Sample Code

Hello World

Use windows, volumes, and immersive spaces to teach people about the Earth.

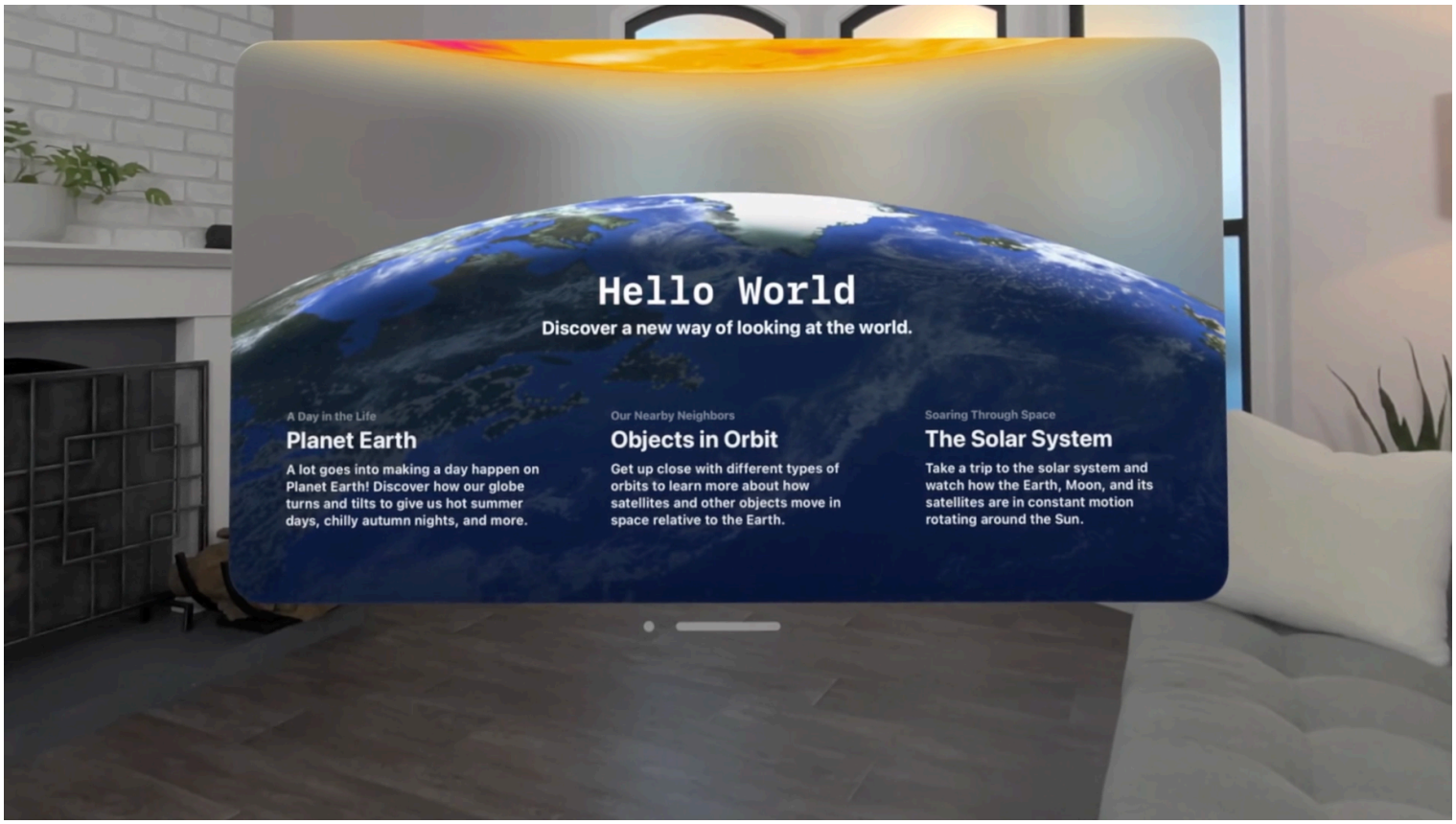
Download

visionOS 2.0+ | Xcode 16.0+

Overview

You can use visionOS scene types and styles to share information in fun and compelling ways. Features like volumes and immersive spaces let you put interactive virtual objects into people's environments, or put people into a virtual environment.

Hello World uses these tools to teach people about the Earth — the planet we call home. The app shows how the Earth's tilt creates the seasons, how objects move as they orbit the Earth, and how Earth appears from space.



Play ⊙

The app uses SwiftUI to define its interface, including both 2D and 3D elements. To create, customize, and manage 3D models and effects, it also relies on the RealityKit framework and Reality Composer Pro.

Create an entry point into the app

Hello World constructs the scene that it displays at launch — the first scene that appears in the `WorldApp` structure — using a `WindowGroup`:

```
WindowGroup("Hello World", id: "modules") {
    Modules()
        .environment(model)
}
.windowStyle(.plain)
```

Like other platforms — for example, macOS and iOS — visionOS displays a window group as a familiar-looking window. In visionOS, people can resize and move windows around the Shared Space. Even if your app offers a sophisticated 3D experience, a window is a great starting point for an app because it eases people into the experience. It's also a good place to provide instructions or controls.

> **Tip**
>
> This particular window group uses the `plain` window style to maintain control over the glass background effect that visionOS would otherwise automatically add.

Present different modules using a navigation stack

After you watch a brief introductory animation that shows the text Hello World typing in, the `Modules` view that defines the primary scene's content presents options to explore different aspects of the world. This view contains a table of contents at the root of a `NavigationStack`:

```
NavigationStack(path: $model.navigationPath) {
    TableOfContents()
        .navigationDestination(for: Module.self) { module in
            ModuleDetail(module: module)
                .navigationTitle(module.eyebrow)
        }
}
```

A visionOS navigation stack has the same behavior that it has in other platforms. When it first appears, the stack displays its root view. When someone chooses an embedded `NavigationLink`, the stack draws a new view and displays a back button in the toolbar. When someone taps the back button, the stack restores the previous view.

The trailing closure of the `navigationDestination(for:destination:)` view modifier in the code above displays a view when someone activates a link based on a `module` input that comes from the corresponding link's initializer:

```
NavigationLink(value: module) { /* The link's label. */ }
```

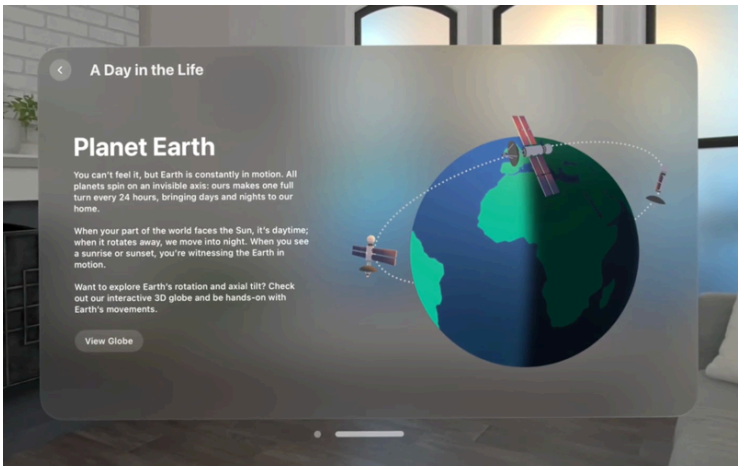
The possible `module` values come from a custom `Module` enumeration:

```
enum Module: String, Identifiable, CaseIterable, Equatable {
    case globe, orbit, solar
```

```
    // ...
}
```

Display an interactive globe in a new scene

The `globe` module opens with a few facts about the Earth in the main window next to a decorative, flat image that supports the content. To help people understand even more, the module includes a button titled View Globe that opens a 3D interactive globe in a new window.



To be able to open multiple scene types, Hello World includes the `UIApplicationSceneManifest` key in its `Information Property List` file. The value for this key is a dictionary that includes the `UIApplicationSupportsMultipleScenes` key with a value of `true`:

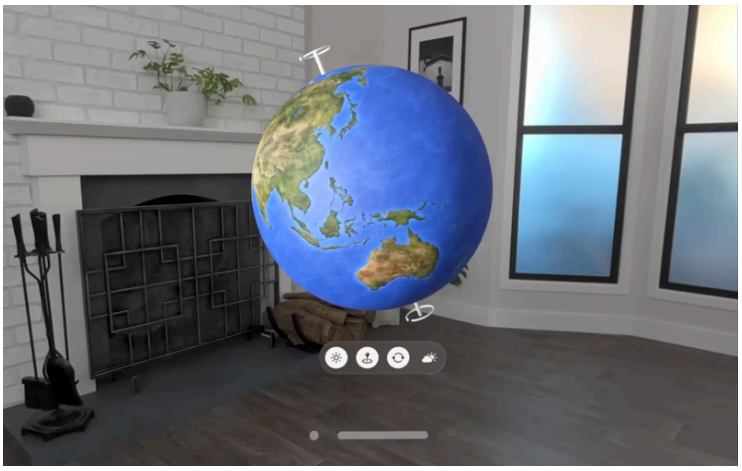
```
<key>UIApplicationSceneManifest</key>
<dict>
    <key>UIApplicationSupportsMultipleScenes</key>
    <true/>
    <key>UISceneConfigurations</key>
    <dict/>
</dict>
```

Declare a volume for the globe

With the key in place, the app makes use of a second `WindowGroup` in its `App` declaration. This new window group uses the `Globe` view as its content:

```
WindowGroup(id: Module.globe.name) {
    Globe()
        .environment(model)
}
.windowStyle(.volumetric)
.defaultSize(width: 0.6, height: 0.6, depth: 0.6, in: .meters)
```

This window group creates a *volume* — which is a container that has three dimensions and behaves like a transparent box — because Hello World uses the `volumetric` window style scene modifier. People can move this box around the Shared Space like they move other window types, and the content remains fixed inside. The `defaultSize(width:height:depth:in:)` modifier specifies a size for the volume in meters, including a depth dimension.

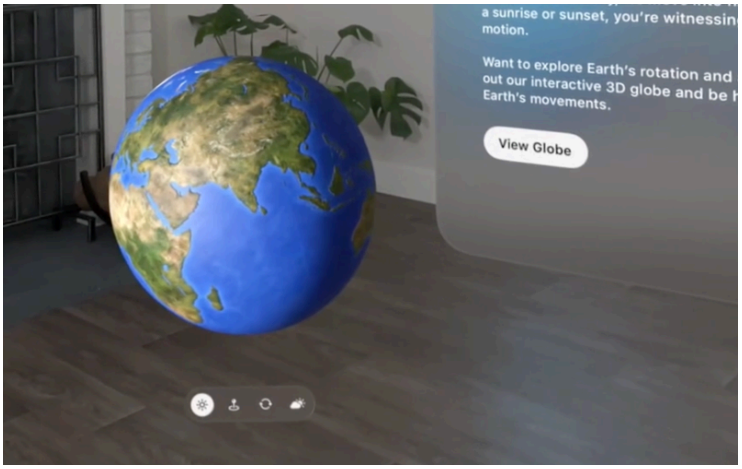


Play

The `Globe` view inside the volume contains 3D content, but is still just a SwiftUI view. It contains two elements in a `ZStack`: a subview that draws a model of the Earth, and another that provides a control panel that people can use to configure the model's appearance.

Open and dismiss the globe volume

The globe module presents a View Globe button that people can tap to display or dismiss the volume, depending on the current state. Hello World achieves this behavior by creating a `Toggle` with the button style, and embedding it in a custom `GlobeToggle` view.

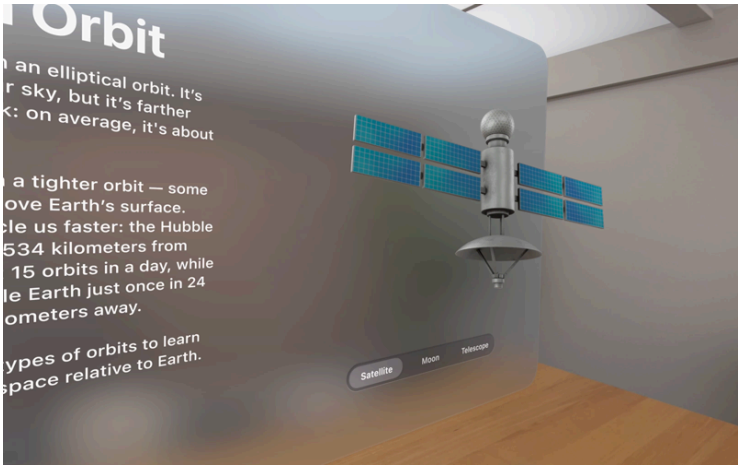



```
struct GlobeToggle: View {
    @Environment(ViewModel.self) private var model
    @Environment(\.openWindow) private var openWindow
    @Environment(\.dismissWindow) private var dismissWindow

    var body: some View {
        @Bindable var model = model

        Toggle(Module.globe.callToAction, isOn: $model.isShowingGlobe)
            .onChange(of: model.isShowingGlobe) { _, isShowing in
                if isShowing {
                    openWindow(id: Module.globe.name)
                } else {
```

```
                    dismissWindow(id: Module.globe.name)
                }
            }
            .toggleStyle(.button)
    }
```

When someone taps the toggle, the `isShowingGlobe` state changes, and the `onChange(of: initial: _:)` modifier calls the `openWindow` or `dismissWindow` action to open or dismiss the volume, respectively. The view gets these actions from the environment and uses an identifier that matches the volume's identifier.

Display objects that orbit the Earth



You use windows in visionOS the same way you do in other platforms. But even 2D windows in visionOS provide a small amount of depth you can use to create 3D effects — like elements that appear in front of other elements. Hello World takes advantage of this depth to present small models inline with 2D content.

The app's second module, Objects in Orbit, provides information about objects that go around the Earth, like the Moon and artificial satellites. To give a sense of what these objects look like, the module displays 3D models of these items directly inside the window.

Hello World loads these models from the asset bundle using a `Model3D` structure inside a custom `ItemView`. The view scales and positions the model to fit the available space, and applies optional orientation adjustments:

```
private struct ItemView: View {
    var item: Item
    var orientation: SIMD3<Double> = .zero

    var body: some View {
        Model3D(named: item.name, bundle: worldAssetsBundle) { model in
            model.resizable()
                .scaledToFit()
                .rotation3DEffect(
```

```
                    Rotation3D(
                        eulerAngles: .init(angles: orientation, order: .xyz)
                    )
                )
                .frame(depth: modelDepth)
                .offset(z: -modelDepth / 2)
        } placeholder: {
            ProgressView()
                .offset(z: -modelDepth * 0.75)
        }
    }
}
```

The app uses this `ItemView` once for each model, placing each in an overlay that only becomes visible based on the current selection. For example, the following overlay displays the satellite model with a small amount of tilt in the x-axis and z-axis:

```
.overlay {
    ItemView(item: .satellite, orientation: [0.15, 0, 0.15])
        .opacity(selection == .satellite ? 1 : 0)
}
```

The `VStack` that contains the models also contains a `Picker` that people use to select a model to view:

```
Picker("Satellite", selection: $selection) {
    ForEach(Item.allCases) { item in
        Text(item.name)
    }
}
.pickerStyle(.segmented)
```

When you add 3D effects to a 2D window, keep this guidance in mind:

- **Don’t overdo it.** These kinds of effects add interest, but can unintentionally obscure important controls or information as people view the window from different directions.
- **Ensure that elements don’t exceed the available depth.** Excess depth causes elements to clip. Account for any position or orientation changes that might occur after initial placement.
- **Avoid models intersecting with the backing glass.** Again, account for potential movement after initial placement.

Show Earth's relationship to its satellites in an immersive space

People can visualize how satellites move around the Earth because the app's orbit module displays the Earth, the Moon, and a communications satellite together as a single system. People can move the system anywhere in their environment or resize it using standard gestures. They can also move themselves around the system to get different perspectives.

> **Note**
>
> To learn about designing with gestures in visionOS, read Gestures in Human Interface Guidelines.

To create this visualization, the app displays the `Orbit` view — which contains a single `RealityView` that models the entire system — in an `ImmersiveSpace` scene with the `mixed` immersion style:

```
ImmersiveSpace(id: Module.orbit.name) {
    Orbit()
        .environment(model)
}
.immersionStyle(selection: $orbitImmersionStyle, in: .mixed)
```

As with any secondary scene in a visionOS app, this scene depends on having the `UIApplicationSupportsMultipleScenes` key in the `Information Property List` file. The app also opens and closes the space using a toggle view that resembles the one used for the globe:

```
struct OrbitToggle: View {
    @Environment(ViewModel.self) private var model
    @Environment(\.openImmersiveSpace) private var openImmersiveSpace
    @Environment(\.dismissImmersiveSpace) private var dismissImmersiveSpace

    var body: some View {
        @Bindable var model = model
```

```
        Toggle(Module.orbit.callToAction, isOn: $model.isShowingOrbit)
            .onChange(of: model.isShowingOrbit) { _, isShowing in
                Task {
                    if isShowing {
                        await openImmersiveSpace(id: Module.orbit.name)
                    } else {
                        await dismissImmersiveSpace()
                    }
                }
            }
            .toggleStyle(.button)
    }
}
```

There are a few key differences from the version that appears in the section Open and dismiss the globe volume:

- `OrbitToggle` uses `openImmersiveSpace` and `dismissImmersiveSpace` from the environment, rather than the window equivalents.
- The dismiss action in this case doesn't require an identifier, because people can only open one space at a time, even across apps.
- The open and dismiss actions for spaces operate asynchronously, and so they appear inside a `Task`.

View the solar system from space using full immersion

The app's final module gives people a sense of the Earth's place in the solar system. Like other modules, this one includes information and a decorative image next to a button that leads to another visualization — in this case so people can experience Earth from space.

When a person taps the button, the app takes over the entire display and shows stars in all directions. The Earth appears directly in front, the Moon to the right, and the Sun to the left. The main window also shows a small control panel that people can use to exit the fully immersive experience.

> **Tip**
>
> People can always close the currently open immersive space by pressing the device's Digital Crown, but it's typically useful when you provide a built-in mechanism to maintain control of the experience within your app.

The app uses another immersive space scene for this module, but here with the `full` immersion style that turns off the passthrough video:

```
ImmersiveSpace(id: Module.solar.name) {
    SolarSystem()
        .environment(model)
}
.immersionStyle(selection: $solarImmersionStyle, in: .full)
```

This scene depends on the same `UIApplicationSupportsMultipleScenes` key that other secondary scenes do, and is activated by a `SolarSystemToggle` that's similar to the ones that the app uses for the other scenes:

```
struct SolarSystemToggle: View {
    @Environment(ViewModel.self) private var model
    @Environment(\.openImmersiveSpace) private var openImmersiveSpace
    @Environment(\.dismissImmersiveSpace) private var dismissImmersiveSpace

    var body: some View {
        Button {
            Task {
                if model.isShowingSolar {
                    await dismissImmersiveSpace()
                } else {
                    await openImmersiveSpace(id: Module.solar.name)
                }
            }
        } label: {
            if model.isShowingSolar {
                Label(
                    "Exit the Solar System",
                    systemImage: "arrow.down.right.and.arrow.up.left")
            } else {
                Text(Module.solar.callToAction)
            }
```

```
        }
    }
}
```

This control appears in the main window to provide a way to begin the fully immersive experience, and separately in the control panel as a way to exit the experience. Because the app uses this control as two distinct buttons rather than as a toggle in one location, it's composed of a `Button` with behavior that changes depending on the app state rather than as a toggle with a button style.

To reuse the main window for the solar system controls, Hello World places both the navigation stack and the controls in a `ZStack`, and then sets the opacity of each to ensure that only one appears at a time:

```
ZStack {
    SolarSystemControls()
        .opacity(model.isShowingSolar ? 1 : 0)

    NavigationStack(path: $model.navigationPath) {
        // ...
    }
    .opacity(model.isShowingSolar ? 0 : 1)
}
.animation(.default, value: model.isShowingSolar)
```

See Also

Related samples

{} Happy Beam

Leverage a Full Space to create a fun game using ARKit.

{} Destination Video

Leverage SwiftUI to build an immersive media experience in a multiplatform app.

{} Diorama

Design scenes for your visionOS app using Reality Composer Pro.

Related articles

Creating your first visionOS app

Build a new visionOS app using SwiftUI and add platform-specific features.

- Adding 3D content to your app

  Add depth and dimension to your visionOS app and discover how to incorporate your app's content into a person's surroundings.

- Creating fully immersive experiences in your app

  Build fully immersive experiences by combining spaces with content you create using RealityKit or Metal.

- Presenting windows and spaces

  Open and close the scenes that make up your app's interface.

- Positioning and sizing windows

  Influence the initial geometry of windows that your app presents.

Related videos

Platforms State of the Union

Meet SwiftUI for spatial computing

Go beyond the window with SwiftUI

Take SwiftUI to the next dimension

Develop your first immersive app

Get started with building apps for spatial computing