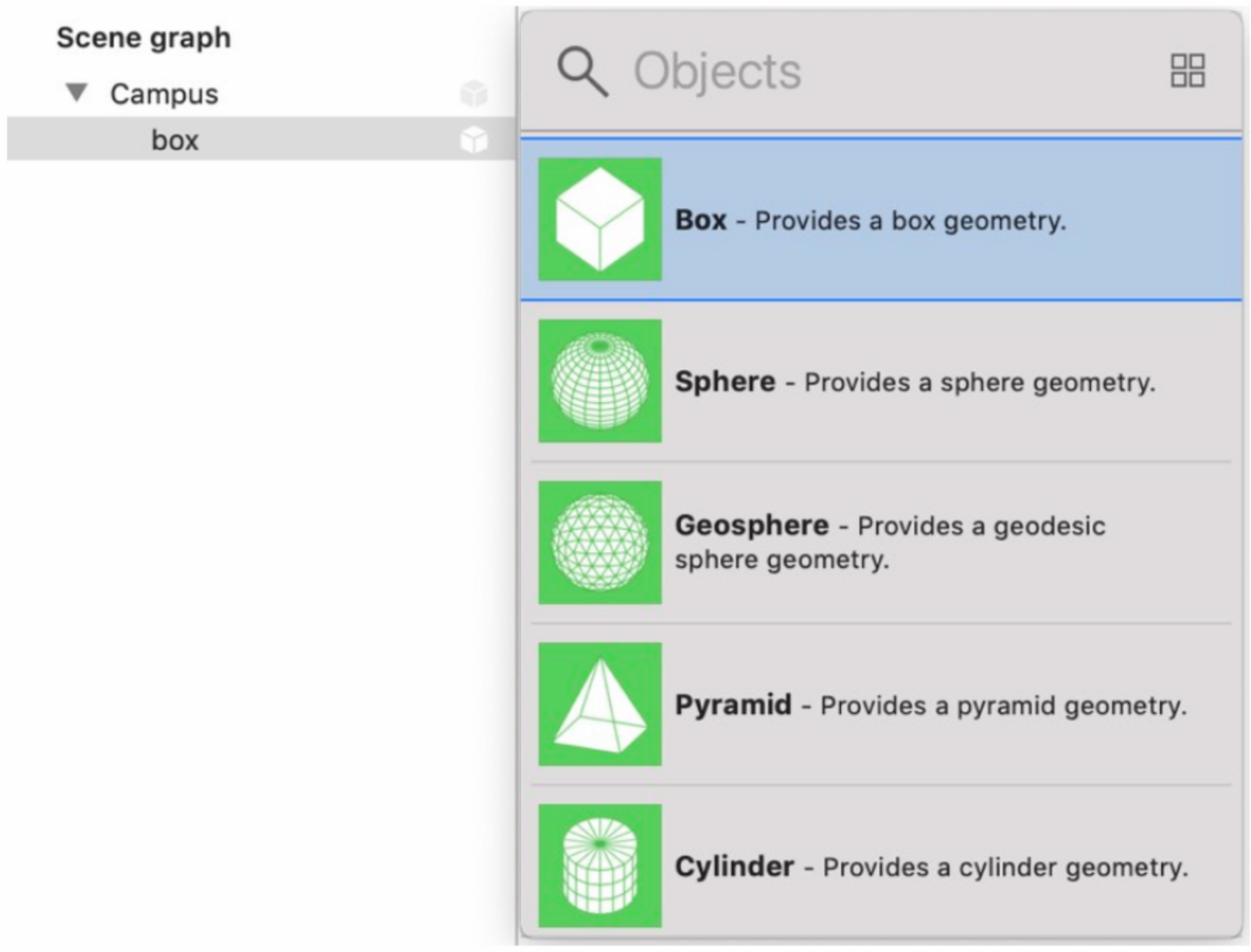
Basic Geometry Types

SceneKit provides a number of basic shapes that you can use in your scenes. These shapes are all subclasses of `SCNGeometry`, which contains the model and its associated values, such as color and the material used (which affects lighting).

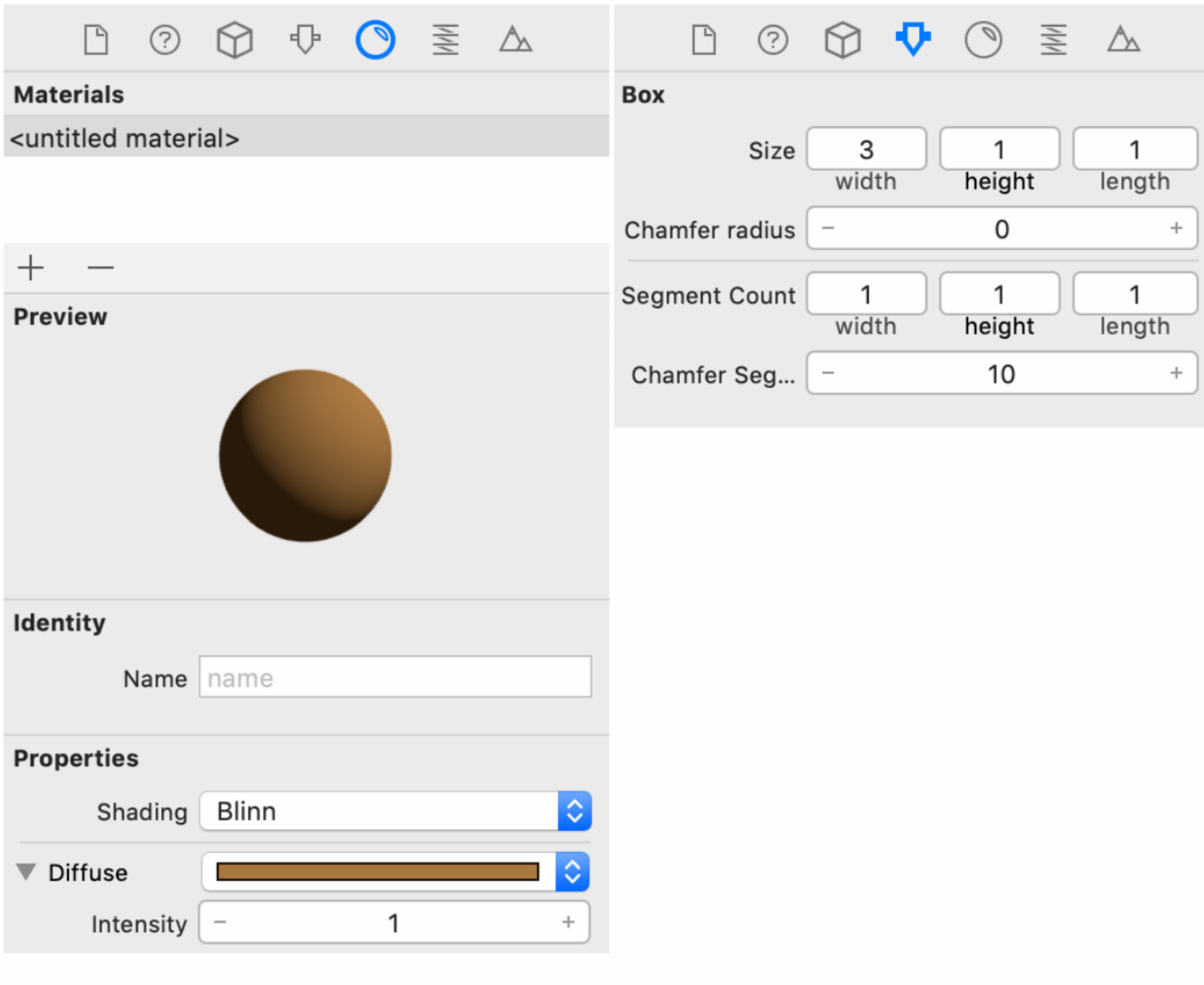
Box

A box, or 3D cube, is a basic rectangular shape that's created using the `SCNBox` class. It has a length, width, and height, as well as a radius that you can use to round the corners.

To represent a school building, locate a "Box" in the Object library and add it as a child node to the `Campus` node you created earlier.



Name the box "Main Building," then color it brown in the Materials inspector. Under the Attributes inspector, change the width to 3.0 meters.



If you were doing this in code, it might look like the following:

```
func loadCampus() {
    // Create a new scene
    let scene = SCNScene(named: "art.scnassets/campus.scn")!

    // Set the scene to the view
    sceneView.scene = scene

    // Load main building
    loadMainBuilding()
}

func loadMainBuilding() {
    let node = SCNNode()
```

```
    let geometry = SCNBox(width: 3.0, height: 1.0, length: 1.0,
        chamferRadius: 0.0)
    geometry.firstMaterial?.diffuse.contents = UIColor.brown
    node.geometry = geometry

    sceneView.scene.rootNode.addChildNode(node)
}
```

Do you see the main building when you build and run the app? It's probably the case that *all* you see is the brown box, because it's so large and is positioned directly on top of the origin. To make the box smaller, you could adjust its length, its width, and its height. But a simpler approach would be to use the `scale` property in the Node inspector to adjust the overall size percentage.

Go ahead and apply a scale of 0.5 to the axes and set the position's z-value to -1. Your AR scene should now have a building of reasonable size and position. Play with these numbers until you find values that work well. Experiment with sizes and colors. Whether you're adjusting the numbers in the SceneKit editor or in code, you can easily change them later—so don't worry about breaking anything in your scene. As you add more shapes into the scene, make sure they're child nodes of a node that's already scaled. That way, you won't have to scale them individually each time you make an adjustment to the size of your campus scene.

For reference, here's how to accomplish the same adjustment to `loadMainBuilding()` using code.

```
func loadMainBuilding() {
    let node = SCNNode()

    let geometry = SCNBox(width: 3.0, height: 1.0, length: 1.0,
        chamferRadius: 0.0)
    geometry.firstMaterial?.diffuse.contents = UIColor.brown
    node.geometry = geometry
    node.scale = SCNVector3(0.5, 0.5, 0.5)
```

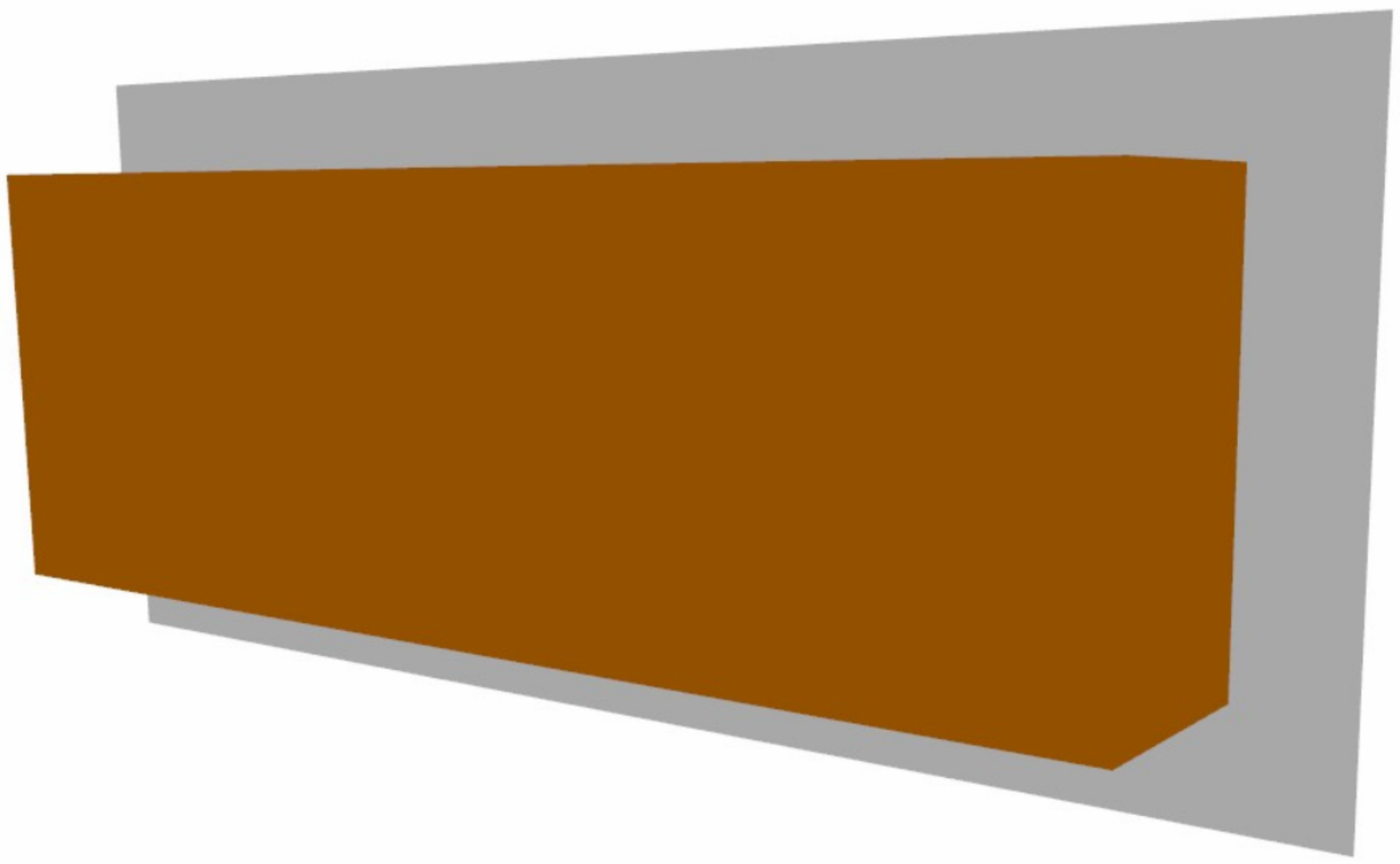
```
        let position = SCNVector3(0.0, 0.0, -1.0)
        node.position = position

        sceneView.scene.rootNode.addChildNode(node)
    }
```

Plane

A plane is a flat surface, represented by the `SCNPlane` class. Since a plane has only two dimensions, it's only viewable at an angle. Combined with colors, you can use planes to create sidewalks and grassy lawns.

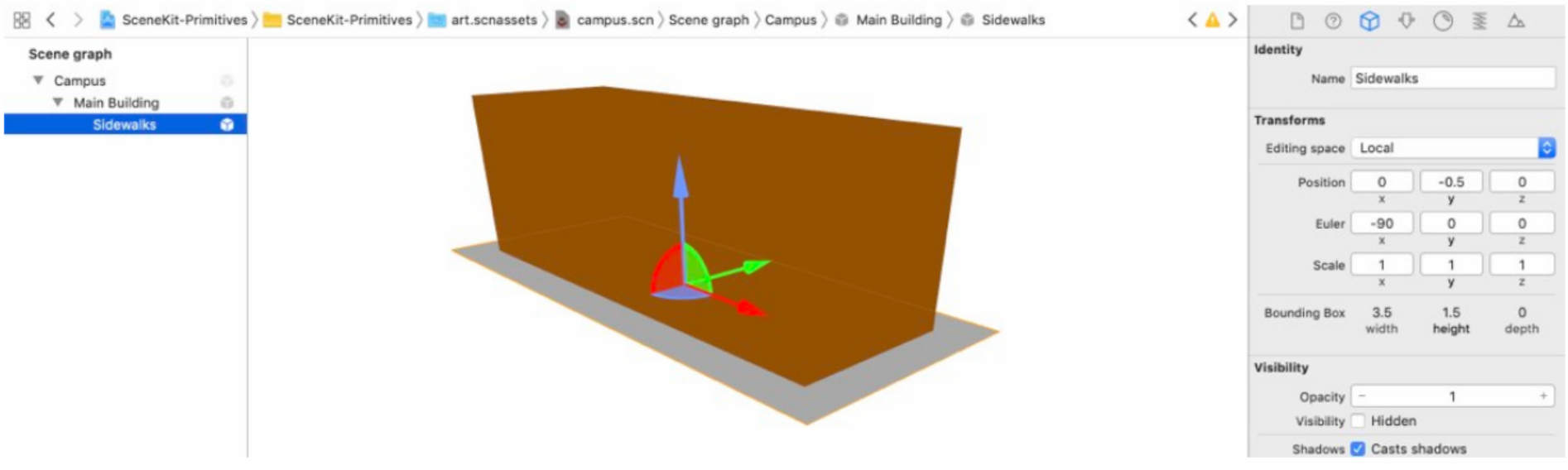
To begin, locate a "Plane" from the Object library, add it as a child node to the `Main Building` node, and name it "Sidewalks." Give it a width of 3.5 and a height of 1.5—or other values that are a little larger than the box you've created. That way, the sidewalk will extend beyond the building.

But you'll quickly notice a problem: The plane is standing vertically, behind your building, rather than lying flat.

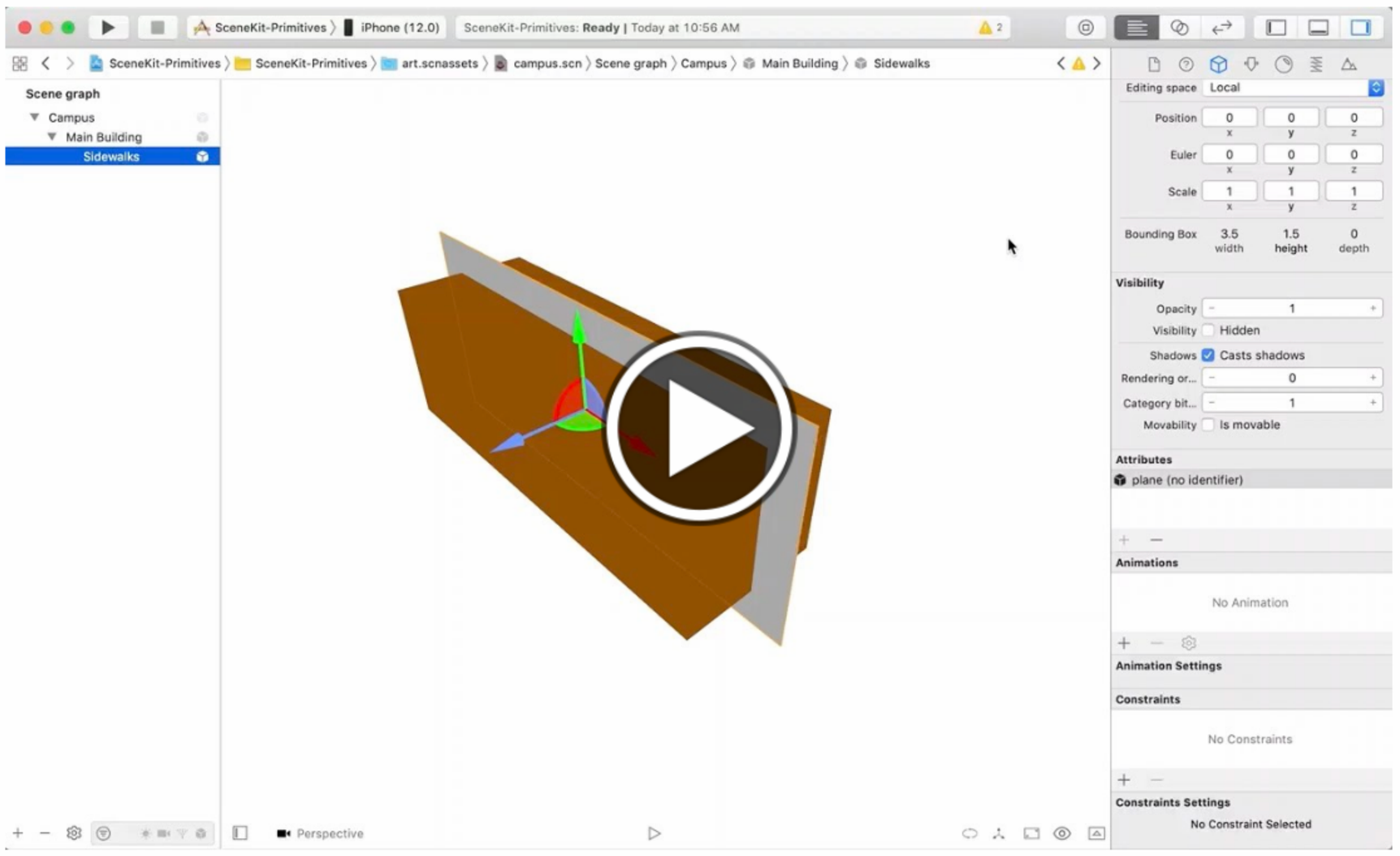


By default, a plane's `width` property extends in the x-axis and the `height` property extends in the y-axis. But in order to create a horizontal sidewalk, the plane should be increasing along the x- and *z*-axes. The simplest solution is to rotate the plane 90 degrees along the x-axis, which can be done by modifying the `euler` property in the Node inspector. (The correct axis for rotating may not be immediately intuitive. Understanding rotations and transformations will come with practice.)

Give your sidewalk a gray color, and size it so that there's an equal width of sidewalk on every side of the building.



If you were creating your scene in code, it might look like the following. (Note the `isDoubleSided` property is set to `true`, ensuring that the plane is visible whether you're viewing it from above or below. When you use the SceneKit editor to introduce a plane into your scene, the property has already been set to `true` for you.)

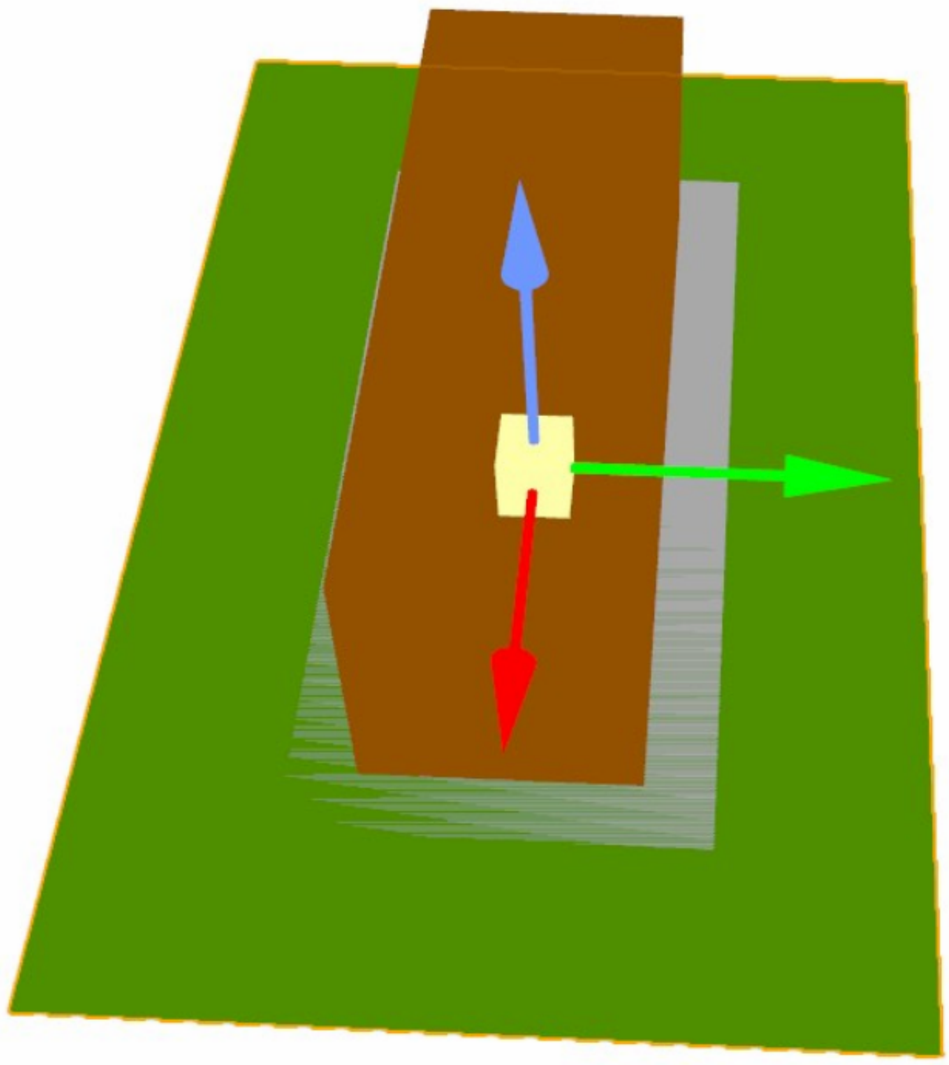


```
func loadSidewalks() {
    let node = SCNNode()

    let geometry = SCNPlane(width: 3.5, height: 1.5)
    geometry.firstMaterial?.diffuse.contents = UIColor.gray
    geometry.firstMaterial?.isDoubleSided = true
    node.geometry = geometry
    node.eulerAngles = SCNVector3(-Float.pi / 2, 0.0, 0.0)

    let position = SCNVector3(0.0, -0.5, 0.0)
    node.position = position

    sceneView.scene.rootNode.addChildNode(node)
}
```

An alternative to setting the `eulerAngles` property is to modify the x-value. Since 360 degrees is equal to 2 * π, 90 degrees is equal to π / 2.

```
node.eulerAngles.x = -Float.pi / 2
```

Add a second, larger plane under the main building. Give it a green color to represent a lawn. Adjust the values until you have as much grass as you'd like, but watch out: If the sidewalk and grass planes have the same y-value, they may overlap slightly, resulting in a graphical bug. This issue is known as depth fighting, where SceneKit is unable to determine how to render objects that have the same depth in a scene.



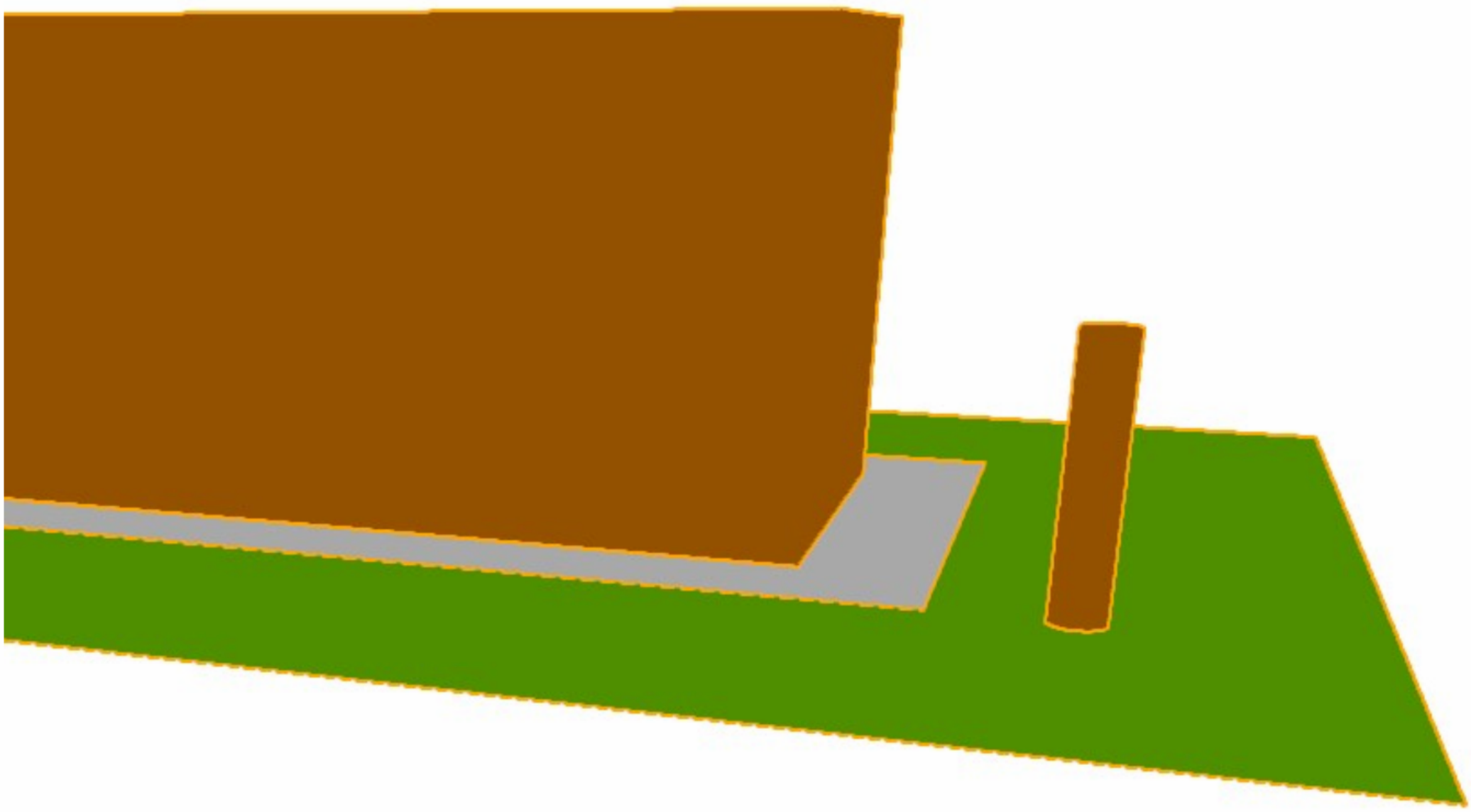
SceneKit offers multiple ways to solve depth fights. The simplest solution is to set the y-value of the lawn to a value that's a tiny bit lower than the sidewalk's. In this following image, the sidewalk plane has a y-value of -0.5, and the lawn plane's is -0.501.

```
func loadGrass() {
    let node = SCNNode()

    let geometry = SCNPlane(width: 4.5, height: 2.0)
    geometry.firstMaterial?.diffuse.contents = UIColor.green
    geometry.firstMaterial?.isDoubleSided = true
    node.eulerAngles.x = -Float.pi / 2
    node.geometry = geometry

    let position = SCNVector3(0.0, -0.501, 0.0)
    node.position = position

    sceneView.scene.rootNode.addChildNode(node)
}
```

Cylinder

A cylinder consists of two parallel circles bound by some amount of material. The `SCNCylinder` initializer requires a radius for both circles and a height for the space between them.

In your campus scene, you can use a cylinder to create a tree trunk. As with the other shapes, find a "Cylinder" in the Object library and set it to a brown color.

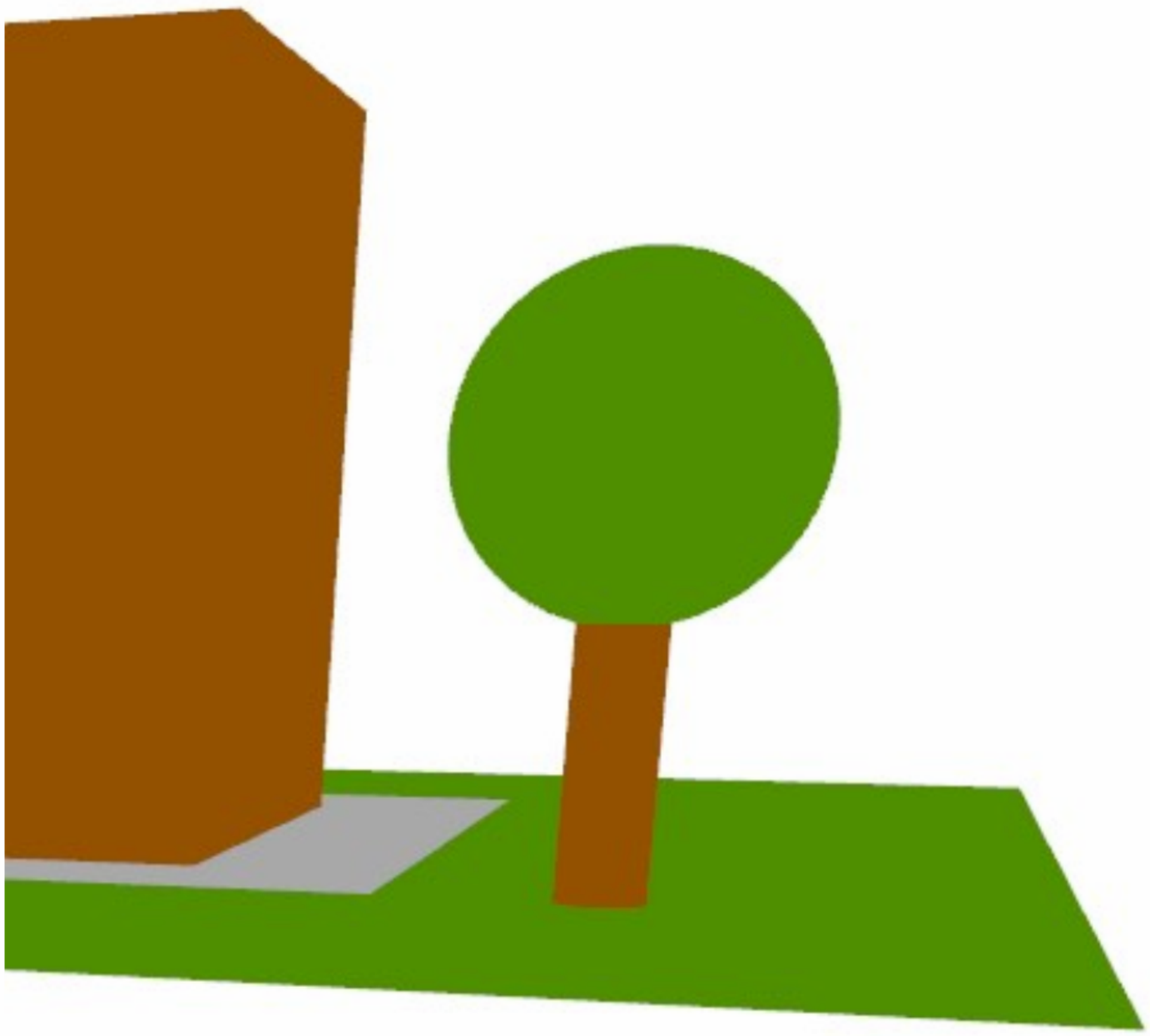


```
func loadTree() {
    let trunkNode = SCNNode()

    let trunkGeometry = SCNCylinder(radius: 0.05, height: 0.5)
    trunkGeometry.firstMaterial?.diffuse.contents
        = UIColor.brown
    trunkNode.geometry = trunkGeometry

    let trunkPosition = SCNVector3(2.0, -0.25, 0.75)
    trunkNode.position = trunkPosition

    sceneView.scene.rootNode.addChildNode(trunkNode)
}
```

Sphere

The last object you'll need is a sphere. Represented by the `SCNSphere` class, a sphere only needs a radius.

You can use a sphere to create the leafy crown for your tree trunk cylinder. But rather than adding this node as a child of the root node (as you've done with other shapes), you'll want to add it as a child of the *trunk* node—since the tree's crown won't exist without its trunk.



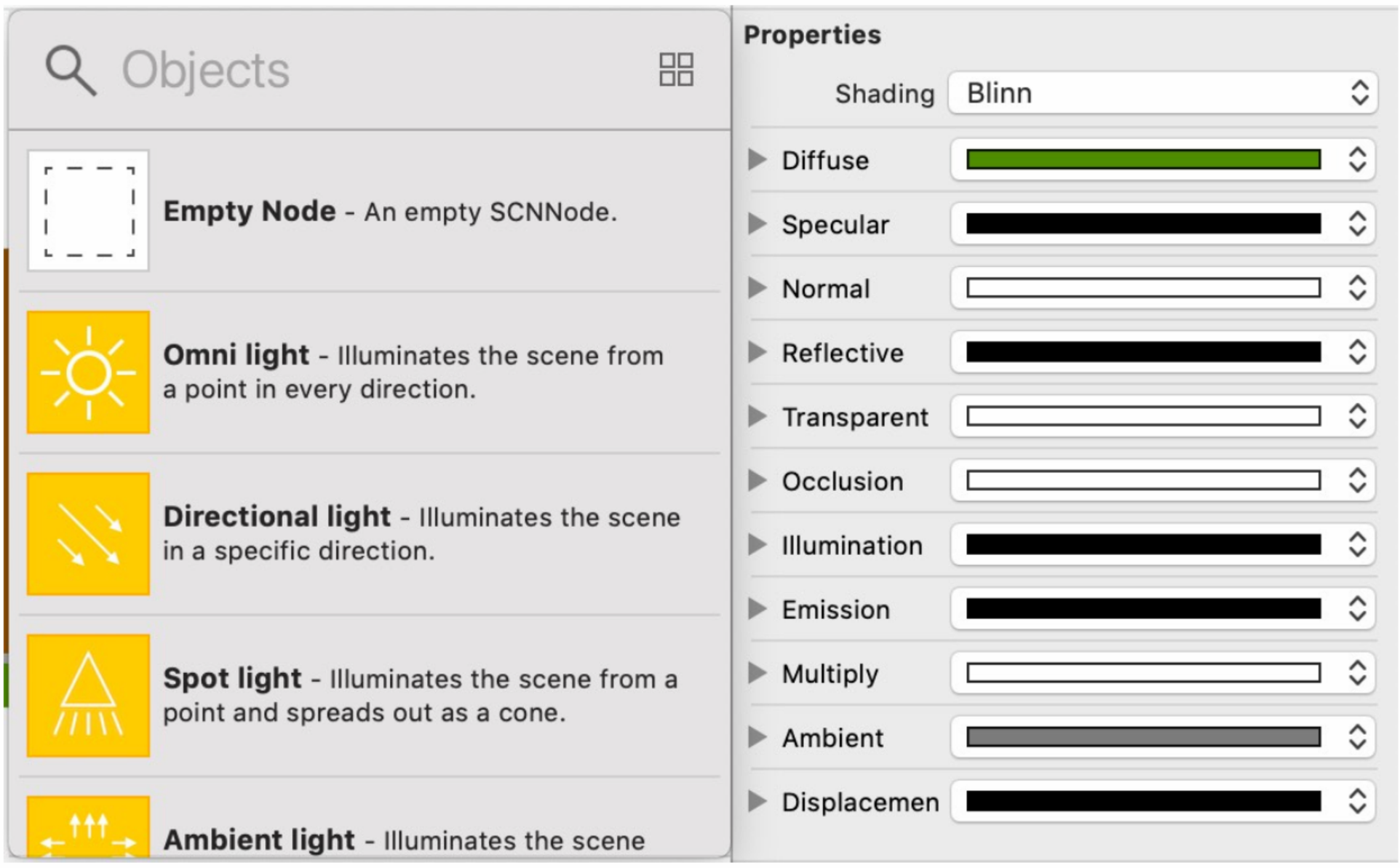
```
func loadTree() {
    let trunkNode = SCNNode()

    let trunkGeometry = SCNCylinder(radius: 0.05, height: 0.5)
    trunkGeometry.firstMaterial?.diffuse.contents
       = UIColor.brown
    trunkNode.geometry = trunkGeometry

    let trunkPosition = SCNVector3(2.0, -0.25, 0.75)
    trunkNode.position = trunkPosition

    sceneView.scene.rootNode.addChildNode(trunkNode)

    let crownNode = SCNNode()
```

```
        let crownGeometry = SCNSphere(radius: 0.2)
        crownGeometry.firstMaterial?.diffuse.contents
            = UIColor.green
        crownNode.geometry = crownGeometry

        let crownPosition = SCNVector3(0.0, 0.25, 0.0)
        crownNode.position = crownPosition

        trunkNode.addChildNode(crownNode)
    }
```

Lighting

Do the shapes you've placed in the campus scene seem a little ... dull? They may appear flat at certain angles, and you may have to move around to perceive them in 3D. This is especially obvious with a sphere: Since it has no edges, it will appear to be a circle, regardless of your viewing angle.

As you'll see, lighting is key to making objects appear three-dimensional, and SceneKit provides myriad ways to introduce lighting into a scene. From the Object library, you can add different types of lights to your nodes. And using the Materials inspector, you can specify exactly how light should interact with your objects. In fact, the `diffuse` property you use to set an object's color or texture is one way you're specifying how the object responds to light.



The quickest way to drastically improve the look of your scene is to set the `autoenablesDefaultLighting` property of your scene view to `true`. This will place into the scene an omnidirectional light source that interacts with all your existing nodes.

```
func viewDidLoad() {
    ...
    sceneView.autoenablesDefaultLighting = true
}
```

EXERCISES

- Using the Xcode documentation, investigate the `SCNCone` class. Then create a new building, separate from the main building, that uses an `SCNCone` as its roof.
- Modify the `loadTree()` function to take in two additional parameters: the x/z position of the tree. Then, use loops to create rows of trees around the school building.
- Create a new scene that more closely resembles your own school campus.

Lesson 3A.3

Finding Flat Surfaces

So far in this unit, you've learned a bit about how SceneKit can help you construct three-dimensional scenes. You've used ARKit to display camera information and give you an origin, but you haven't tapped into its most valuable function: detecting planes.

In this lesson, you'll learn how to enable plane detection, respond to the discovery of new planes, and update existing ones. From there, you can take the SceneKit models you've created and place them on surfaces, rather than leave them floating in space.

What You'll Learn

- How to enable detection of flat surfaces
- How to respond to plane detection
- How to visualize planes that have been detected
- How to place SceneKit assets on discovered planes

Vocabulary

- anchor
- `extent`
- feature point
- plane detection

Related Resources

- Building Your First AR Experience

FEATURE POINTS

At its core, ARKit simplifies some very complicated tasks for iOS developers. You've already observed that ARKit automatically responds to the motion and orientation of your device. In each of your projects, objects in the scene appear to be fixed in space as you move around. But the framework is also taking in video frames from the camera and analyzing them for feature points. You can think of feature points as detectable objects—such as a coffee cup, the edge of a sofa, or a flat floor—discovered when a scene is analyzed.

To visualize ARKit's list of feature points, add `ARSCNDebugOptions.showFeaturePoints` to the collection of `debugOptions` attached to your scene view.

```
sceneView.debugOptions = [ARSCNDebugOptions.showFeaturePoints,
    ARSCNDebugOptions.showWorldOrigin]
```

Recommendations for Tracking Feature Points

Try adding the these options to a new AR project, then build and run the code on your device. Take note of which surfaces and/or environments seem to contain the most feature points. As you might expect, more feature points will enable more accurate placement and tracking of objects in augmented reality.

Here are some recommendations to improve tracking of feature points:

- **Surface texture.** If a surface is too plain, as with a solid brown tabletop or desktop, ARKit will have a hard time detecting the object as a feature. To improve feature detection, try positioning the camera over a textured surface, such as a carpeted floor or an upholstered chair.
- **Lighting.** Without enough light, ARKit can't process camera information effectively. Shiny surfaces also make it difficult for ARKit to detect a feature. Try running your AR projects in a well-lit room on nonreflective surfaces.
- **Movement.** To track movement, ARKit uses camera information and movement detected from the device's accelerometer and gyroscope. But sometimes, if you're moving too rapidly, ARKit may not be able to properly track your position, causing 3D objects to move away from their original location. As you move throughout a scene, try moving at a slow, steady pace and turning gradually to view your surroundings through the camera.

PLANE DETECTION

When enough feature points contain the same y-coordinate, ARKit can determine that a horizontal plane exists in the scene. But first, you'll need to tell the system that you're *looking* for planes.

As part of the scene view's session, there's a configuration called `ARWorldTrackingConfiguration`. To tweak this configuration to search for horizontal planes, you can enable the `planeDetection` property and set its value to include `.horizontal`.

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()

    // Enable horizontal plane detection
    configuration.planeDetection = [.horizontal]

    // Run the view's session
    sceneView.session.run(configuration)
}
```

How do you know when ARKit has discovered a plane in the scene? Recall the following line of code from the Augmented Reality App template:

```
sceneView.delegate = self
```

This code tells the `sceneView` object to send any information to `self` (i.e., the view controller). Option-click `delegate` in Xcode, then click the `ARSCNViewDelegate` link in the popup to view the documentation.

```
12          // Set the view's delegate
13          sceneView.delegate = self
14
```

Summary

An object you provide to mediate synchronization of the view's AR scene information with SceneKit content.

Declaration

```
weak var delegate: ARSCNViewDelegate? { get set }
```

Open in Developer Documentation

Xcode documentation lists five methods for handling content updates. The first method of interest is `renderer(_:, didAdd:, for:)`, which will be triggered whenever ARKit reports finding a new anchor.

Using Xcode's autocompletion and context-awareness features, begin typing "renderer" inside a class definition that conforms to `ARSCNViewDelegate`. Select the `renderer(_ renderer:, didAdd:, for:)` method from the list, then press the Return key to add the method header and empty body to your class.

ARPlaneDetection 〉 ARPlaneDetection 〉 ViewController.swift 〉 No Selection

```
1  //
2  //  ViewController.swift
3  //  ARPlaneDetection
4  //
5
6  import UIKit
7  import SceneKit
8  import ARKit
9
10 class ViewController: UIViewController, ARSCNViewDelegate {
11
12     @IBOutlet var sceneView: ARSCNView!
13
14     override func viewDidLoad() {
15         super.viewDidLoad()
16
17         // Set the view's delegate
18         sceneView.delegate = self
19
20         // Show statistics such as fps and timing information
21         sceneView.showsStatistics = true
22
23         // Show debug options
24         sceneView.debugOptions = [ARSCNDebugOptions.showFeaturePoints, ARSCNDebugOptions.showWorldOrigin]
25     }
26
27     override func viewWillAppear(_ animated: Bool) {
28         super.viewWillAppear(animated)
29
30         // Create a session configuration
31         let configuration = ARWorldTrackingConfiguration()
32
33         // Enable horizontal plane detection
34         configuration.planeDetection = [.horizontal]
35
36         // Run the view's session
37         sceneView.session.run(configuration)
38     }
39
40     override func viewWillDisappear(_ animated: Bool) {
41         super.viewWillDisappear(animated)
42
```

An anchor is both a position and an orientation for placing objects in an AR scene. With horizontal plane detection enabled, ARKit will automatically create new anchors for each discovered plane. The `renderer(_:, didAdd:, for:)` method will be called, providing you with an `SCNNode` that you can use for adding child nodes.

To test that your view controller is receiving messages about new anchor points, you can have it print a simple message whenever a new plane is detected. Since there are multiple types of anchors (`ARPlaneAnchor, ARImageAnchor`), you can use a `guard` statement to ensure that you only execute the `print` function when an `ARPlaneAnchor` is the discovered anchor type.

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node:
    SCNNode, for anchor: ARAnchor) {
     guard let planeAnchor = anchor as? ARPlaneAnchor else {
         return
     }

     print("A new plane has been discovered.")
}
```

Build and run your app. Then examine the floor, a desk, a table, or any other flat surface to check for print statements.

VISUALIZING PLANES

But wouldn't you rather be able to *see* the plane that was detected? You can add an `SCNPlane` object that covers the entirety of the discovered anchor. Since multiple planes may be detected as you move around the scene, it would be helpful to have a method that returns a new `SCNNode` with an `SCNPlane` as its geometry.

Here's the first take at a method (similar to the `loadSidewalks` method in Lesson 2) for making a semitransparent plane that allows you to see through to the floor:

```
func createFloor() -> SCNNode {
    let node = SCNNode()

    let geometry = SCNPlane(width: 1.0, height: 1.0)
    node.geometry = geometry

    node.eulerAngles.x = -Float.pi / 2
    node.opacity = 0.25

    return node
}
```

Inside the `renderer(_:, didAdd:, for:)` method, create a new floor for each plane anchor that's discovered, and attach it to the provided `SCNNode`:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node:
    SCNNode, for anchor: ARAnchor) {
    guard let planeAnchor = anchor as? ARPlaneAnchor else {
        return
    }

    let floor = createFloor()
    node.addChildNode(floor)
}
```

If you run your app with this new code, you'll discover two issues. First, the semitransparent plane doesn't accurately reflect the size of the plane in the scene; it's always a 1m-by-1m square. Second, you may see two planes overlap one another, when in reality they should merge into one larger plane.

Take a look at how to address these issues.

Adjusting Plane Size

Within an `ARPlaneAnchor` is an `extent` property that gives the estimated width and height of the detected plane. Passing the `ARPlaneAnchor` as a parameter to the `createFloor` method would allow you to use its `extent` values to size your `SCNPlane` appropriately. Since the y-value of any horizontal plane will be 0, you should use the x- and z-values for sizing.

```
func createFloor(planeAnchor: ARPlaneAnchor) -> SCNNode {
    let node = SCNNode()

    let geometry = SCNPlane(width:
       CGFloat(planeAnchor.extent.x), height:
       CGFloat(planeAnchor.extent.z))
    node.geometry = geometry

    node.eulerAngles.x = -.pi / 2
    node.opacity = 0.25

    return node
}

func renderer(_ renderer: SCNSceneRenderer, didAdd node:
   SCNNode, for anchor: ARAnchor) {
    guard let planeAnchor = anchor as? ARPlaneAnchor else {
        return
    }

    let floor = createFloor(planeAnchor: planeAnchor)
    node.addChildNode(floor)
}
```

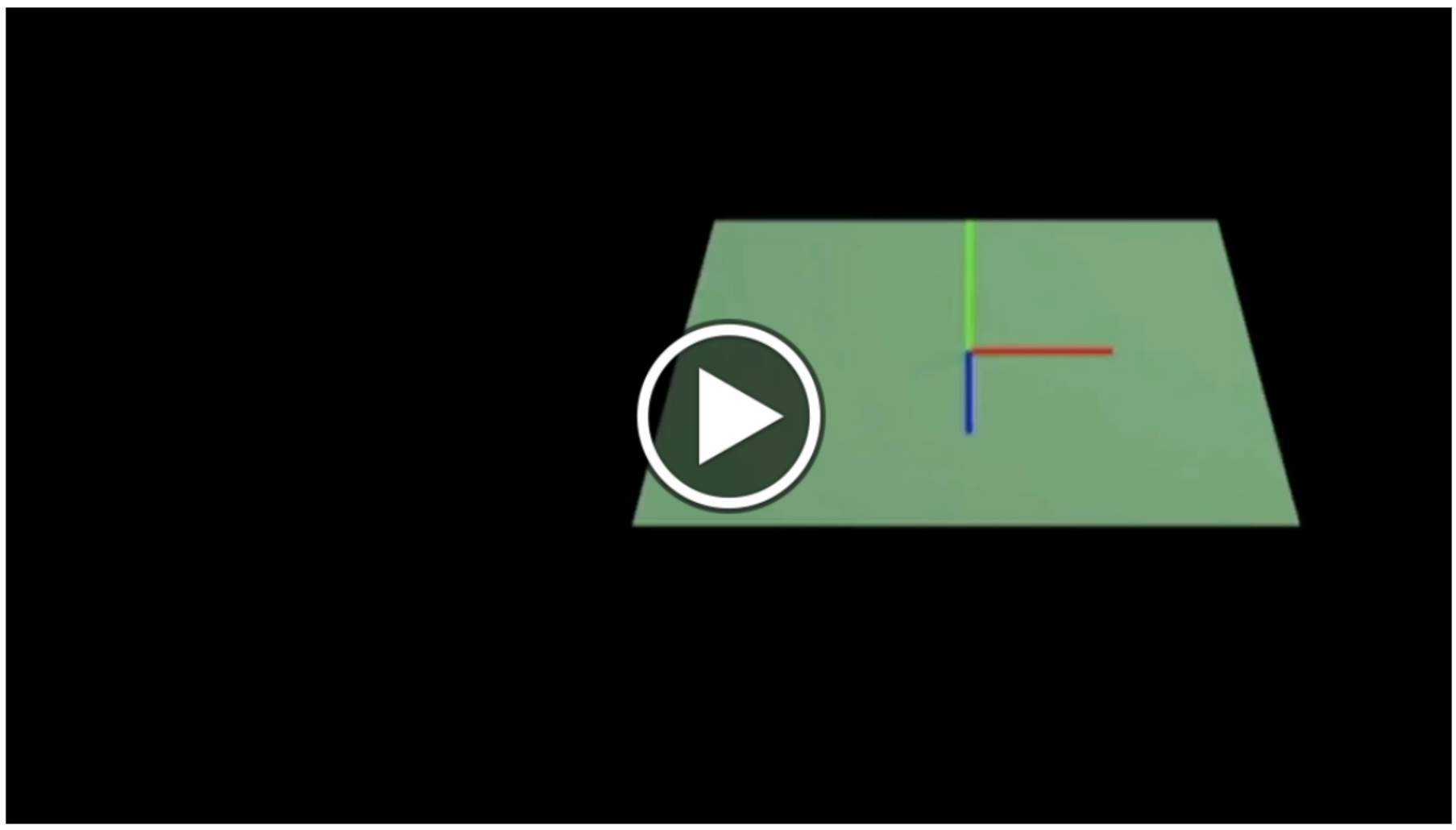
Now, when ARKit detects a horizontal plane, your semitransparent floor will match the size of the plane when it was originally detected.

Updating and Merging Planes

As ARKit continues to gather more information about a scene, it may discover that two `ARPlaneAnchor` objects are actually one large plane. With this discovery, it will automatically delete one of the anchors and enlarge the other to span the entire area. To be notified when a node is removed, use the `renderer(_:, didRemove:, for:)` method. To be notified when a node's properties have been updated to match the detected anchor, use `renderer(_:, didUpdate:, for:)`.

What about the instances of `SCNPlane` in your scene graph? Since ARKit is removing the parent node connected to your semitransparent planes, any child node will be removed automatically. However, if the plane anchor's size has grown, you'll need to update the size of your `SCNPlane`. You'll also need to update its position, because when the `extent` of an anchor changes, so does its `center`.

The following clip shows how a plane anchor's `center` property adjusts as its `extent` gets bigger.



Here's one way you could update your `SCNPlane`. You can use a `guard` statement (similar to the `guard` statement in the previous method) to ensure that the updated anchor is an `ARPlaneAnchor` and to verify that the first child node's geometry is an `SCNPlane`. When that has been established, you can match the node's position and size to the anchor's respective properties.

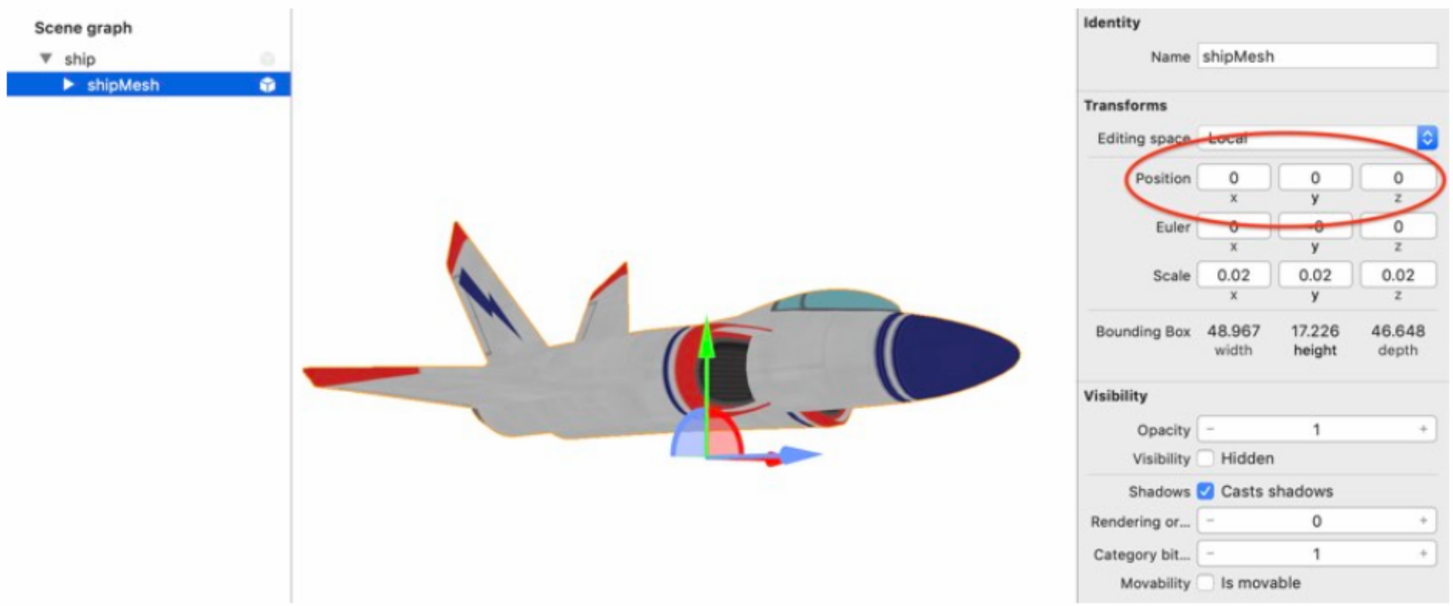
```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node:
    SCNNode, for anchor: ARAnchor) {
     guard let planeAnchor = anchor as? ARPlaneAnchor,
         let planeNode = node.childNodes.first,
         let plane = planeNode.geometry as? SCNPlane
         else { return }

     planeNode.position = SCNVector3(planeAnchor.center.x, 0,
        planeAnchor.center.z)
     plane.width = CGFloat(planeAnchor.extent.x)
     plane.height = CGFloat(planeAnchor.extent.z)
  }
```

As a result, you can see an existing plane grow as more information about the scene is discovered—rather than seeing overlapping smaller planes.

CONNECTING ASSETS TO PLANES

Now that you know how to create new nodes over the top of discovered anchors, you'll find it's really simple to connect the scenes you've created to physical surfaces. Imagine you want to place the ship from the Augmented Reality App template over every plane that's discovered, while keeping the ship centered above the plane, regardless of how the anchors change over time.

Open the `ship.scn` file in your project and change the position of the ship back to `(0,0,0)`. (By default, the ship's initial position is 0.1 meters above the origin and 0.8 meters behind it.)



In addition to adding the floor whenever a new plane is discovered, you'll want to create a node with a geometry that matches the ship. Since you might locate multiple planes, the `clone` method creates a copy of the ship. (If you don't make a copy, the single ship will jump to each newly discovered plane.) And just like the floor, you can use the anchor's `center` property to place the ship in the middle.

```
func createShip(planeAnchor: ARPlaneAnchor) -> SCNNode {

    let node = SCNScene(named: "art.scnassets/
       ship.scn")!.rootNode.clone()
    node.position = SCNVector3(planeAnchor.center.x, 0,
       planeAnchor.center.z)

    return node
}

func renderer(_ renderer: SCNSceneRenderer, didAdd node:
  SCNNode, for anchor: ARAnchor) {
    guard let planeAnchor = anchor as? ARPlaneAnchor
       else {return
    }

    let floor = createFloor(planeAnchor: planeAnchor)
    node.addChildNode(floor)

    let ship = createShip(planeAnchor: planeAnchor)
    node.addChildNode(ship)
}
```

As more data is gathered and as discovered planes get bigger, you can update the position of the ship and the semitransparent plane to match the plane anchor's `center`. As noted in the previous section, you'll also need to update the size of the plane (but the size of the ship can stay the same).

```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node:
    SCNNode, for anchor: ARAnchor) {
     guard let planeAnchor = anchor as? ARPlaneAnchor
         else { return }

     for node in node.childNodes {
         node.position = SCNVector3(planeAnchor.center.x, 0,
            planeAnchor.center.z)
         if let plane = node.geometry as? SCNPlane {
             plane.width = CGFloat(planeAnchor.extent.x)
             plane.height = CGFloat(planeAnchor.extent.z)
         }
     }
}
```

EXERCISES

- Import the school campus into your project, and place the scene on top of the first discovered plane.
- ARKit can also detect vertical planes. Create an app that covers the walls in red wallpaper. (Hint: `ARPlaneAnchor` has an `alignment` property to differentiate between horizontal and vertical planes.)
- Modify your app to enable both horizontal and vertical plane detection. Place red planes on all walls and green boxes on all horizontal surfaces, making sure that the objects you place are resized to cover the entire surface. (Hint: `ARPlaneAnchor` has an `alignment` property to differentiate between horizontal and vertical planes.)

Lesson 3A.4

Interacting with Augmented Reality

In the last lesson, you constructed scenes that take advantage of ARKit's ability to quickly and easily detect planes. Using those discovered surfaces, you automatically placed 3D assets into an augmented world.

But what if you want to place objects in a *particular* location, or interact with those objects once they've been placed? In this lesson, you'll discover how to recognize taps on the screen, approximate the tap's location in the real world, and create new objects that can interact with already existing assets. As part of learning these concepts, you'll construct a basketball hoop and allow the user to shoot hoops.

What You'll Learn

- How to translate a user's tap into a real-world position
- How to place 3D objects when discovered planes are selected
- How to add physics to an augmented reality scene

Vocabulary

- force
- hit test
- physics body
- physics shape

Related Resources

- Sample Code: Interactive Content with ARKit
- Handling 3D Interaction and UI Controls in Augmented Reality

PROJECT SETUP

Begin by creating a new project called "ARShots" using the ARKit template you've used in earlier lessons.

Creating a Basketball Hoop

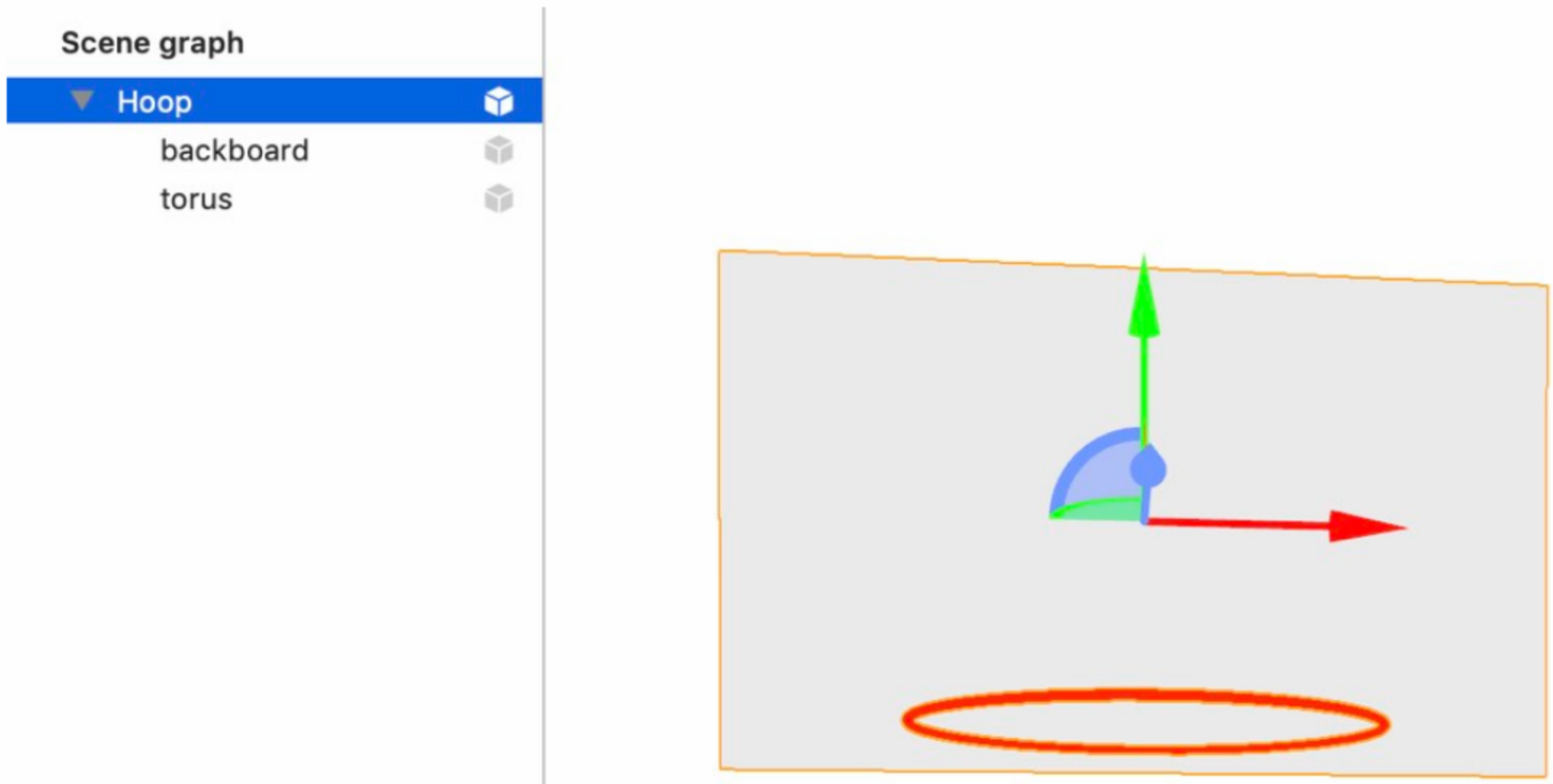
The two key components in a basketball hoop are the backboard and the rim of the basket. You can create these 3D objects using SceneKit primitives—as you did in Lesson 2.

Start by creating a new file in `art.scnassets` named `hoop.scn`. Open `hoop.scn` with the SceneKit editor, removing any cameras that may have been created, then add an "Empty Node" from the Object library at the top level of the scene. Using the Node inspector, give the empty node the name, "Hoop."

To create the backboard, add a "Box" as the `Hoop` node's first child. Resize the box inside the Attributes inspector. A typical backboard is about 1.8 meters wide, 1.1 meters tall, and 0.1 meters thick.

For the rim of the basket, you'll need a 3D shape called a torus. You can think of a torus as the product of two circles: a ring, and a pipe that encircles the ring. In SceneKit, it's represented by the `SCNTorus` class.

From the Object library, add a "Torus" as a second child of the `Hoop` node. Set its ring radius to 0.45 and its pipe radius to 0.1. Then adjust the torus' position so that the rim is in front of the backboard, but still touching it. When complete, your hoop should look like the following image.



Vertical Plane Detection

In the previous lesson, you dealt mostly with horizontal planes. As you'd expect, vertical plane detection works in the same way. To place the hoop against a wall, ARKit needs to know to search for vertical planes. Update the `ARWorldTrackingConfiguration` accordingly:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    let configuration = ARWorldTrackingConfiguration()
    configuration.planeDetection = .vertical

    sceneView.session.run(configuration)
}
```

In the last lesson, you used a few of the `ARSCNViewDelegate` methods to add visuals over the top of discovered planes. Typically, you'll want to provide some indication that planes have been discovered—go ahead and add those methods if you'd like. Otherwise, this lesson assumes that you've discovered a vertical plane in your scene, and instruction on these methods has been omitted.

PLACING THE BASKETBALL HOOP

In your app, the user will start by tapping the screen to place the hoop. But how do you know when a user has tapped the screen? Based on previous units of this course, you might think to use a `UIButton`. After all, you could place a large, clear button over the entire `SCNView` and you'd know whenever the user tried to interact with the screen. However, that's not the best approach for an AR project. Here's why.

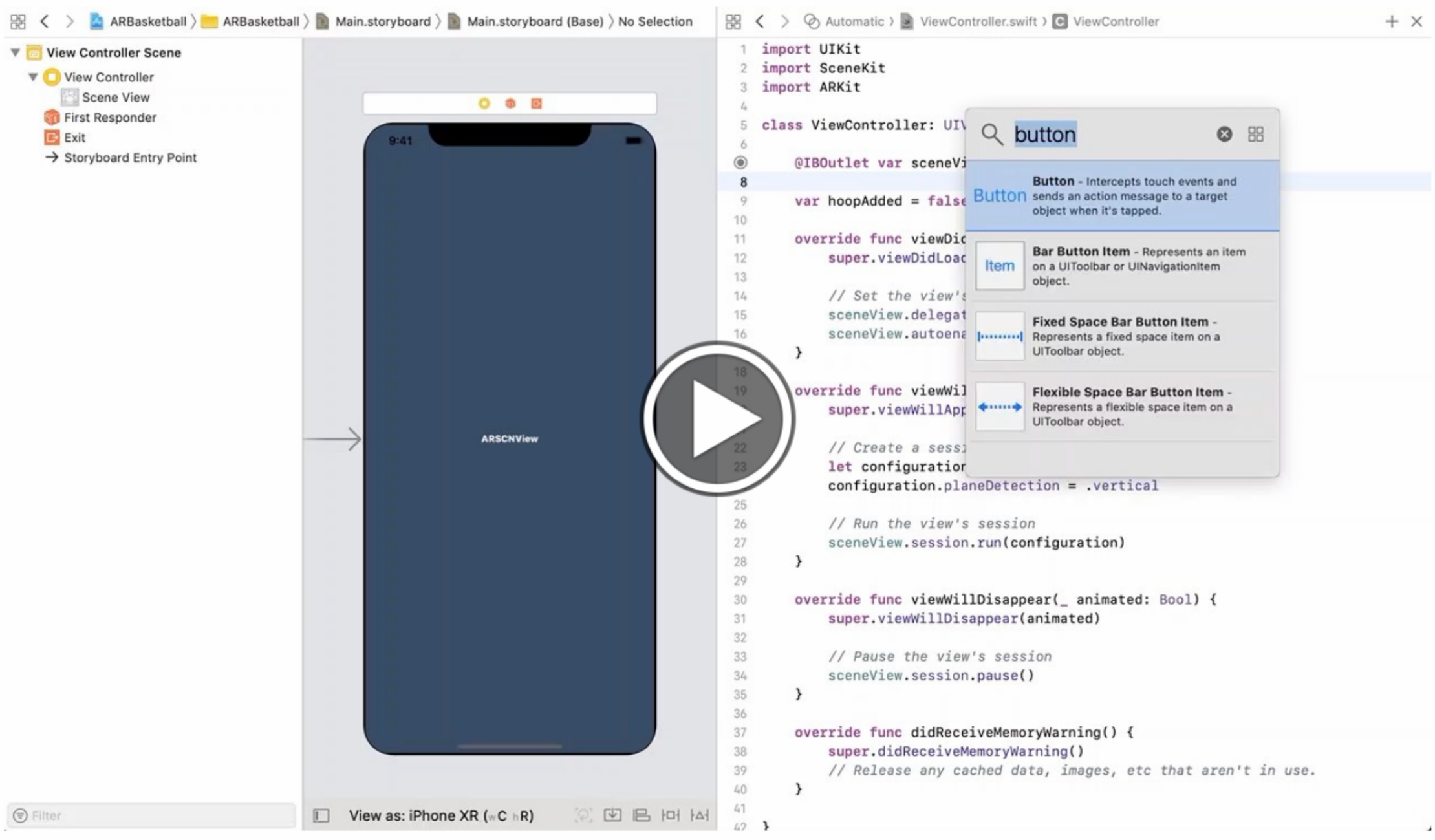
First, in Interface Builder, you can't easily add a `UIView` as a subview to an `SCNView`. If you want to place a `UIButton` over the top of the scene view, you'd need to add it as a subview of the view controller's view and then constrain it to fill the same space as the `SCNView`.

Secondly, while a `UIButton` can recognize when it's tapped, it doesn't really care *where* it's tapped. In an AR scene, the location of the user's tap is crucial for determining if and where the hoop will be placed. Instead of using a `UIButton` to recognize a screen tap,

you can use the `UITapGestureRecognizer` method. (Refer to Lesson 2.9 to review gesture recognizers.)

From the Object library, drag a "Tap Gesture Recognizer" on top of the scene view. Next, open the assistant editor, and Control-drag from the gesture recognizer to an available spot in the `ViewController` class definition. Change the Connection field from Outlet to Action, then give it the name, "screenTapped."

```
@IBAction func screenTapped(_ sender: UITapGestureRecognizer

}
```



Hit Testing

Instead of relying on automatic placement over the first discovered plane, your app will allow the user to place the hoop over the nearest vertical plane using a simple screen tap. To accomplish this, the app will need to perform a hit test.

Imagine sending out a beam (like a ray of light) from the exact position of the screen tap and at the same angle as the camera. If the ray were to collide with a previously discovered plane, the hit test would contain a result.

Here's an implementation of hit testing inside the `screenTapped(_:)` method:

```
@IBAction func screenTapped(_ sender: UITapGestureRecognizer
    let touchLocation = sender.location(in: sceneView)
    let hitTestResult = sceneView.hitTest(touchLocation, types:
       [.existingPlaneUsingExtent])

    if let result = hitTestResult.first {
        print("Ray intersected a discovered plane")
    }
}
```

The first constant, `touchLocation`, is the x/y position where the user tapped the screen. Using that location, the `hitTest(_:, types:)` will send out an imaginary ray in search of an existing plane, specified using the `.existingPlaneUsingExtent` option. The result of that method, `hitTestResult`, is an array containing all the results of the hit test—or an empty array if no planes were found. Even though there may be multiple results, you'll typically only need a single result to be discovered. You'll use that result to position and place objects into your scene.

Assuming a positive hit test result, you'll add the hoop at that location. Create a method that will add a hoop to the scene, using the hit test result for positioning. Following is an example of this method, including some code that locates the `Hoop` node in the `hoop.scn` file.

```
func addHoop(result: ARHitTestResult) {
    // Retrieve the scene file and locate the Hoop node
    let hoopScene = SCNScene(named: "art.scnassets/hoop.scn"

    guard let hoopNode = hoopScene?.rootNode.childNode(withName:
       "Hoop", recursively: false) else {
```

```
        return
    }

    // Place the node in the correct position (Not yet complete)

    // Add the node to the scene
        sceneView.scene.rootNode.addChildNode(hoopNode)
}
```

How do you determine *where* the tap and the vertical plane crossed paths? The Xcode documentation for `ARHitTestResult` reveals a property called `worldTransform`, which gives the position and orientation of the result in the real world's x/y/z space. The `worldTransform` property is a 4x4 matrix of values. The values in the first three columns of the matrix define the rotations of an object along the x-, y-, and z-axes. The values in the fourth column give the position of the result—and will be used to place the hoop against the vertical plane.

You can use the values in the `worldTransform` matrix to set the `position` property of the basket node, so that it matches the position of the user's tap instead of the default `(0,0,0)` position. The third column's values can be queried and used to initialize an `SCNVector3` that updates the `position` property.

```
func addHoop(result: ARHitTestResult) {
    // Retrieve the scene file and locate the "Hoop" node
    let hoopScene = SCNScene(named: "art.scnassets/hoop.scn"

    guard let hoopNode = hoopScene?.rootNode.childNode(withName:
      "Hoop", recursively: false) else {
        return
    }

    // Place the node in the correct position
    let planePosition = result.worldTransform.columns.3
    hoopNode.position = SCNVector3(planePosition.x,
```

```
        planePosition.y, planePosition.z)

    // Add the node to the scene
    sceneView.scene.rootNode.addChildNode(hoopNode)
}

@IBAction func screenTapped(_ sender: UITapGestureRecognizer
    let touchLocation = sender.location(in: sceneView)
    let hitTestResult = sceneView.hitTest(touchLocation, types:
       [.existingPlane])

    if let result = hitTestResult.first {
        addHoop(result: result)
    }
}
```

Build and run the app, then tap the screen over the top of a discovered vertical plane. The basketball hoop should find its correct position. If taps fail to place the hoop, you may want to add the plane detection visualization code from Lesson 3 to ensure that vertical planes are being found.

Creating and Positioning Basketballs

What about the basketball? You can add an `SCNSphere` to your augmented reality scene. Here's a method that creates an orange ball and adds it to the scene:

```
func createBasketball() {
    let ball = SCNNode(geometry: SCNSphere(radius: 0.25))
    ball.geometry?.firstMaterial?.diffuse.contents =
    UIColor.orange

    sceneView.scene.rootNode.addChildNode(ball)
}
```

To shoot hoops, you'll want the ball to be directly in front of the camera, but how do you position it? You can't use hit testing for placement because the ball's position isn't relative to any existing planes.

As ARKit analyzes the information it receives from the camera, it generates a video frame image, stored in the session's `currentFrame` property. A frame contains information about what the camera has captured, including the image, a timestamp, and the camera's change in position from the origin via the `camera.transform` property.

To position the basketball in the same place as the camera, just set its `transform` property equal to that of the camera. Because the camera's `transform` is of type `matrix_float4x4` and the ball's `transform` is of type `SCNMatrix4`, you'll need to do a quick conversion.

```
func createBasketball() {
    guard let currentFrame = sceneView.session.currentFrame else
        { return }

    let ball = SCNNode(geometry: SCNSphere(radius: 0.25))
    ball.geometry?.firstMaterial?.diffuse.contents =
        UIColor.orange

    let cameraTransform =
        SCNMatrix4(currentFrame.camera.transform)
    ball.transform = cameraTransform

    sceneView.scene.rootNode.addChildNode(ball)
}
```

When should you call the `createBasketball()` method? In this app, the user taps the screen to place the hoop, and each subsequent tap will add a basketball. You've already set up a gesture recognizer that responds to screen taps. With a simple Boolean, you can control whether a tap should attempt to place a hoop or to add a basketball.

```
var hoopAdded = false
```

```
@IBAction func screenTapped(_ sender: UITapGestureRecognizer
    if !hoopAdded {
        let touchLocation = sender.location(in: sceneView)
        let hitTestResult = sceneView.hitTest(touchLocation,
            types: [.existingPlane])

        if let result = hitTestResult.first {
            addHoop(result: result)
            hoopAdded = true
        }
    } else {
        createBasketball()
    }
}
```

Build and run your app to make sure that the first tap places the hoop. Subsequent taps should place a basketball in the scene directly at the camera's location. (Since the ball will be created at your exact location, you may need to step back to see it.)

INCORPORATING PHYSICS

How will the basketball interact with the hoop? It's time to add some physics to your scenes.

In addition to primitive shapes and nodes, the SceneKit framework includes a number of ways to change object positions over time, either by animating `position` and `transform` properties of a node, or by incorporating a physics simulation. With each new frame rendered in SceneKit, physics calculations are performed to determine how to update the position of nodes. These calculations can include gravity, friction, and collisions with other nodes.

Physics Body

To add a node to SceneKit's physics simulation, a physics body must be associated with the node. Every `SCNNode` can have a `physicsBody` property of type `SCNPhysicsBody`.

This property describes how the node will interact with other nodes in the physics simulation.

The initializer for `SCNPhysicsBody` requires two parameters: a type and a shape. The type can be set to one of three values:

- `static` — The node is unaffected by forces or collisions, but still participates in the physics simulation.
- `dynamic` — The node can be affected by forces or collisions.
- `kinematic` — The node can be moved, but is unaffected by forces or collisions.

Since the basketball will be affected by its interaction with the backboard and the rim, its type should be `dynamic`.

Physics Shape

The physics shape determines how an object *responds* to interactions with other bodies. Think about how different-shaped objects collide with a wall. A cone will bounce differently depending on whether its tip or its base hits the wall first. But a sphere will react the same regardless of its rotation in space.

One of the initializers for `SCNPhysicsBody` makes this very simple, allowing you to pass the node itself to determine the right shape:

```
let physicsBody = SCNPhysicsBody(type: .dynamic, shape:
   SCNPhysicsShape(node: ball))
ball.physicsBody = physicsBody
```

Try adding the above code to the `createBasketball()` method, then run the app. What do you see? Point your iOS device towards the ground and you should see the basketball falling into the abyss. Since `physicsBody` has a value, it's affected by SceneKit's simulation of gravity. And since no other objects exist in the scene, it just keeps falling forever.

It's no surprise that gravity pulls the ball downward. But how can you shoot hoops with it? In order for the basketball to move toward the hoop (in the same direction as the camera), there needs to be some sort of push, or force, behind it.

You already have the camera details stored in the current frame's `camera.transform` property. The `transform` property's third column of values indicates the orientation of the camera: The first row `(m31)` represents the x-value, the second row `(m32)` represents the y-value, and the third row `(m33)` represents the z-value. When these values are multiplied by some amount of "power," the ball will be launched in the proper direction. Use the `applyForce(_:, asImpulse:)` method to give the ball the desired force. By setting `asImpulse` to `true`, the force is applied instantaneously.

```
let physicsBody = SCNPhysicsBody(type: .dynamic, shape:
   SCNPhysicsShape(node: ball))
ball.physicsBody = physicsBody

let power = Float(10.0)
let force = SCNVector3(-cameraTransform.m31*power, -
   cameraTransform.m32*power, -cameraTransform.m33*power)

ball.physicsBody?.applyForce(force, asImpulse: true)
```

The ball launches forward in the intended direction, but it goes right through the backboard. Do you understand why? Take a moment to consider what you know about SceneKit's physics and the rules that have been laid out so far. Which objects participate in SceneKit's physics simulation, and why?

After some thought, you may realize that you haven't yet given the hoop a `physicsBody` —which means the backboard and the rim can't participate in the physics simulation and interact with the basketballs. Since the hoop doesn't move, its physics body's `type` should be set to `static`, and its `shape` should be equal to the node's shape.

```
hoopNode.physicsBody = SCNPhysicsBody(type: .static, shape:
   SCNPhysicsShape(node: hoopNode))
```

However, there's another problem. When you combine the backboard and the rim, you get an irregularly shaped body. To achieve a high level of physics detail about the hoop's shape, you need to add some options when you initialize the `SCNPhysicsShape` of the hoop and the ball.

Beginning with the hoop, you can set a value of `SCNPhysicsShape.ShapeType.concavePolyhedron` to the key, `SCNPhysicsShape.Option.type`. By doing so, you're telling the physics body to closely follow the surface of the geometry. This means the ball will react differently with the backboard than it will with the rim—just like in real life.

```
func addHoop(result: ARHitTestResult) {
    // Retrieve the scene file and locate the "Hoop" node
    let hoopScene = SCNScene(named: "art.scnassets/hoop.scn"

    guard let hoopNode = hoopScene?.rootNode.childNode(withName:
       "Hoop", recursively: false) else {
        return
    }

    // Place the node in the correct position
    let planePosition = hitTestResult.worldTransform.columns.3
    hoopNode.position = SCNVector3(planePosition.x,
       planePosition.y, planePosition.z)

    //Apply the correct physics body
    hoopNode.physicsBody = SCNPhysicsBody(type: .static, shape:
       SCNPhysicsShape(node: hoopNode, options:
       [SCNPhysicsShape.Option.type :
       SCNPhysicsShape.ShapeType.concavePolyhedron]))

    // Add the node to the scene
    sceneView.scene.rootNode.addChildNode(basketNode)
}
```

For the ball, there's an option called `collisionMargin`. As the name implies, it defines the distance between itself and other physics bodies before the object will react to a collision. Setting this to a very small number such as `0.01` will ensure that the ball must be within a centimeter of the hoop or rim before it bounces back.

```
let physicsBody = SCNPhysicsBody(type: .dynamic, shape:
    SCNPhysicsShape(node: ball, options:
    [SCNPhysicsShape.Option.collisionMargin: 0.01]))
```

With everything in place, you should have a functioning app that shoots basketballs toward a hoop, bouncing and interacting accurately with the backboard and rim.

EXERCISE

- Build an app that creates a golf green along the floor. After the green has been placed, subsequent taps should position a golf ball on top of the green, aiming the ball in the same direction as the camera.

Lesson 3A.5

Image Recognition in ARKit

The ARKit framework goes beyond discovering horizontal and vertical planes. The latest version can now find any images that are specified in your app. After instructing the framework to search for a particular set of images, your app can respond to their discovery using the same approach you learned with plane detection.

What You'll Learn

- How to enable image detection
- How to specify which images to locate in the physical world
- How to respond to the discovery of an image
- How to perform simple animations with SceneKit objects

Vocabulary

- AR Resource Group
- `ARImageAnchor`
- `ARReferenceImage`
- `SCNAction`

Related Resources

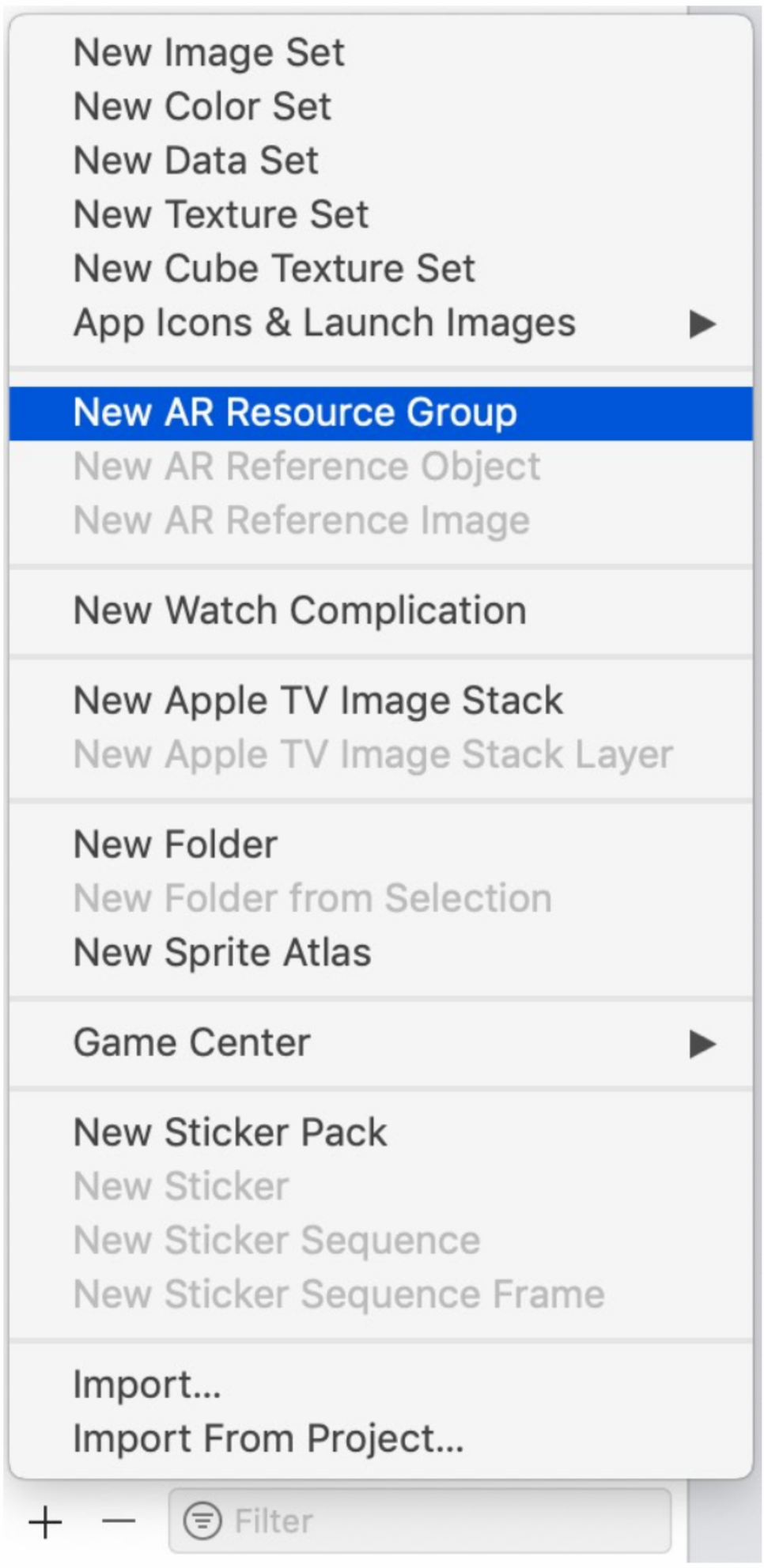
- Recognizing Images in an AR Experience

PROJECT SETUP

Use the Augmented Reality App template to create a new project, naming it "ARImageFinder." Remove the portions of code that add the ship to the scene, as well as the `ship.scn` file.

Adding Images

Which images do you want ARKit to detect? You'll need to add all of them to the asset catalog of your Xcode project. Open the `Assets.xcassets` folder, then click the plus (+) button at the bottom of the asset list to bring up a menu of options. Select New AR Resource Group, and a folder called `AR Resources` will be created.

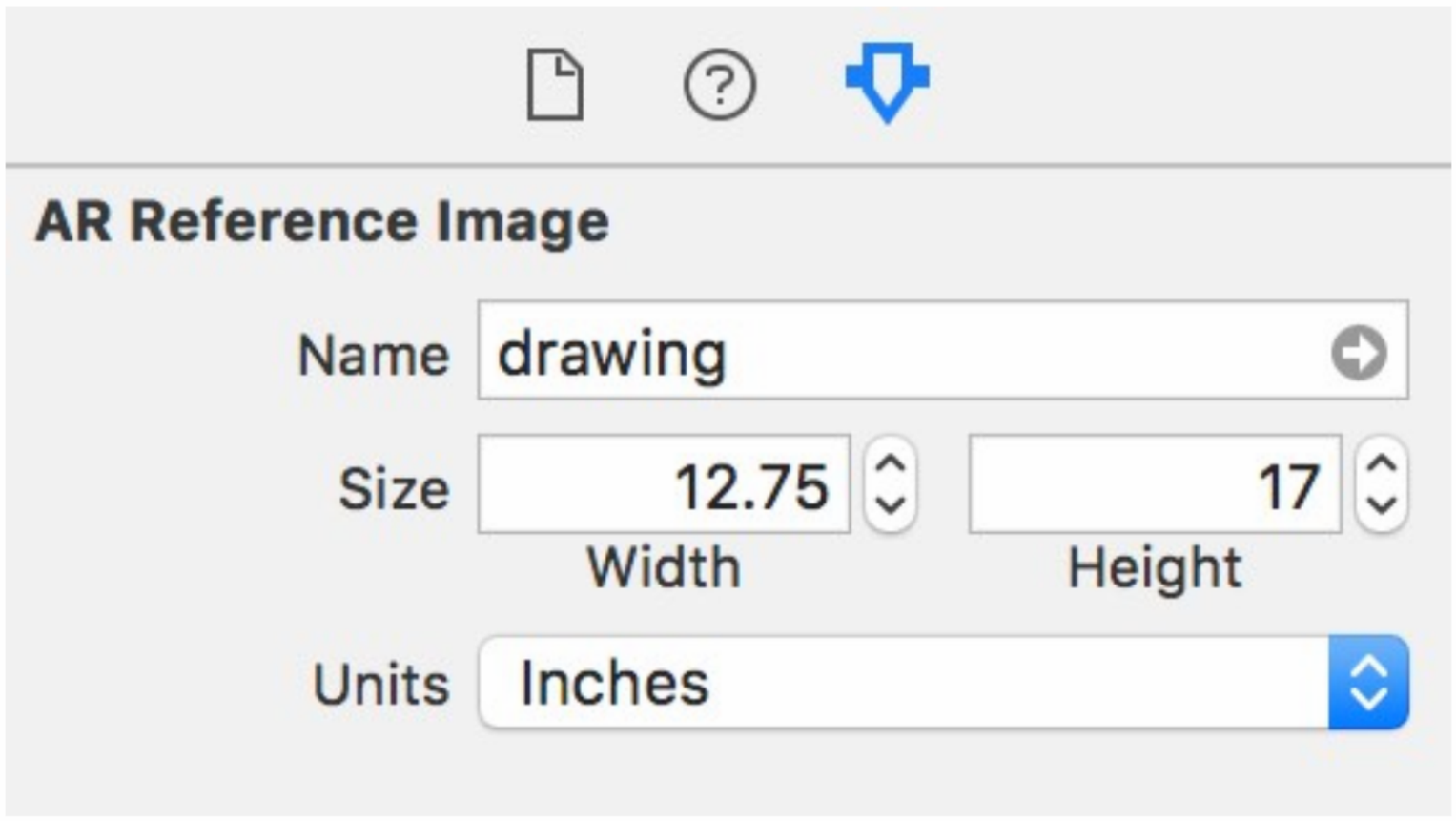


Choose some images that you wish to detect, such as photos of your friends, book covers, or famous paintings. Drag the images into the asset folder you just created. Keep the following things in mind when selecting your images:

- Images should not include any transparent data, which will return an error when you build and run the app.
- High-quality images will increase the chances of discovery in the physical world (even though they'll increase the size of the app).
- Images with higher contrast (larger differences between light and dark) will also make detection easier for ARKit.

Once supplied with your high-quality, high-contrast images, ARKit will need the real-world size of each image. This information enables ARKit to estimate the distance between the image and the camera, and to position 3D objects correctly as they're added into the scene.

Select an image in your `AR Resources` folder, then open the Attributes inspector. Input the width and height of the image, then repeat for other assets you've included. (Note that you should only need to input one value: When you change the width, the height will automatically adjust to guarantee the correct aspect ratio.) Select the correct unit of measurement (inches, meters, etc.) to match the width and height values you've supplied.



ENABLING IMAGE DETECTION

The `ARWorldTrackingConfiguration` has a property called `detectionImages`, a collection of the images you want your app to discover. Typically, when you locate images from the `Assets.xcassets` folder, you'll use the `UIImage(named:)` initializer to create a `UIImage`. But `detectionImages` requires a *collection* of `ARReferenceImage` objects, so you'll need to use another method.

What to do? As always, the best place to look for a solution is in the Xcode documentation. The `ARReferenceImage` docs include a section on Loading Reference Images, which lists the class method `referenceImages(inGroupNamed:, bundle:)`.

In the first argument, you specify the `String` name of the folder in `Assets.xcassets`. For this project, it's "AR Resources." (If your collection of images were located in another library or framework, you'd specify the location in the second argument. In this case, the parameter is `nil`.) The method returns an *optional* collection, but `detectionImages` requires a non-optional one. To force-unwrap the optional, you can add an exclamation point (`!`) to the end of the method. Note that your app will crash if the folder isn't found, so make sure that the `String` in the first parameter matches the folder name exactly.

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    let referenceImages =
       ARReferenceImage.referenceImages(inGroupNamed:
       "AR Resources", bundle: nil)!

    let configuration = ARWorldTrackingConfiguration()
    configuration.detectionImages = referenceImages
    sceneView.session.run(configuration)
}
```

RESPONDING TO IMAGE DETECTION

You've already learned the `renderer(_:, didAdd:, for:)` method to respond to detected planes. You'll use the same method when images are discovered. However, as you'd expect, the `anchor` will be of type `ARImageAnchor`, rather than `ARPlaneAnchor`. You can verify the anchor's type using type inspection.

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node:
    SCNNode, for anchor: ARAnchor
     guard let imageAnchor = anchor as? ARImageAnchor else {
         return
     }

     print("A new image has been discovered.")
}
```

Don't let this method become unwieldy. If your app needs to detect both planes and images, it would be smart to break up the method into smaller chunks. Simply pass the `node` and `anchor` to other methods once the anchor's type has been verified. (Remember that plane detection won't occur unless you enable it in the configuration via the `planeDetection` property.)

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node:
    SCNNode, for anchor: ARAnchor) {
     if let imageAnchor = anchor as? ARImageAnchor else {
         nodeAdded(node, for: imageAnchor)
     } else if let planeAnchor = anchor as? ARPlaneAnchor else {
         nodeAdded(node, for: planeAnchor)
     }
}
```

Rather than using a chain of `if-else` statements, you might think to use a `switch` statement—a more elegant approach that improves the readability of your code. The form of pattern matching you'll need is `case let`, which attempts to cast the value of concern to the indicated type. If the cast succeeds, the supplied variable is available in the scope o the clause below. (As an added benefit, using `switch` makes it easier to handle errors. The longer your list of types, the more you'll see the advantage of this approach.)

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node:
    SCNNode, for anchor: ARAnchor) {
     switch anchor {
     case let imageAnchor as ARImageAnchor:
         nodeAdded(node, for: imageAnchor)
     case let planeAnchor as ARPlaneAnchor:
         nodeAdded(node, for: planeAnchor)
     default:
         print("An anchor was discovered, but it is not for
             planes or images.")
     }
}

func nodeAdded(_ node: SCNNode, for imageAnchor: ARImageAnchor)
    {
     // Handle image detection
}

func nodeAdded(_ node: SCNNode, for planeAnchor: ARPlaneAnchor)
    {
     // Handle plane detection
}
```

Covering an Image

As with plane detection, image detection can give your app some interesting functionality. For example, you could load a 3D model that's positioned some distance from a detected image. Or you could present new information related to the image—such as the date it was created—inside a `UIViewController`.

Let's say you want to place an `SCNPlane` over the top of an image, similar to how you covered horizontal planes in Lesson 3. First, you need to retrieve the size of the image in the physical world (which you already supplied in the Attributes inspector when you moved images into the `Assets.xcassets` folder).

Every `ARImageAnchor` has a `referenceImage` property of type `ARReferenceImage`, the same type used when you assigned the `detectionImages` property. This object has a `physicalSize` that includes `width` and `height`, which you can use to size your `SCNPlane`. From there, construct an `SCNNode` using the plane, dialing down the `opacity` so that you can still see the image under the plane, then add the node to the scene.

```
func nodeAdded(_ node: SCNNode, for imageAnchor:
    ARImageAnchor) {
     let referenceImage = imageAnchor.referenceImage

     let plane = SCNPlane(width:
        referenceImage.physicalSize.width,
                          height:
                             referenceImage.physicalSize.height)
     plane.firstMaterial?.diffuse.contents = UIColor.blue
     let planeNode = SCNNode(geometry: plane)
     planeNode.opacity = 0.25

     node.addChildNode(planeNode)
  }
```

Build and run the app. Are the results what you expected? If not, check that your `renderer(_:, didAdd:, for:)` method is being called in the first place. If you're unable to detect an image, check for the following:

- Ensure that the Xcode console debugger isn't giving you warnings related to the images you've provided.
- Try printing your images on different kinds of paper. Notice how certain images look when printed on glossy paper and how the reflective quality of the paper interferes with image transfer. The same interference may occur when the image is displayed on the screen of a computer or device.

If your image was detected, you should see your plane positioned perpendicular to the image. Recall from Lesson 2 that an `SCNPlane` needs to be rotated 90 degrees along the x-axis to line up with the orientation in ARKit. This is an easy fix:

```
planeNode.eulerAngles.x = -Float.pi / 2
```

Rediscovering an Image

As you develop AR apps, you'll probably find situations where you want to display information about an image, dismiss it after some time has passed, then allow the image to be rediscovered when the user returns to the image's location in the scene. For example, imagine an AR app that displays information about the paintings in a museum. To mitigate cluttering the scene with too many floating shapes or 3D text, you could remove these SceneKit objects after a specified period of time.

Unlike an `ARPlaneAnchor`, which is maintained and tracked throughout the session's lifespan, an `ARImageAnchor` is tracked only for a short duration. After the image is dismissed and then rediscovered subsequently, the `renderer(_:, didAdd:, for:)` method will be called again, leading to duplicate nodes. To correct this issue, you'll need to do some cleanup of the nodes you've created.

To remove a view in UIKit, you call the `removeFromSuperview()` method on the view to be removed. With SceneKit nodes, there's a comparable method, `removeFromParentNode()`. Calling the method is simple, but implementing *when* it should be called is a bit trickier. How do you accomplish this?

Unlike its UIKit counterpart (which doesn't return a value), `removeFromParentNode()` returns an `SCNAction`, SceneKit's version of an animation. Using the `sequence(_:)` method, you can chain together multiple animations by placing them in a collection. To wait five seconds before removing the node, the call to `sequence(_:)` would look like the following:

```
SCNAction.sequence([.wait(duration: 5.0),
    .removeFromParentNode()])
```

Check out the `SCNAction` documentation for a full list of animation sequences. For example, use the following sequence to display your SceneKit objects for five seconds, then slowly fade them out over a two-second duration.

```
SCNAction.sequence([.wait(duration: 5.0),
    .fadeOut(duration: 2.0), .removeFromParentNode()])
```

To begin running an animation, call the `runAction(_:)` method on the node you want to animate. In the following snippet, the `SCNAction` has been placed into a computed property so that you can reuse the action in multiple methods, if needed.

```
func nodeAdded(_ node: SCNNode, for imageAnchor:
ARImageAnchor) {

    ...

    node.addChildNode(planeNode)

    planeNode.runAction(waitRemoveAction)
}

var waitRemoveAction: SCNAction {
    return .sequence([.wait(duration: 5.0), .fadeOut(duration:
        2.0), .removeFromParentNode()])
}
```

Differentiating Between Images

What if you need to know *which* image was discovered so that you can add image-specific content to the scene? Each `ARReferenceImage` includes a `name` property to help differentiate between the images you've added to your project. Inspect the property when an image anchor is created.

```
func nodeAdded(_ node: SCNNode, for imageAnchor:
ARImageAnchor) {
    let referenceImage = imageAnchor.referenceImage
    switch referenceImage.name {
    case "Jane-Selfie":
        // Create nodes relate to the image of Jane
    case "Todd-Selfie":
        // Create nodes related to the images of Todd
    default:
        // Create nodes related to other images
    }
}
```

EXERCISES

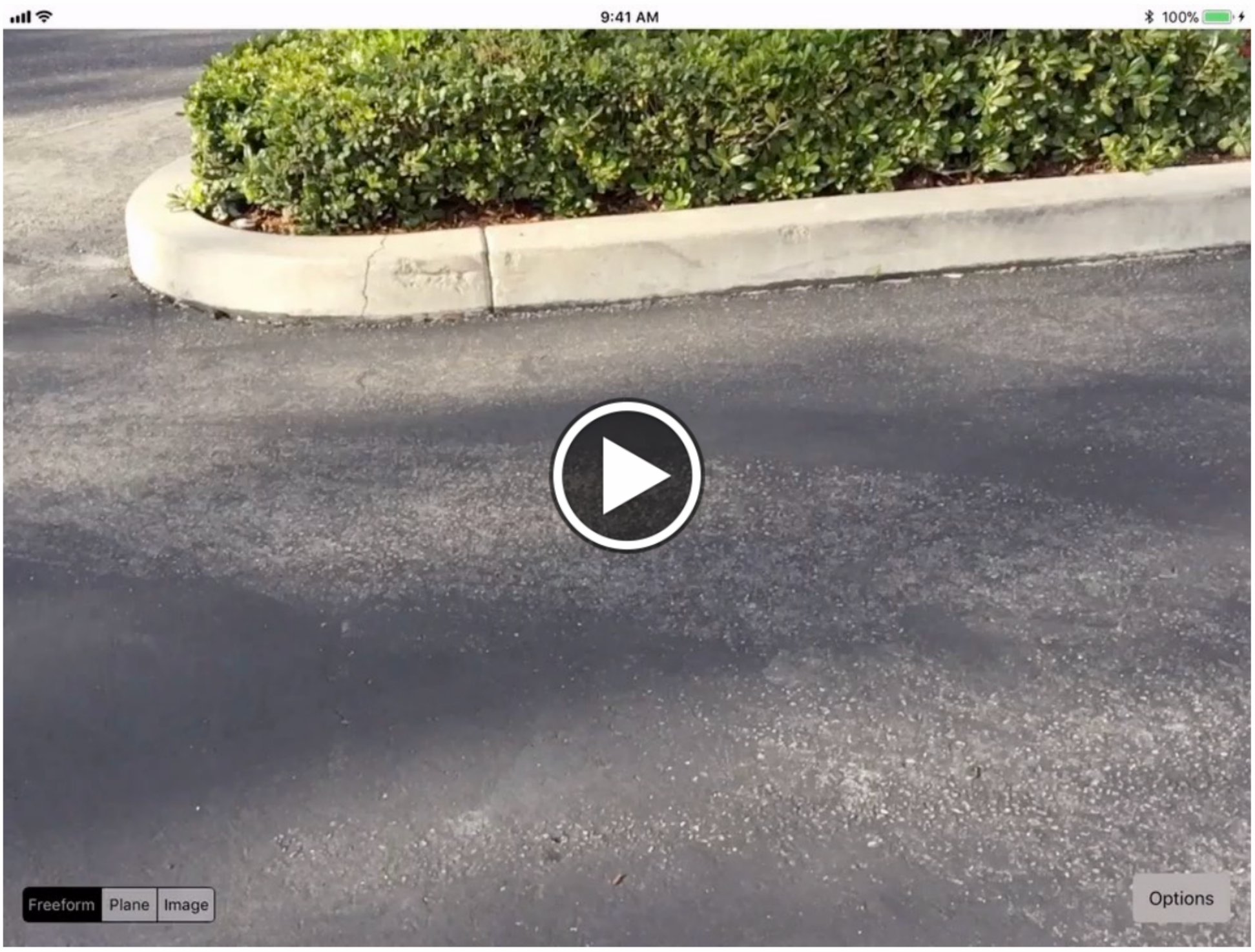
- Build a simple treasure hunt. Include three images in your project. The second image can't be discovered until the first has been detected. Likewise, the third image can't be discovered until the second has been detected.
- Experiment with more of the prebuilt `SCNAction` sequences. Modify your animation to fade in an object, wait for a duration (of your choice), then fade out.

Lesson 3A.6

Guided Project: AR Drawing

Throughout this unit, you've learned about a number of ARKit topics, including plane detection, positioning objects in 3D, and image recognition. Now it's time to combine that knowledge into a single app that takes advantage of many of those ARKit features.

In this guided project, you'll build your own 3D drawing app, called AR Drawing, that uses SceneKit objects to construct an AR scene. From a list of shapes and models, the user can select what they want to place in the scene. Then they'll use a set of controls to place objects wherever they choose: in front of the camera, on top of planes, or in relation to detected images.



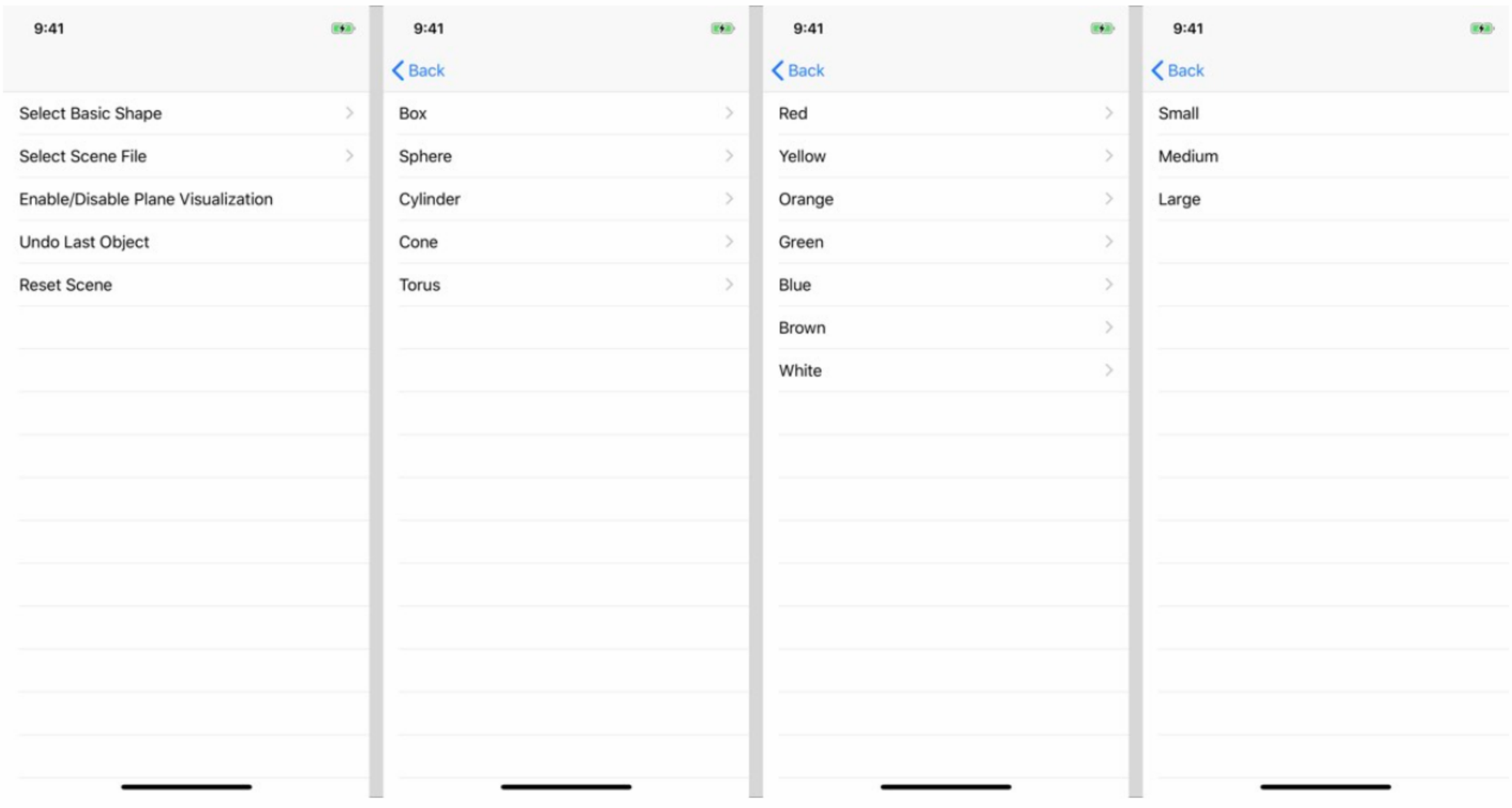
PART ONE—THE STARTER PROJECT

In your student resources folder, find the starter project called "AR Drawing." When you launch the project, you'll see a prebuilt interface, including a working popover, table views, and a segmented control. Before digging in any further, make sure that you can build and run the app without any errors. If you do encounter an issue, it's most likely that your physical iOS device isn't set as the target device. As with other AR apps, this project won't run on the iOS Simulator.

What's already in the AR Drawing app? Compare what you see on the screen to the files in the project.

Beginning with `Main.storyboard`, the segmented control has been properly constrained to the bottom left of the view, and the view controller's `changeObjectMode(_:)` method is set as its action. The Options button has been constrained to the bottom right of the view, and it performs a popover segue to the view controller in `Options.storyboard`. This is the view controller that manages the object selection process, and the `Options` group in the project navigator contains the code related to displaying the lists and the transitions between the lists.

`BasicShape.swift` contains a few enumerations that correspond to the user's options. Each enum has `String` values that represent the titles displayed in the popover—for example, "Box" and "Orange." For the Select Basic Shape option, the `ShapeOption` enum contains the five cases in the first list. Users can choose a shape (using the `Shape` enum), a color, and a size (using the `Size` enum). For the Select Scene File option, users can choose from a list of models you placed in `models.scnassets` in Lesson 5. The other three options do nothing at the moment. You'll come back to them as you work through the project.



`ViewController.swift` contains additional code that facilitates working with the user interface controls. You'll find an `objectMode` variable of type `ObjectPlacementMode` that represents the three selections in the segmented control. The previously mentioned `changeObjectMode(_:)` method updates `objectMode` to the proper value, which helps the app determine how objects should be placed in the scene.

At the bottom of the file is a list of methods that will be called when items in the Options menu are selected.

- `objectSelected(node:)` is called after the user selects a shape, color, and size, and after the user selects a model.
- `togglePlaneVisualization()` is called when the user taps Enable/Disable Plane Visualization.
- `undoLastObject()` is called when the user taps Undo Last Object.
- `resetScene()` is called when the user taps Reset Scene.

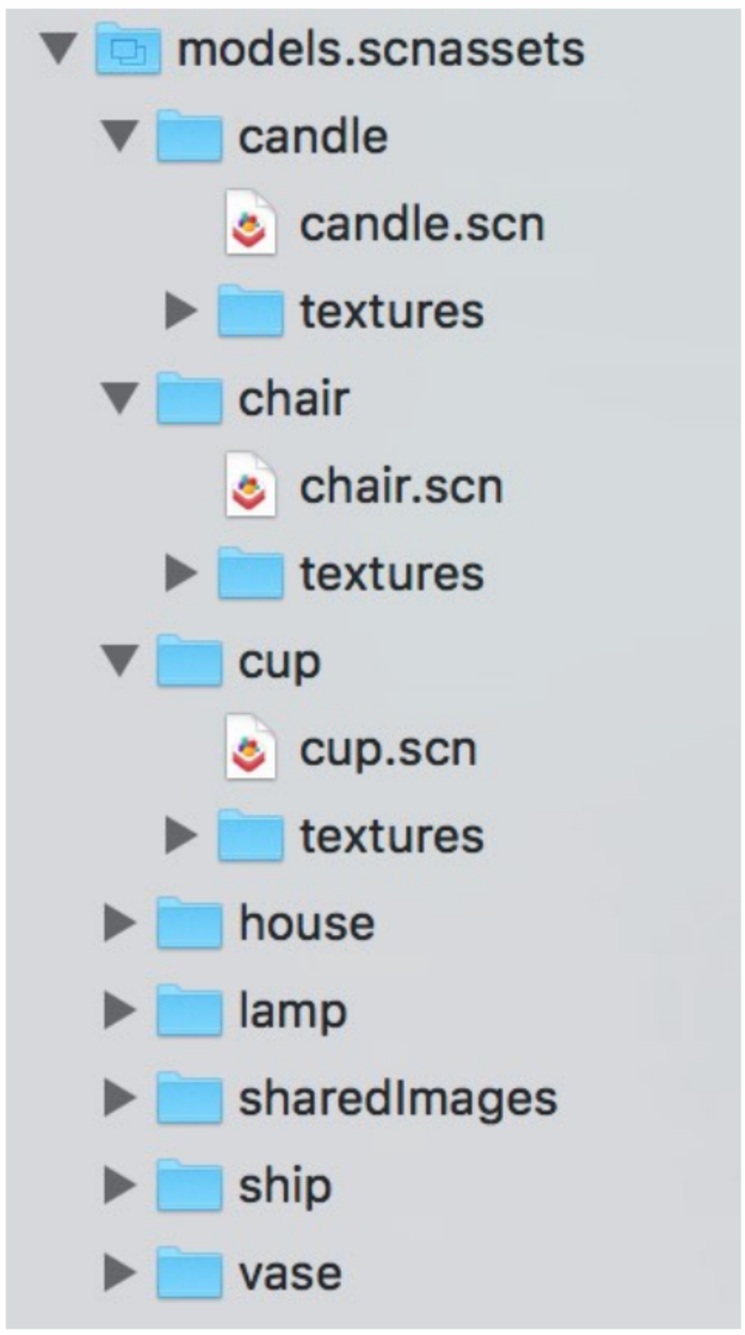
PART TWO—CUSTOMIZE THE OBJECT LISTS

There are plenty of opportunities to customize the assets available for scene creation. In addition to deciding which SceneKit primitives the user can select, you can also specify which colors and sizes they can be. As you proceed through this section of the guide, feel free to use the provided code snippets—they're here to help you learn how to

display the different choices. But it's strongly recommended that you take the time to choose your *own* options so that you can differentiate your version of the app.

Custom Models

By choosing Select Scene File from the popover, users can place any models that live inside `models.scnassets`. But the `.scn` files need to be placed inside a folder with the same name as the model; otherwise they won't be listed properly. In the following image, you can see that `candle.scn` is located inside the "candle" folder, `chair.scn` inside the "chair" folder, and so on.



Where did these models come from? Although not provided in the starter project, they're easy to obtain. If you'd like to use them in your app, check out the sample code, Handling 3D Interaction and UI Controls in Augmented Reality. You can download the project and copy the models from its `.scnassets` folder.

Shapes

In the unmodified version of the starter project, the `ShapeOption` enum has a box, a sphere, a cylinder, a cone, and a torus as its options. But what if you want to replace the torus with a pyramid? Begin by updating the enum to contain a new set of cases.

```
enum Shape  String
    case box = "Box", sphere = "Sphere", cylinder = "Cylinder",
       cone = "Cone", pyramid = "Pyramid"
}
```

When you compile the code, you'll receive two errors. The first is in the `shapePicker()` method in `OptionsContainerViewController`, which has a collection of `Shape` cases.

```
let shapes: [Shape] = [.box, .sphere, .cylinder, .cone, .torus]
```

Since `.torus` is no longer a valid enum case, you'll need to update the method to use `.pyramid` instead.

```
let shapes: [Shape] =
   [.box, .sphere, .cylinder, .cone, .pyramid]
```

The second error is in the `createNode(shape:, color:, size:)` method, which is called when the user finishes selecting the shape, its color, and its size. Similar to the previous error, `.torus` is no longer valid in the `switch` statement, so you'll need to update the case to initialize an `SCNPyramid` instead of an `SCNTorus`.

```
case .pyramid:
    geometry = SCNPyramid(width: meters, height: meters * 1.5,
       length: meters)
```

Shape Sizes

In the starter project, the choices for a shape's size are small, medium, and large. But what if you wanted to add an extra large option? Just as you did with the `Shape` enum, you'll update the `Size` enum to include a new case.

```
enum Size  String
    case small = "Small", medium = "Medium", large = "Large",
       extraLarge = "Extra Large"
}
```

Build and run the app. It's probably no surprise that you'll receive an error. Once again, the `createNode(shape:, color:, size:)` is the culprit, because the `switch` statement doesn't know how to handle the `.extraLarge` case. You can add a new case that sets the `meters` constant to resolve the issue.

```
case .extraLarge:
    meters = 0.6
```

Take a moment to study what this method does. It takes the user's choice of size to set `meters`, which is then used to size each of the different shapes. When the user selects Box as the shape, the initialized `SCNBox` object has a length, width, and height equal to `meters`. But you're free to decide how each shape is sized in relation to the `meters` value. To create your own sizes, play around with different values for `meters`, observing how the value is utilized in the initializer for each geometry.

Shape Colors

You may have noticed that there isn't an enum to control which colors are listed in the popover. Locate the `colorPicker()` method in the project, where you'll discover that the color choices are a collection of `(String, UIColor)` tuples. The first tuple value is the title to be displayed, and the second value is the actual color that will be applied to the shape. Adjust this collection to include the colors you want to use in your app.

```
let colors: [(String, UIColor)] = [("Red", .red),
   ("Yellow", .yellow), ("Orange", .orange), ("Green", .green),
   ("Blue", .blue), ("Brown", .brown), ("White", .white)]
```

Images

In Lesson 5, you learned how to specify which images should be discovered by ARKit. Open the `Assets.xcassets` folder in the project navigator, then locate the AR Resources Group that's already created. Add your choice of images there—and don't forget to declare each image's width and height in the Attributes inspector.

Up to this point, you've declared which models, shapes, shape attributes, and images you'd like to be used in your app. Now it's time to start adding functionality so that objects can be placed into the scene.

PART THREE—FREEFORM OBJECTS

The segmented control's Freeform option will place the user's selected shape or model 20cm in front of the camera's current position. But first, the app will need to know *which* SceneKit object has been selected, and you'll need to write some code that detects a tap on the screen.

Store the Selected Object

In Part One of this guided project, you learned that the `objectSelected(node:)` method is called when the user is finished selecting a shape or model. To store that information (so you can use it whenever the screen is tapped), declare a `selectedNode` variable of type `SCNNode?`, then update its value using the method's `node` parameter. Since the user won't have selected anything when the app is first launched, the variable needs to be an optional.

```
var selectedNode: SCNNode?

func objectSelected(node: SCNNode) {
    dismiss(animated: true, completion: nil)
    selectedNode = node
}
```

Detecting Taps

In other apps, you've used buttons and gesture recognizers to detect screen taps. Could one of them be useful here? Review the video of the completed app at the beginning of

this guide. The landing pad was drawn by dragging a finger across the display—which means you can't use a button or a tap gesture recognizer. Even though you're not yet placing objects on discovered planes, it's important to consider how to structure your code to allow for that interaction.

There's a method, `touchesBegan(_:, with:)`, that's part of the `UIViewController` class. It's triggered whenever a finger or multiple fingers touch the device's screen. The first parameter is a collection of one or more touches, so it should contain at least one value that corresponds to a finger. Inside this method, you can decide how to respond to the touch depending on the current value of `objectMode`. Here's an example implementation:

```
override func touchesBegan(_ touches: Set<UITouch>, with event:
    UIEvent?) {
     super.touchesBegan(touches, with: event)

     guard let node = selectedNode, let touch =
         touches.first else { return }

     switch objectMode {
     case .freeform:
         addNodeInFront(node)
     case .plane:
         break
     case .image:
         break
     }
}
```

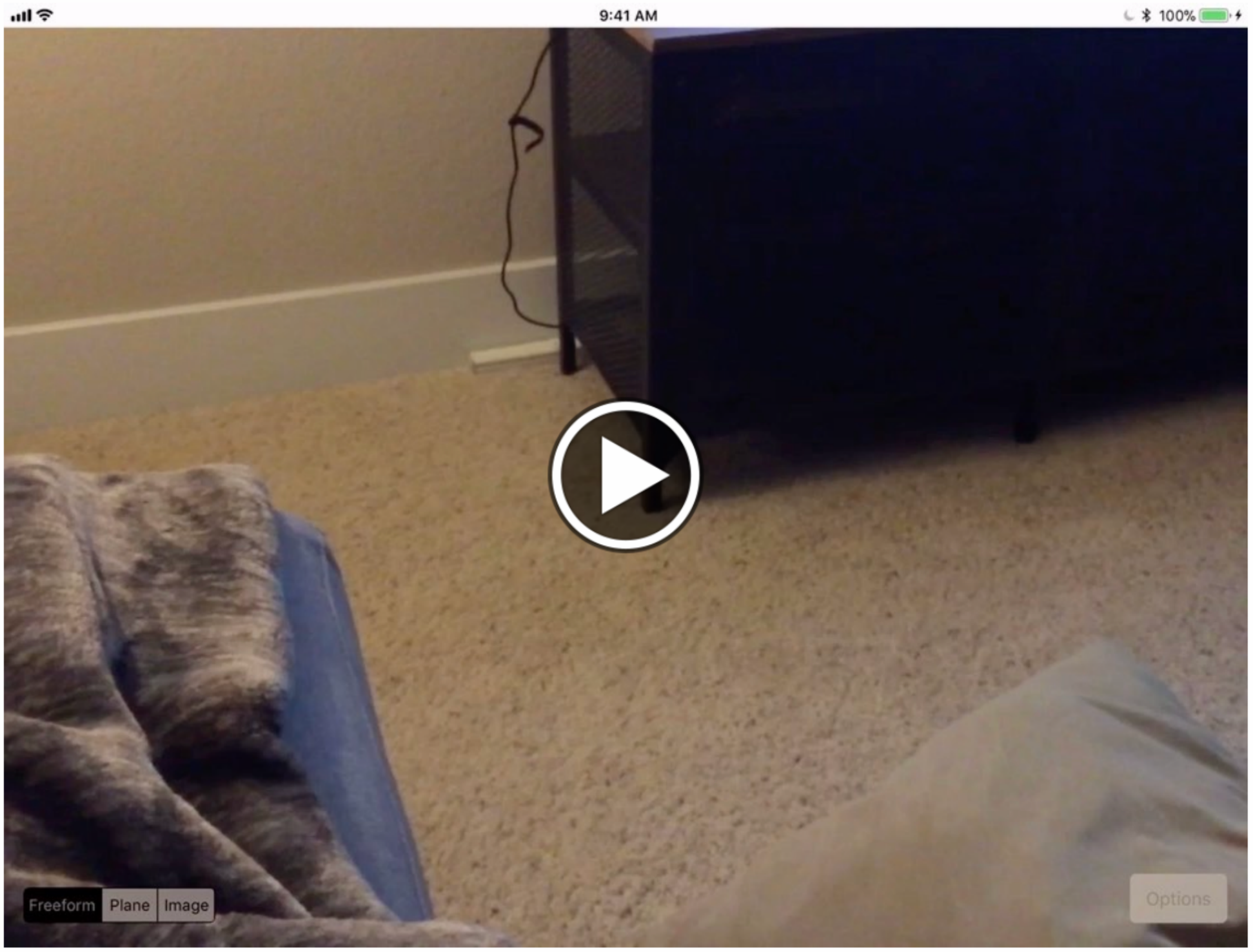
Following the standard convention for other `override` methods (such as `viewDidLoad()`), begin by using `super` to call the parent's implementation of the method. To verify that the user has chosen an object from the popover, you can check if `selectedNode` has a value, then check to see if at least one finger has touched the screen. If both of these conditions are satisfied, inspect the `objectMode` variable to determine how the object should be placed in the scene. For the Freeform setting (which

is selected by default in the segmented control), call a new method and pass it the node. For all other cases, you can `break` out of the `switch` statement (because you haven't yet implemented them).

In earlier lessons, you placed objects in front of the camera—so the `addNodeInFront(_:)` method should be relatively straightforward. Grab the camera's transform by accessing the session's `currentFrame`, then set the node's transform to be equal to the camera's transform, plus an additional 20cm in front (along the z-axis). Since you can place multiple copies of the node with multiple taps, you can use the node's `clone()` method and add the resulting copy to the scene.

```
func addNodeInFront(_ node: SCNNode) {
    guard let currentFrame = sceneView.session.currentFrame
        else { return }

    // Set transform of node to be 20cm in front of camera
    var translation = matrix_identity_float4x4
    translation.columns.3.z = -0.2
    node.simdTransform =
        matrix_multiply(currentFrame.camera.transform,
        translation)

    let cloneNode = node.clone()
    sceneView.scene.rootNode.addChildNode(cloneNode)
}
```

Build and run your app. After selecting an object, tap the screen and the selected object should appear in front of the camera. Depending on the size and shape of the object, you may need to adjust your position to get a better view.



Thinking Ahead

So far, you've completed one of the three modes for placing objects. Take a moment to consider if any of the code you wrote to handle Freeform placement is applicable to the other two modes (placing objects on top of planes or in response to image detection). You should also check to see if the code you've written may conflict with other options, such as the undo functionality.

Cloning Nodes

Regardless of whether an object is placed in front of the camera, on a horizontal surface, or on top of an image, the node you create needs to be cloned and added to the scene's root node. Perform this work in a reusable method called `addNodeToSceneRoot(_:)`, then replace the necessary lines in `addNodeInFront(_:)`.

```
func addNodeInFront(_ node: SCNNode) {
    guard let currentFrame = sceneView.session.currentFrame
```

```
        else { return }

    // Set transform of node to be 20cm in front of camera
    var translation = matrix_identity_float4x4
    translation.columns.3.z = -0.2
    node.simdTransform =
        matrix_multiply(currentFrame.camera.transform,
        translation)

    addNodeToSceneRoot(node)
}

func addNodeToSceneRoot(_ node: SCNNode) {
    let cloneNode = node.clone()
    sceneView.scene.rootNode.addChildNode(cloneNode)
}
```

Storing Placed Objects

The Undo Last Object option presents an interesting problem: How do you know which node should be removed from the scene? You might think removing the root node's last child node is the solution—and that works in most cases. But you'll run into an issue when visualizing detected planes, because you'll undo nodes that don't represent objects placed by the user.

The best way to prevent this problem is to maintain your own collection of nodes. When the user taps the screen, you can add the node into a collection called `placedNodes`. Nodes added during plane detection can be added into a separate collection, `planeNodes`. With two distinct collections, implementing Undo Last Object, Reset Scene, and toggling plane visualization should be trivial.

```
var placedNodes = [SCNNode]()
var planeNodes = [SCNNode]()

func addNodeToSceneRoot(_ node: SCNNode) {
    let cloneNode = node.clone()
```

```
        sceneView.scene.rootNode.addChildNode(cloneNode)
        placedNodes.append(cloneNode)
    }
```

PART FOUR—IMAGE DETECTION

Now that your code is better structured, you're ready to add the ability to place objects in the scene using image detection. The `renderer(_:, didAdd:, for:)` method is used for both image and plane detection, but `ARImageAnchor` is a little easier to work with than `ARPlaneAnchor`—since it doesn't grow in size or need visual feedback. When the method is called, inspect the anchor and determine whether it's an `ARImageAnchor` or an `ARPlaneAnchor`.

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node:
    SCNNode, for anchor: ARAnchor) {
     if let imageAnchor = anchor as? ARImageAnchor {
         nodeAdded(node, for: imageAnchor)
     } else if let planeAnchor = anchor as? ARPlaneAnchor {
         nodeAdded(node, for: planeAnchor)
     }
}

func nodeAdded(_ node: SCNNode, for anchor: ARPlaneAnchor

}

func nodeAdded(_ node: SCNNode, for anchor: ARImageAnchor

}
```

What should you do when an image is discovered? First, you need to verify that `selectedNode` contains a value; otherwise, you don't know what object to place on top of the image. If the user has selected an object from the popover, you can proceed with object placement. As you'll recall from Lesson 5, nodes you create in response to image

detection should be added as children of the node that ARKit created when the image was discovered—not as children of the root node. Instead of using `addNodeToSceneRoot(_:)`, you'll need a method that does nearly identical work, attaching the node to the proper parent.

```
func nodeAdded(_ node: SCNNode, for anchor: ARImageAnchor) {
    if let selectedNode = selectedNode {
        addNode(selectedNode, toImageUsingParentNode: node)
    }
}

func addNode(_ node: SCNNode, toImageUsingParentNode parentNode:
   SCNNode) {
    let cloneNode = node.clone()
    parentNode.addChildNode(cloneNode)
    placedNodes.append(cloneNode)
}
```

Adjusting the Configuration

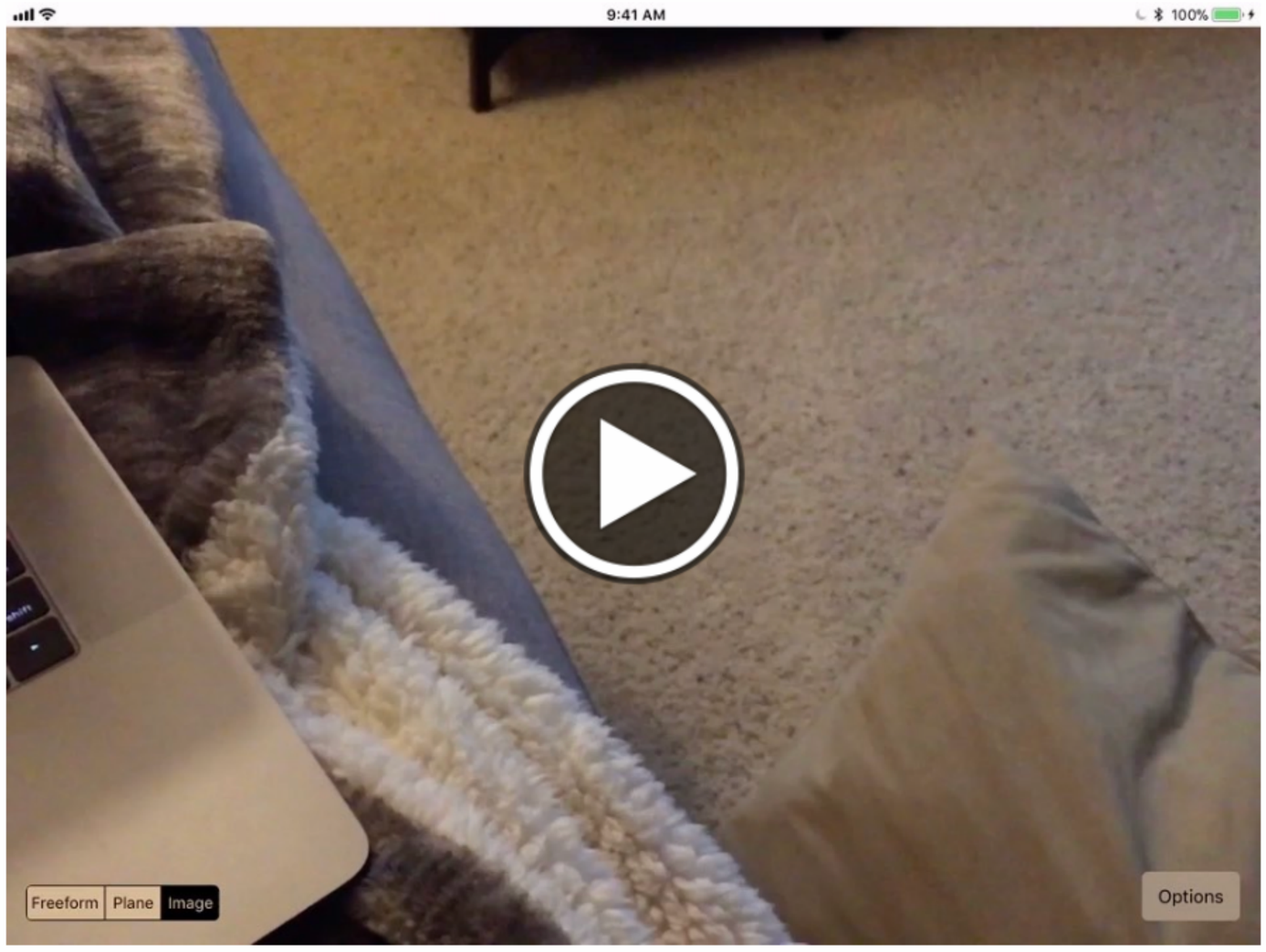
To finish this feature, you'll instruct ARKit to look for images inside the AR Resources Group. But should image detection take place if the segmented control is set to Freeform or to Plane? The answer is no. Objects should only be appearing on top of discovered images when Image is selected. So you'll need to provide a way to reload the session configuration whenever there's a change in the segmented control's state.

Create a method called `reloadConfiguration()`. This method will update the session configuration's `detectionImages` property based on the value of `objectMode`. If the app is in Image mode, `detectionImages` should contain the images in `AR Resources`; if the app is in one of the other two modes, `detectionImages` should be `nil`. Call the method in `viewWillAppear(_:)`, as well as whenever `objectMode` changes.

```
var objectMode: ObjectPlacementMode = .freeform {
    didSet {
        reloadConfiguration()
    }
}
```

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    reloadConfiguration()
}

func reloadConfiguration() {
    configuration.detectionImages = (objectMode == .image) ?
        ARReferenceImage.referenceImages(inGroupNamed: "AR
        Resources", bundle: nil) : nil

    sceneView.session.run(configuration)
}
```

Build and run your app. Select a SceneKit object, then switch to Image mode via the segmented control. As you hover over discovered images, the object should be placed above it.



PART FIVE—PLANE DETECTION AND VISUALIZATION

To enable plane detection, set the configuration's `planeDetection` to a value inside `reloadConfiguration()`. You can search for horizontal planes, vertical planes, or both. Is there a time when plane detection should be disabled? Unlike image detection, you should allow plane detection at all times so that ARKit can constantly update and improve its knowledge of the surroundings.

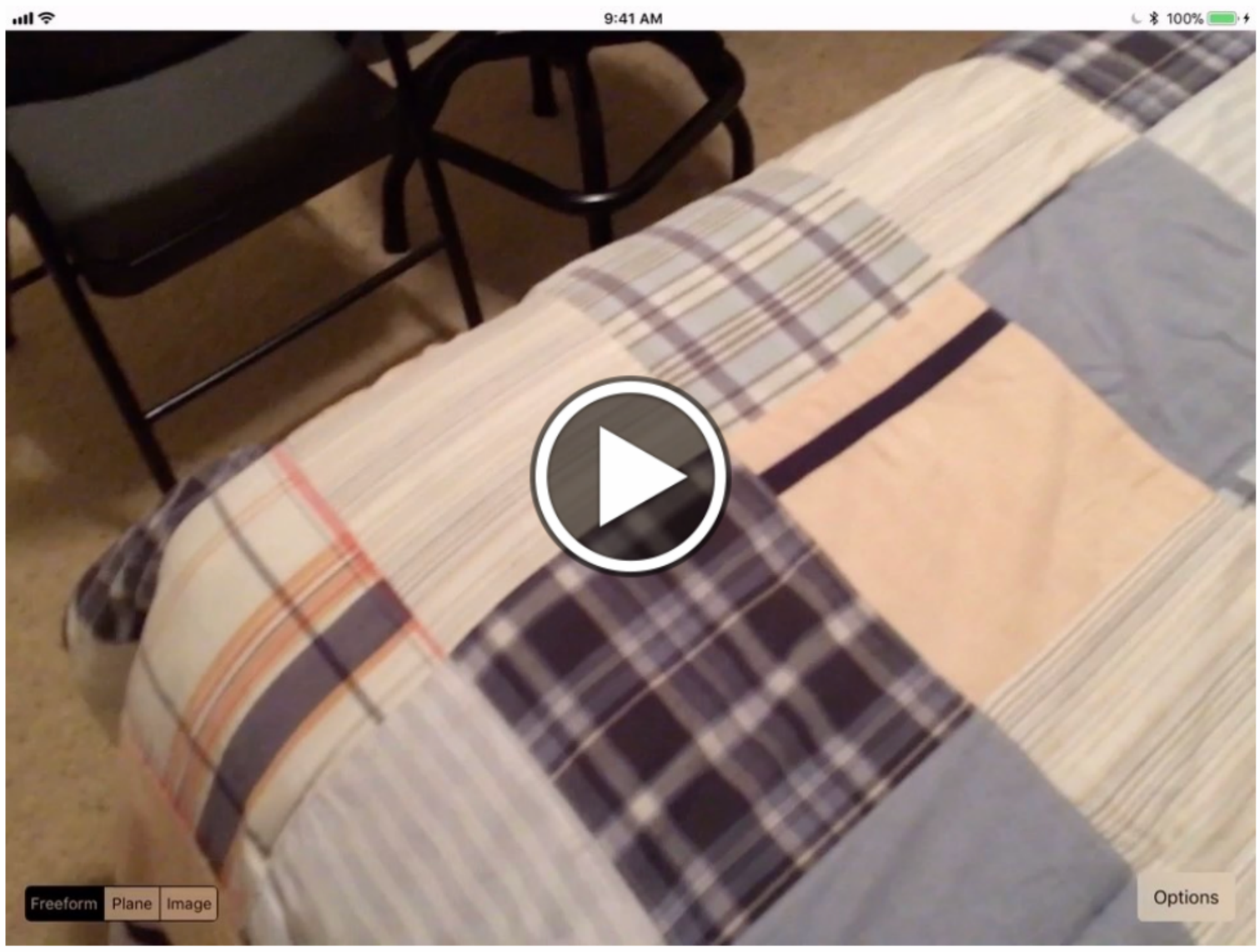
```
func reloadConfiguration() {
    configuration.planeDetection = .horizontal
    configuration.detectionImages = (objectMode == .image) ?
       ARReferenceImage.referenceImages(inGroupNamed:
       "AR Resources", bundle: nil) : nil

    sceneView.session.run(configuration)
}
```

In Part Four, you created a `nodeAdded(_:, for:)` method for dealing with an `ARPlaneAnchor`, but it's still empty. You'll need to create an `SCNPlane` that's sized to match the dimensions of the plane anchor—similar to the work you did in Lesson 3. Next, rotate the plane by -90 degrees to account for the orientation differences between ARKit and SceneKit. Add the node to the scene, add it to your `planeNodes` collection, then update the position and size of the plane as more information is gathered by the framework.

```
func nodeAdded(_ node: SCNNode, for anchor: ARPlaneAnchor
    let floor = createFloor(planeAnchor: anchor)

    node.addChildNode(floor)
    planeNodes.append(floor)
}

func renderer(_ renderer: SCNSceneRenderer, didUpdate node:
   SCNNode, for anchor: ARAnchor) {
```

```
        guard let planeAnchor = anchor as? ARPlaneAnchor,
            let planeNode = node.childNodes.first,
            let plane = planeNode.geometry as? SCNPlane else {
                return
            }

        planeNode.position = SCNVector3(planeAnchor.center.x, 0,
            planeAnchor.center.z)
        plane.width = CGFloat(planeAnchor.extent.x)
        plane.height = CGFloat(planeAnchor.extent.z)
    }

    func createFloor(planeAnchor: ARPlaneAnchor) -> SCNNode {
        let node = SCNNode()

        let geometry = SCNPlane(width:
            CGFloat(planeAnchor.extent.x), height:
            CGFloat(planeAnchor.extent.z))
        node.geometry = geometry

        node.eulerAngles.x = -Float.pi / 2
        node.opacity = 0.25

        return
    }
```

Build and run your app. You should see a semitransparent plane over the top of detected surfaces.



Disabling Plane Visualization

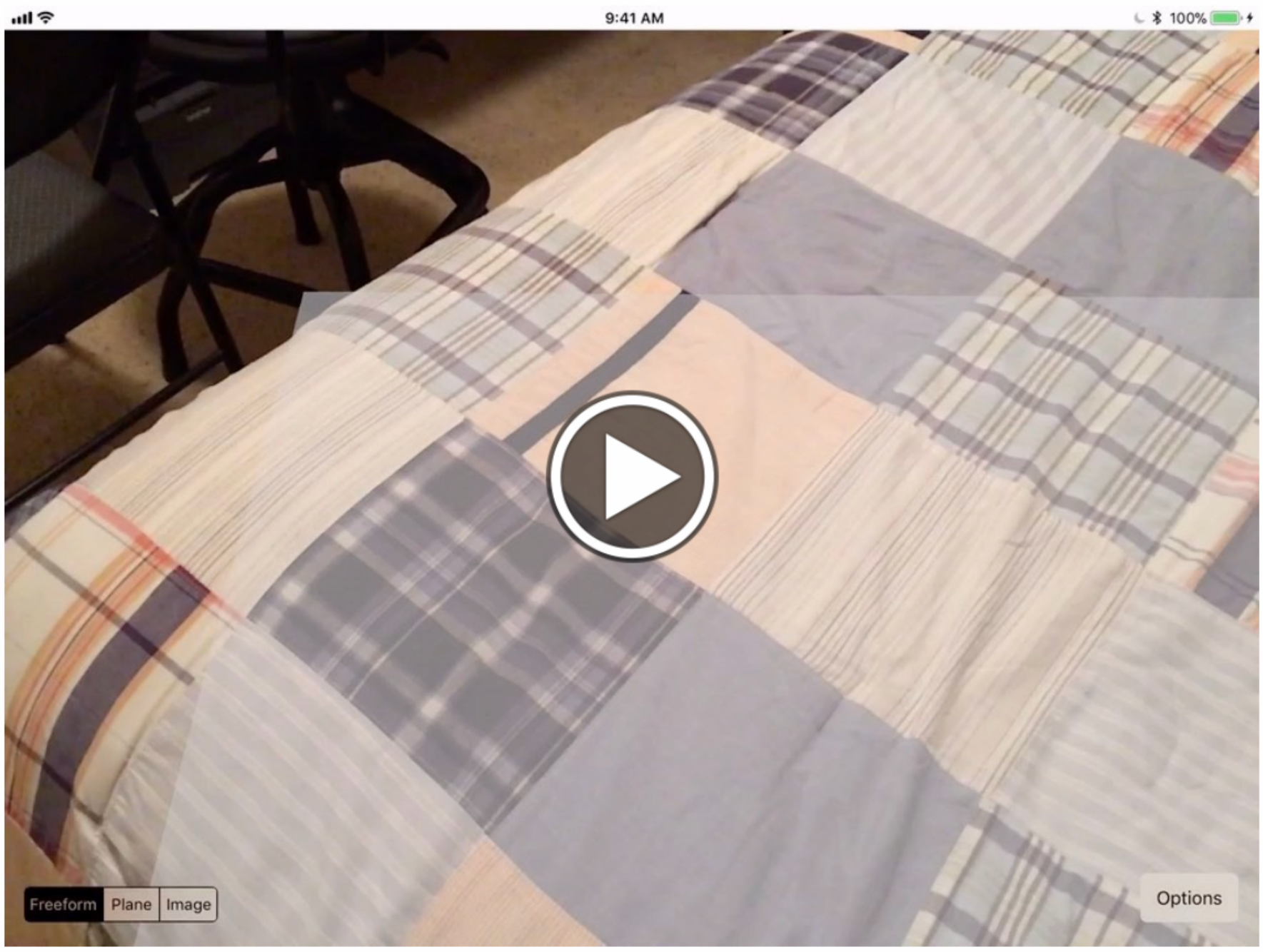
Plane visualization is a great feature for verifying where you can place objects in Plane mode. At the same time, however, the planes disrupt the appearance of the scene. To enable the user to show/hide `SCNPlane` nodes, `togglePlaneVisualization` can be triggered whenever the user selects Enable/Disable Plane Visualization from the popover.

Because plane nodes are either visible or invisible, a Boolean would work well to control whether to show the planes. When the Boolean value changes from `false` to `true`, or from `true` to `false`, update the `isHidden` property on the nodes to match. This is a simple loop, since you're already maintaining a list of the planes inside `planeNodes`.

```
var showPlaneOverlay = false {
    didSet
        for node in planeNodes {
            node.isHidden = !showPlaneOverlay
        }
```

```
    }
}

func nodeAdded(_ node: SCNNode, for anchor: ARPlaneAnchor
    let floor = createFloor(planeAnchor: anchor)
    floor.isHidden = !showPlaneOverlay

    node.addChildNode(floor)
    planeNodes.append(floor)
}

func togglePlaneVisualization() {
    dismiss(animated: true, completion: nil)
    showPlaneOverlay = !showPlaneOverlay
}
```

Build and run the app. You should now be able to toggle between enabling and disabling plane visualization.



PART SIX—PLACING OBJECTS ON PLANES

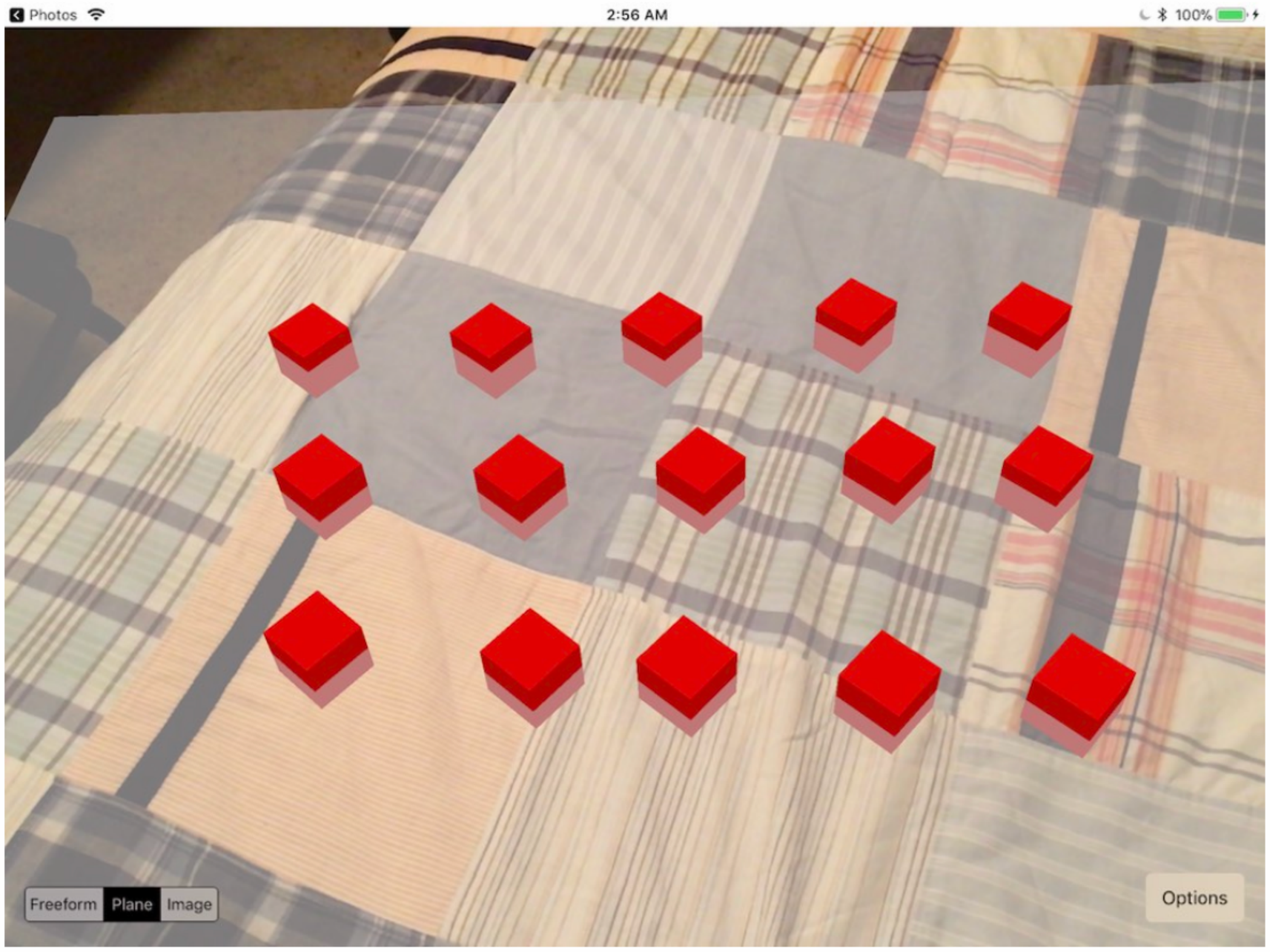
You've made it to the last mode for placing objects in the scene. When you were placing objects in Freeform mode, you used the `touchesBegan(_:, with:)` method to detect taps on the screen. Now it's time to revisit that method and determine what to do in Plane mode.

Perform a Hit Test

In Plane mode, a tap should be ignored if it doesn't land on a detected surface. How will ARKit know? You'll perform a hit test to determine if the tap intersected with an existing plane. First grab the location of the user's finger on the screen, then perform a hit test at that location. If there's a match, update the node's position to the tap's location in the real world, then add it to the scene.

```
override func touchesBegan(_ touches: Set<UITouch>,
    with event: UIEvent?) {
     ...
     case .plane:
         let touchPoint = touch.location(in: sceneView)
         addNode(node, toPlaneUsingPoint: touchPoint)
     ...
}

func addNode(_ node: SCNNode, toPlaneUsingPoint point:
    CGPoint) {
     let results = sceneView.hitTest(point, types:
        [.existingPlaneUsingExtent])

     if let match = results.first {
         let t = match.worldTransform
         node.position = SCNVector3(x: t.columns.3.x, y:
            t.columns.3.y, z: t.columns.3.z)

         addNodeToSceneRoot(node)
     }
}
```

Build and run the app. Use the segmented control to switch to Plane mode, then add an object onto a plane. If you're not seeing the object in the scene, use the plane visualization feature you just created to ensure that you're tapping on a discovered surface.



Enabling Finger Dragging

There's another fun feature of Plane mode. Users should be able to drag a finger along the screen, creating new nodes that follow the same path. There are two additional methods that you can implement from `UIViewController` that will provide updated information about the touch position. The first is `touchesMoved(_:, with:)`, called after the finger has moved from its starting location. The second method is `touchesEnded(_:, with:)`, which provides the location of the user's final touch on the screen.

Here's a simple implementation of `touchesMoved(_:, with:)` to get you started. It does roughly the same work as `touchesBegan(_:, with:)`.

```
override func touchesMoved(_ touches: Set<UITouch>,
    with event: UIEvent?) {
     super.touchesMoved(touches, with: event)

     guard objectMode == .plane,
         let node = selectedNode,
         let touch = touches.first else {
             return
         }

     let newTouchPoint = touch.location(in: sceneView)
     addNode(node, toPlaneUsingPoint: newTouchPoint)
}
```

Build and run the app. With this code in place, you should be able to drag your finger across the screen, and a constant stream of nodes will pour out from under your fingertip. In fact, you'll see that it's *too many* nodes—the objects simply blur together.

Limiting Node Creation

How can you reduce the number of nodes created by the drag of a finger? You can set a threshold constant, `touchDistanceThreshold`, that determines how far a finger will travel before adding another node. You'll start by storing the finger's location when its touch created the last node—the `lastObjectPlacedPoint` in the following snippet. Update the variable's value in `addNode(_:, toPlaneUsingPoint:)`.

```
var lastObjectPlacedPoint: CGPoint?
let touchDistanceThreshold: CGFloat = 40.0

func addNode(_ node: SCNNode, toPlaneUsingPoint point:
    CGPoint) {
     let results = sceneView.hitTest(point, types:
        [.existingPlaneUsingExtent])

     if let match = results.first {
          let t = match.worldTransform
          node.position = SCNVector3(x: t.columns.3.x, y:
             t.columns.3.y, z: t.columns.3.z)

          addNodeToSceneRoot(node)
          lastObjectPlacedPoint = point
     }
  }
```

Now you're ready to calculate the distance between `lastObjectPlacedPoint` and the newest touch position. To calculate the distance between any two points, you can use the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

In this scenario, *a* represents the x-distance between the points, *b* is the y-distance, and *c* is the distance between two. Using Swift code to solve for *c*, it would look something like the following:

```
override func touchesMoved(_ touches: Set<UITouch>, with
event: UIEvent?) {
    super.touchesMoved(touches, with: event)

    guard objectMode == .plane,
        let node = selectedNode,
        let touch = touches.first,
        let lastTouchPoint = lastObjectPlacedPoint else {
            return
        }

    let newTouchPoint = touch.location(in: sceneView)
    let distance = sqrt(pow((newTouchPoint.x -
    lastTouchPoint.x), 2.0) + pow((newTouchPoint.y -
    lastTouchPoint.y), 2.0))

    if distance > touchDistanceThreshold {
        addNode(node, toPlaneUsingPoint: newTouchPoint)
    }
}
```

When the user lifts their finger from the screen, the dragging interaction has ended, so the value stored in `lastObjectPlacedPoint` no longer applies.

```
override func touchesEnded(_ touches: Set<UITouch>,
   with event: UIEvent?) {
    super.touchesEnded(touches, with: event)
```

```
        lastObjectPlacedPoint = nil
    }
```

Build and run the app with this code in place. After doing some testing, adjust the threshold value to an amount that makes sense for your scene.

PART SEVEN—USABILITY FEATURES

You're almost to the end! Users can place a model or shape into augmented reality and position the object based on the camera position, by image detection, or over the top of detected planes. But there are still two popover options you haven't implemented: Undo Last Object and Reset Scene.

Undo Last Object

From the starter project, the `undoLastObject()` method is the only method that doesn't call `dismiss(animated:, completion:)` at the beginning. Can you guess why? Imagine a user wants to undo the last ten objects that were added to the scene. If the popover were dismissed after tapping Undo Last Object, you'd have to redisplay it

before the next undo operation could be performed—very tedious! The solution is to not dismiss the popover.

Undoing the most recent object placement is as simple as removing the most recently placed node. Locate the `last` node in `placedNodes`, remove it from the scene, then remove it from your collection of nodes.

```
func undoLastObject() {
    if let lastNode = placedNodes.last {
        lastNode.removeFromParentNode()
        placedNodes.removeLast()
    }
}
```

Build and run the app to check the undo functionality. Add multiple objects to the scene, then remove them one at a time with each tap of the Undo button.

Reset Scene

Resetting the scene is a little more complicated. In addition to removing nodes, it involves removing any discovered anchor points for both planes and images. Inside `reloadConfiguration()`, you can supply the option `.removeExistingAnchors` to the `run(_:, options:)` method. But since `reloadConfiguration()` is called whenever the user interacts with the segmented control, you'd trigger an unintentional removal of anchors.

One solution is to update `reloadConfiguration()` to include a `removeAnchors` Boolean argument. The Boolean can control whether or not `.removeExistingAnchors` is an option to include. If the Boolean is `true`, include it; if not, leave `options` blank.

```
func reloadConfiguration(removeAnchors: Bool = true) {
    configuration.planeDetection = [.horizontal, .vertical]
    configuration.detectionImages = (objectMode == .image) ?
       ARReferenceImage.referenceImages(inGroupNamed:
       "AR Resources", bundle: nil) : nil
```

```
        let options: ARSession.RunOptions

        if removeAnchors {
            options = [.removeExistingAnchors]
            for node in planeNodes {
                node.removeFromParentNode()
            }
            planeNodes.removeAll()
            for node in placedNodes {
                node.removeFromParentNode()
            }
            placedNodes.removeAll()
        } else {
            options = []
        }
        sceneView.session.run(configuration, options: options)
    }
```

When the segmented control is changed, it updates the value of `objectMode`, which calls `reloadConfiguration()`. This is a case where you do *not* want to remove anchors, so pass `false` as the parameter:

```
    var objectMode: ObjectPlacementMode = .freeform {
        didSet
            reloadConfiguration(removeAnchors: false)
        }
    }
```

When `resetScene()` is called, you should remove the anchors, so use the default value (`true`):

```
func resetScene() {
    dismiss(animated: true, completion: nil)
    reloadConfiguration()
}
```

Build and run your app. Use the Reset Scene feature to wipe out the entire scene and begin reconstructing the scene from a clean slate.

WRAPUP

Congratulations on building an app with ARKit! If you've never coded in 3D before, you've gained a lot of knowledge in five short lessons. Maybe you came into this project with some hesitation. Your work on this app should have helped you understand when to call particular delegate methods and how to position objects properly.