visionOS Development

visionOS 1 • Xcode 15

STUDENT GUIDE

ABOUTOBJECTS

Contact

About Objects
11951 Freedom Drive
Suite 700
Reston, VA 20190

main: 571–346–7544
email: info@aboutobjects.com
web: www.aboutobjects.com

Course Information

Author: Jonathan Lehr
Revision: 1.0.0
Last Update: 06/23/2024

Classroom materials for a course that provides a rapid introduction to visionOS development in SwiftUI.

Cover photo by Bram Van Oost on Unsplash

Copyright Notice

visionOS Development

STUDENT GUIDE

About the Author

Jonathan Lehr
Co-Founder and VP, Training
About Objects

jonathan@aboutobjects.com

SECTION ONE

visionOS Overview

The visionOS Framework

Both UIKit and SwiftUI provide spatial computing support.

Note that many SwiftUI components depend on UIKit components such as `UIView`, `UIViewController`.

Dependencies are largely hidden; however, your code may sometimes need to call directly into UIKit.

Developer Tools

Xcode's SwiftUI Preview Pane

visionOS simulator

RealityComposer Pro for configuring 3D models and shaders

RealityKit

Framework for presentation and management of 3D content.

Integrated with ARKit for when your app needs access to sensor data, such as scene understanding or hand tracking input.

Provides APIs for loading and managing 3D models and offscreen objects such as audio emitters and lights.

Reacts to user input as well as changes in the user's environment, for example, changes in ambient lighting and motion of real-world objects.

```
// Imports the RealityKit API.
import RealityKit

// Use if your code calls APIs that access
// RealityComposer Pro content.
import RealityKitContent
```

SECTION TWO
Scene Management

Main Window

An app is composed of one or more instances that conform to the `Scene` protocol, where a scene is a container for a view hierarchy.

Inside your app's body property (which returns a Scene), define one or more instances of `WindowGroup` as its content.

Note that `ImmersiveSpace`, which we'll cover shortly, also conforms to `Scene` and can also be defined in the body.

```
import SwiftUI

@main
struct WindowGroupsApp: App {

    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Auxiliary Windows

To add another window, define another `WindowGroup` and pass an explicit value for the initializer's `id` parameter.

You can use the `id` value later to programmatically open and dismiss the window.

```
import SwiftUI

@main
struct WindowGroupsApp: App {

    var body: some Scene {
        WindowGroup {
            ContentView()
        }

        WindowGroup(id: Self.windowTwo) {
            Text(Self.windowTwo)
                .font(.title)
        }
    }
}
```

Presenting a Window

Use the `openWindow` and `dismissWindow` environment values to present and dismiss windows programmatically. (There are also `openImmersiveSpace` and `dismissImmersiveSpace` actions.)

Note that the above actions are defined as structs that implement the `callAsFunction(id:)` method to provide syntactic sugar.

```
struct ContentView: View {
    @Bindable var viewModel: ViewModel

    @Environment(\.openWindow) private var openWindow
    @Environment(\.dismissWindow) private var dismissWindow

    var body: some View {
        VStack {
            Text("WindowGroups Example")
            Toggle("Show Window Two",
                   isOn: $viewModel.isShowingWindowTwo)
        }
        .onChange(of: viewModel.isShowingWindowTwo) { _, shouldShow in
            if shouldShow {
                openWindow(id: "Window Two")
            } else {
                dismissWindow(id: "Window Two")
            }
        }
    }
}
```

Scene Phases

Check the current scene phase to react to a scene's state transitioning between `active`, `inactive`, and `background`.

```
@main
struct WindowGroupsApp: App {

    @State private var viewModel = ViewModel()
    @Environment(\.scenePhase) private var scenePhase

    var body: some Scene {
        WindowGroup {
            ContentView(viewModel: viewModel)
        }

        WindowGroup(id: "Window Two") {
            Text("Window Two")
                .font(.title)
        }
        .onChange(of: scenePhase) { _, phase in
            if phase == .background {
                print("Background")
                viewModel.isShowingWindowTwo = false
            }
        }
    }
}
```

Volumes

A volume is a window with added depth along the z-axis.

It automatically displays regular window controls, allowing the user to move and dismiss the volume.

As with a regular window, contents are clipped to the window's bounds.

```
WindowGroup(id: "Window Three") {
    Text("Window Three")
        .font(.extraLargeTitle)
        .frame(maxWidth: .infinity, maxHeight: .infinity)
        .background(Color.orange)
}
.defaultSize(width: 400, height: 300, depth: 200)
.windowStyle(.volumetric)
```

Model3D

`Model3D` is a SwiftUI view that loads a 3D model asynchronously and presents it. A Model3D can:

- Load models from a USD file or a Reality file.
- Present a placeholder temporarily while loading.

```
import SwiftUI
import RealityKit

struct ContentView: View {

    var body: some View {
        HStack(spacing: 60) {
            VStack {
                Text("Toy Car")
                    .font(.title)

                Model3D(named: "toy_car") { model in
                    model
                        .resizable()
                        .frame(width: 240, height: 240)
                        .background(Color.orange.gradient,
                                    in: Circle())
                } placeholder: {
                    ProgressView()
                }
            }
        ...
```

Ornaments

Use the `.ornament` modifier to attach a view to the current window.

You specify an attachment anchor (i.e., top, bottom, etc.) and an optional alignment, which offsets or insets the view a small amount.

```
VStack {
    Text("Ornaments and Toolbars")
        .font(.title)
        .padding(.bottom, 50)
}
.padding()
.ornament(attachmentAnchor: .scene(.top),
          contentAlignment: .bottom) {

    HStack(spacing: 15) {
        Button(action: { }) { Image(systemName: "visionpro") }
        Button(action: { }) { Image(systemName: "macbook") }
    }
    .padding(15)
    .glassBackgroundEffect()
}
```

Tab Bar

You create a spatial tab bar using the standard SwiftUI tab bar API, however its visual appearance and behavior are slightly enhanced on the visionOS platform.

On visionOS, the tab bar is presented as an ornament on the leading edge of the window that contains it.

As on other platforms,

- The `.tabItem` modifier specifies the content of a given tab
- The `.tag` modifier allows you to track the selected tab.

```
TabView(selection: $viewModel.selectedTab) {
    BookBrowser(viewModel: viewModel)
        .tabItem {
            Label("Books", systemImage: "books.vertical.fill")
        }
        .tag(Tab.books)
    SpatialObjectBrowser(viewModel: viewModel)
        .tabItem {
            Label("Models", systemImage: "view.3d")
        }
        .tag(Tab.objects)
    SettingsView()
        .tabItem {
            Label("Settings", systemImage: "gear")
        }
        .tag(Tab.settings)
}
```

Toolbars

Use the `.toolbar` modifier to specify toolbar content and placement.

Nest one or more `ToolBarItem` or `ToolBarItemGroup` structs to define the views to be presented in the toolbar.

```
.toolbar {
    ToolbarItemGroup(placement: .bottomBar) {
        Button("Bottom Bar Button One", action: { })
        Button("Bottom Bar Button Two", action: { })
    }
}
```

To place the toolbar in the bottom ornament, specify `.bottomOrnament`.

```
.toolbar {
    ToolbarItemGroup(placement: .bottomOrnament) {
        Button("Toolbar Button One", action: { })
        Divider()
        Button("Toolbar Button Two", action: { })
    }
}
```

Navigation Bar

You specify navigation bar items with `.toolbar`, using the placements `.topBarLeading` or `.topBarTrailing`.

```
.toolbar {
    ToolbarItemGroup(placement: .topBarTrailing) {
        Button("Nav Item One", action: { })
        Button("Nav Item Two", action: { })
    }
}
```

SECTION THREE

Entities and Components

ECS Architecture

RealityKit implements an Entity–Component–System (ECS) architecture to promote code reuse and streamlined performance.

An **Entity** can be a shape, a 3D model, a light, or an invisible item such as an anchor or a sound emitter.

An entity contains a list of **Components**, which are simple containers for specific kinds of state; some examples are: `AnchoringComponent`, `GroundingShadowComponent`, and `Transform`.

Systems provide dynamic behavior for entities. A system dynamically searches for entities based on criteria you specify. It then executes your custom code on every frame to update the state of any matched entities and their components.

Entities

An **entity** encapsulates the state of a RealityKit object.

At a minimum, an entity defines its location in space, even if it has no visual appearance.

An entity can contain other entities as children, potentially forming a graph of entities.

Each entity contains a set of `Component` instances.

Model3D

A model can be loaded directly from the app bundle or from a RealityComposerPro package.

Use `Model3D` to load a model into a `View`.

```
var body: some View {

    VStack {
        Text("Toy Car")
            .font(.title)

        Model3D(named: "toy_car") { model in
            model
                .resizable()
                .frame(width: 240, height: 240)
                .background(Color.orange.gradient, in: Circle())
        } placeholder: {
            ProgressView()
        }
    }
}
```

Loading a Model

Use `ModelEntity` initializers to load a model into a RealityKit entity.

```
// Load with entity name
RealityView { content in
    if let entity = try? await ModelEntity(named: "toy_car") {
        content.add(entity)
    }
}

// Load with URL
RealityView { content in
    if let entity = try? await ModelEntity(contentsOf: modelUrl) {
        content.add(entity)
    }
}
```

The APIs in the above examples load entities with the following initial components.

```
ComponentSet(entity: ▿ 'toy_car' : ModelEntity
  ◈ Transform
  ◈ SynchronizationComponent
  ◈ ModelComponent
```

Component

A **Component** is a simple container representing a particular aspect of an entity's state, and conforms to the `Component` marker protocol.

Some examples:

- `ModelComponent` defines an entity's visual appearance with a mesh and materials.
- `CollisionComponent` contains a shape and other info used for collision detection.
- `InputTargetComponent` specifies that an entity can receive system input such as gestures.
- `GroundingShadowComponent` for model entities that cast a shadow.

Custom Components

A custom component can be used to capture custom state, or can simply be used as a marker type.

```
/// A marker component for blue things.
struct BlueComponent: Component {

}
```

Call the static `registerComponent()` method to register a custom component.

```
@main
struct GesturesApp: App {

    init() {
        BlueComponent.registerComponent()
        PinkComponent.registerComponent()
    }
    ...
```

Configuring Components

To configure components, add them to an entity's `components` property.

```
RealityView { content in
    if let entity = try? await ModelEntity(named: "toy_car") {
        content.add(entity)

        // Either call components.set()...
        entity.components.set(BlueComponent())
        entity.components.set(HoverEffectComponent())
        entity.components.set(
            InputTargetComponent(allowedInputTypes: .indirect))
        entity.components.set(
            GroundingShadowComponent(castsShadow: true))

        // or set the value directly (good for dynamic replacement).
        entity.components[PinkComponent.self] = PinkComponent()
    }
}
```

Configuring the Transform

Entities have a `Transform` component by default.

Convenience APIs to configure the transform's values are provided by the `HasTransform` protocol.

Includes methods for obtaining and modifying the entity's:

- Scale
- Position
- Orientation
- Transform matrix

as well as for converting values from local space to a reference entity.

Also provides APIs for moving an entity over a provided duration with an animation effect.

Anchors

Use an anchor to fix an entity at a given location within a scene.

For example, the code below creates an anchor for a vertical plane:

```
let anchorEntity = AnchorEntity(.plane(.vertical,
                                       classification: .wall,
                                       minimumBounds: [0.5, 0.5]))
```

You can then add child entities to the anchor, as follows:

```
let planeMesh = MeshResource.generatePlane(
    width: 1,
    depth: 0.75,
    cornerRadius: 0.1
)
...

let entity = ModelEntity(mesh: planeMesh, materials: [imageMaterial])

anchorEntity.addChild(entity)
```

Image-based Lighting

IBL requires your Xcode project to have a directory named **Resources** with a nested directory named **Sunlight.skybox** containing a file typically named **Sunlight.png** or **Sunlight.exr**.

That image can then be referenced by corresponding components as shown below:

```
RealityView { content in
    if let entity = try? await ModelEntity(named: "toy_car") {
        content.add(entity)

        Task {
            guard let resource = try? await EnvironmentResource(
                named: "Sunlight") else { return }
            let component = ImageBasedLightComponent(
                source: .single(resource),
                intensityExponent: 9)

            entity.components.set(component)
            entity.components.set(
                ImageBasedLightReceiverComponent(
                    imageBasedLight: entity))
        }
    }
}
```

Creating Entities Programmatically

To create an entity programmatically, start by defining a mesh.

Call the the `ModelEntity` initializer, passing the mesh and one or more materials.

```
private func makeModelEntity(material: SimpleMaterial) -> Entity {

    let mesh = MeshResource.generateBox(size: 0.25,
                                        cornerRadius: 0.01)

    let blueMaterial = SimpleMaterial(color: .systemBlue,
                                      roughness: 0.3,
                                      isMetallic: false)

    let entity = ModelEntity(mesh: mesh,
                             materials: [material])
    ...

    return entity
}
```

PhysicallyBasedMaterial

Materials generally define the surface properties of a 3D model's mesh, such as color, texture, and reflectivity.

`PhysicallyBasedMaterial` provide a variety of parameters you can use to fine-tune a model's appearance.

```
let myPhysicallyBasedMaterial = {
    var material = PhysicallyBasedMaterial()
    material.baseColor = .init(tint: .systemGreen)
    material.metallic = 0.5
    material.clearcoat = 0.9
    material.roughness = 0.4
    material.specular = 0.8
    material.emissiveIntensity = 0.1
    material.emissiveColor = .init(color: .systemBlue)
    // Uncomment the line below to render the mesh.
    // material.triangleFillMode = .lines
    return material
}()
```

RealityComposer Pro

RealityComposer Pro is a visual tool for configuring entities, components, materials, textures, shaders, lights, and audio emitters.

It compiles code and assets into a separate Swift package that can be shared by multiple Xcode targets.

```
import RealityKit
import RealityKitContent

struct ImmersiveView: View {

    var body: some View {

        RealityView { content in
            if let scene = try? await Entity(named: "Toys",
                in: realityKitContentBundle) {

                content.add(scene)

                if let entity = scene.findEntity(
                    named: "ToyBiplane") {

                    ...
                }
```

SECTION FOUR

Gestures

Targeting Gestures

A gesture can be targeted to any entity, to all entities that match a query based on their component configuration, or to a specific entity.

```
// Targeted to entities that contain an instance of BlueComponent
var blueTapGesture: some Gesture {
    TapGesture()
        .targetedToEntity(where: .has(BlueComponent.self))
        .onEnded { value in
            ...
        }
}

// Targeted to any entity
var dragGesture: some Gesture {
    DragGesture()
        .targetedToAnyEntity()
        .onChanged { value in
            ...
        }
        .onEnded { _ in
            ...
        }
```

Configuring Gestures

Configure gestures with the `gesture()` modifier.

Multiple gestures can be configured using the `simultaneousGesture()` modifier.

```
var body: some View {
    RealityView { content in
        ...
    }
    .gesture(dragGesture)
    .simultaneousGesture(blueTapGesture)
    .simultaneousGesture(pinkTapGesture)
}
```

Working with Transforms – 1

The `Entity` type conforms to the `HasTransform` protocol which defines a convenience API for setting properties of its `Transform` component.

For example, the `scale` property in the example below sets the corresponding value in the entity's transform:

```
struct ImmersiveView: View {

    @State private var isScaled = false

    var blueTapGesture: some Gesture {
        TapGesture()
            .targetedToEntity(where: .has(BlueComponent.self))
            .onEnded { value in
                isScaled.toggle()
                value.entity.scale += isScaled ? 1 : -1
            }
    }
```

Working with Transforms – 2

On the other hand, the Tranform's `rotation` property is not covered in the `HasTransform` protocol, so instead you set it directly, as shown below:

```
struct ImmersiveView: View {
    ...
    @State private var isRotated = false
    ...

    var pinkTapGesture: some Gesture {
        TapGesture()
            .targetedToEntity(where: .has(PinkComponent.self))
            .onEnded { value in

                let rotation1 = simd_quatf(
                    angle: .pi / 4,
                    axis: [0, 1, 0]
                )

                let rotation2 = simd_quatf(
                    angle: 0,
                    axis: [0, 1, 0]
                )

                isRotated.toggle()

                value.entity.transform.rotation = isRotated ?
                    rotation1 : rotation2
            }
    }
```

Working with Transforms – 3

When working with the `translation` property, you may need to convert values from an entity's local coordinate system to that of its parent or of the scene.

```
struct ImmersiveView: View {
    ...
    @State private var currentTransform: Transform?
    ...

    var dragGesture: some Gesture {
        DragGesture()
            .targetedToAnyEntity()
            .onChanged { value in
                let entity = value.entity

                if currentTransform == nil {
                    currentTransform = entity.transform
                }

                let translation = value.convert(
                    value.translation3D,
                    from: .local,
                    to: entity.parent!
                )
                entity.transform.translation =
                    currentTransform!.translation + translation
            }
            .onEnded { _ in
                currentTransform = nil
            }
```

SECTION FIVE

Attachments and Systems

Attachments

You can bind a custom view to an entity by configuring an `Attachment`.

To do so, include the optional `attachments` parameter in a `RealityView's` intializer:

```
...
let biplaneLabel = "Biplane"

// Assume this gets loaded elsewhere...
@State private var toyPlaneEntity = Entity()
...

var body: some View {
    RealityView { content, attachments in

        if let biplaneAttachment =
            attachments.entity(for: biplaneLabel) {

            toyPlaneEntity.parent?.addChild(biplaneAttachment)
        }
    } attachments: {
        Attachment(id: biplaneLabel) {
            Text("Biplane")
                .font(.title).bold()
                .padding()
                .glassBackgroundEffect()
        }
    }
}
```

Systems

A RealityKit **System** defines behavior that affects one or more entities in a scene on every frame. Because their code is executed once per frame, make sure any systems you implement avoid doing any unnecessary work.

Note that systems can be nested in a parent system's `dependencies` property, forming a hierarchy.

```
static let query = EntityQuery(where: .has(MyCustomComponent.self))

public func update(context: SceneUpdateContext) {
    let entities = context.scene.performQuery(Self.query).map { $0 }

    guard !entities.isEmpty else { return }

    for entity in entities {
        ...
    }
}
```

Call your custom system's static method `registerSystem()` to tell RealityKit to create and manage an instance for you.

```
struct AttachmentsApp: App {

    init() {
        ...
        MyCustomSystem.registerSystem()
    }
```

ARKit Session

To access sensor input, for example to obtain the current device pose, your custom system will need to configure and run an ARKit session.

```
private let arkitSession = ARKitSession()
private let worldTrackingProvider = WorldTrackingProvider()

public init(scene: RealityKit.Scene) {
    configureArkitSession()
}

func configureArkitSession() {
    Task {
        do {
            try await arkitSession.run([worldTrackingProvider])
        } catch {
            print(error)
        }
    }
}
```

Accessing Device Information

Once you have an ARKit session running, your system's `update(context:)` method will have access to real-time device information, as shown in the following example:

```
public func update(context: SceneUpdateContext) {
    let entities = context.scene.performQuery(Self.query).map { $0 }

    // Get the current device pose
    guard !entities.isEmpty,
          let deviceAnchor = worldTrackingProvider.queryDeviceAnchor(
              atTimestamp: CACurrentMediaTime())
    else { return }

    let translation = Transform(
        matrix: deviceAnchor.originFromAnchorTransform).translation

    // Orient the entities towards the device
    for entity in entities {
        entity.look(at: translation,
                    from: entity.position(relativeTo: nil),
                    relativeTo: nil,
                    forward: .positiveZ)
    }
}
```