Article

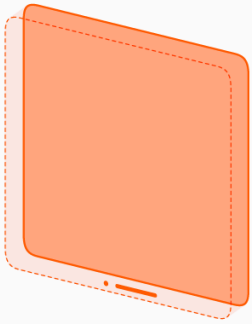
# Adding 3D content to your app

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

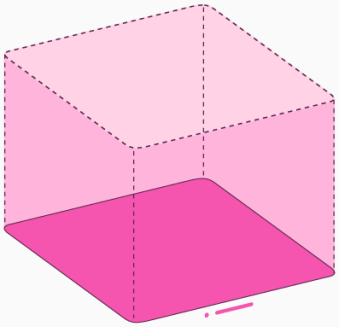
## Overview

A device with a stereoscopic display lets people experience 3D content in a way that feels more real. Content appears to have real depth, and people can view it from different angles, making it seem like it's there in front of them.

When building an app for visionOS, think about ways you might add depth to your app's interface. The system provides several ways to display 3D content, including in your existing windows, in a volume, and in an immersive space. Choose the options that work best for your app and the content you offer.



Window

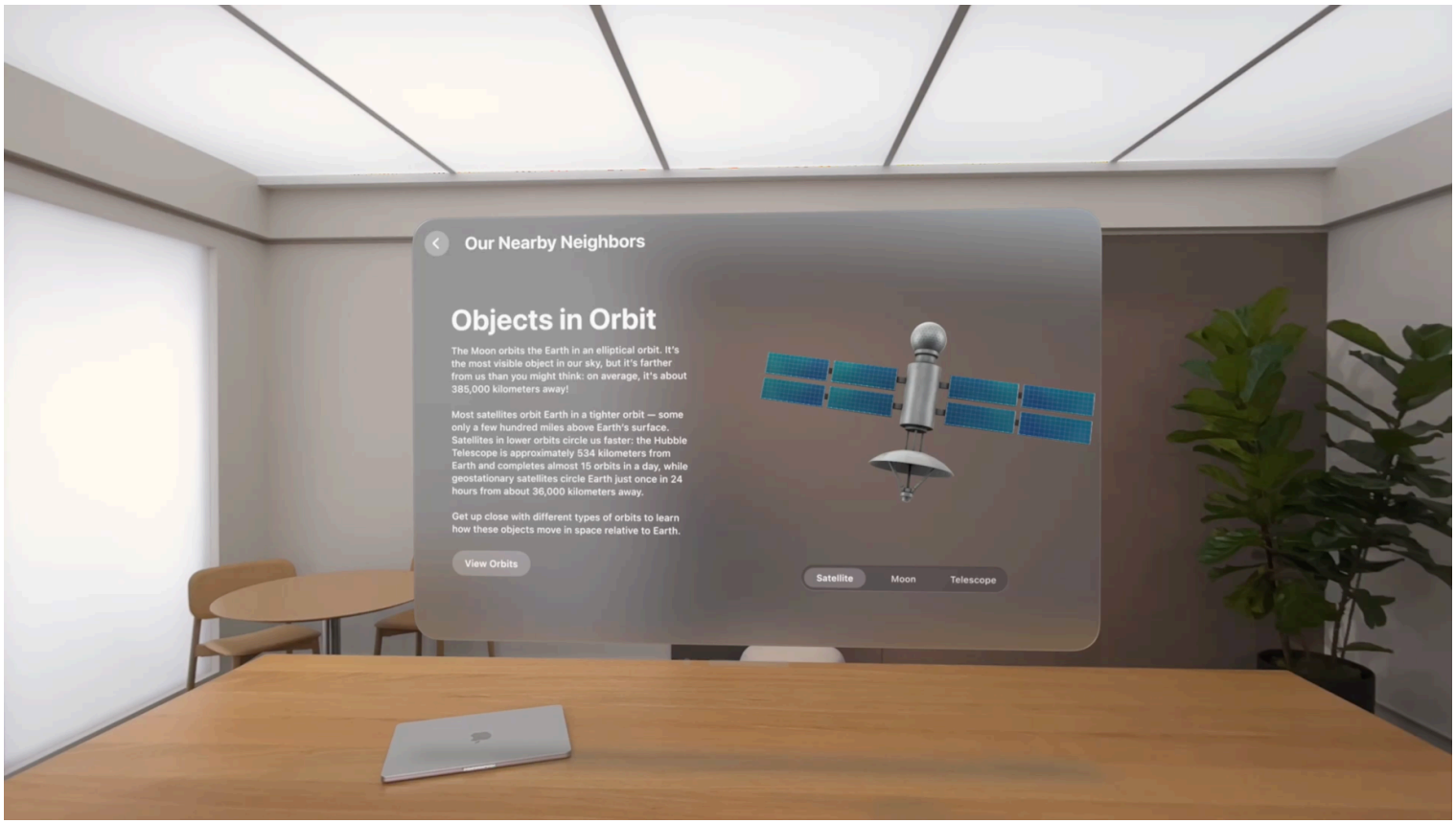


Volume

Immersive space

## Add depth to traditional 2D windows

Windows are an important part of your app's interface. With visionOS, apps automatically get materials with the visionOS look and feel, fully resizable windows with spacing tuned for eyes and hands input, and access to highlighting adjustments for your custom controls.




Play ⊙

Incorporate depth effects into your custom views as needed, and use 3D layout options to arrange views in your windows.

- Apply a `shadow(color:radius:x:y:)` or `visualEffect(_:)` modifier to the view.
- Lift or highlight the view when someone looks at it using a `hoverEffect(_:in:isEnabled:)` modifier.
- Lay out views using a `ZStack`.
- Animate view-related changes with `transform3DEffect(_:)`.
- Rotate the view using a `rotation3DEffect(_:axis:anchor:anchorZ:perspective:)` modifier.

In addition to giving 2D views more depth, you can also add static 3D models to your 2D windows. The `Model3D` view loads a USDZ file or other asset type and displays it at its intrinsic size in your window. Use this in places where you already have the model data in your app, or can download it from the network. For example, a shopping app might use this type of view to display a 3D version of a product.

## Display dynamic 3D scenes using RealityKit

RealityKit is Apple's technology for building 3D models and scenes that you update dynamically onscreen. In visionOS, use RealityKit and SwiftUI together to seamlessly couple your app's 2D and 3D content. Load existing USDZ assets or create scenes in Reality Composer Pro that incorporate

animation, physics, lighting, sounds, and custom behaviors for your content. To use a Reality Composer Pro project in your app, add the Swift package to your Xcode project and import its module in your Swift file. For more information, see Managing files and folders in your Xcode project.


When you're ready to display 3D content in your interface, use a `RealityView`. This SwiftUI view serves as a container for your RealityKit content, and lets you update that content using familiar SwiftUI techniques.

The following example shows a view that uses a `RealityView` to display a 3D sphere. The code in the view's closure creates a RealityKit entity for the sphere, applies a texture to the surface of the sphere, and adds the sphere to the view's content.

```
struct SphereView: View {
    var body: some View {
        RealityView { content in
            let model = ModelEntity(
                        mesh: .generateSphere(radius: 0.1),
                        materials: [SimpleMaterial(color: .white, isMetallic: true)
            content.add(model)
        }
    }
}
```

When SwiftUI displays your `RealityView`, it executes your code once to create the entities and other content. Because creating entities is relatively expensive, the view runs your creation code only once. When you want to update the state of your entities, change the state of your view and use an update closure to apply those changes to your content. The following example uses an update closure to change the size of the sphere when the value in the `scale` property changes:

```
struct SphereView: View {
    var scale = false

    var body: some View {
        RealityView { content in
            let model = ModelEntity(
                        mesh: .generateSphere(radius: 0.1),
                        materials: [SimpleMaterial(color: .white, isMetallic: true)
            content.add(model)
        } update: { content in
            if let model = content.entities.first {
                model.transform.scale = scale ? [1.2, 1.2, 1.2] : [1.0, 1.0, 1.0]
            }
        }
    }
}
```

For information about how to create content using RealityKit, see RealityKit.

## Respond to interactions with RealityKit content

To handle interactions with the entities of your RealityKit scenes:

- Attach a gesture recognizer to your `RealityView` and add the `targetedToAnyEntity()` modifier to it.
- Attach an `InputTargetComponent` to the entity or one of its parent entities.
- Add collision shapes to the RealityKit entities that support interactions.

The `targetedToAnyEntity()` modifier provides a bridge between the gesture recognizer and your RealityKit content. For example, to recognize when someone drags an entity, specify a `DragGesture` and add the modifier to it. When the specified gesture occurs on an entity, SwiftUI executes the provided closure.

The following example adds a tap gesture recognizer to the sphere view from the previous example. The code also adds `InputTargetComponent` and `CollisionComponent`

components to the shape to allow the interactions to occur. If you omit these components, the view doesn't detect the interactions with your entity.

```
struct SphereView: View {
    @State private var scale = false

    var body: some View {
        RealityView { content in
            let model = ModelEntity(
                mesh: .generateSphere(radius: 0.1),
                materials: [SimpleMaterial(color: .white, isMetallic: true)])

            // Enable interactions on the entity.
            model.components.set(InputTargetComponent())
            model.components.set(CollisionComponent(shapes: [.generateSphere(radius:
            content.add(model)
        } update: { content in
            if let model = content.entities.first {
                model.transform.scale = scale ? [1.2, 1.2, 1.2] : [1.0, 1.0, 1.0]
            }
        }
        .gesture(TapGesture().targetedToAnyEntity().onEnded { _ in
            scale.toggle()
        })
    }
}
```

## Display 3D content in a volume

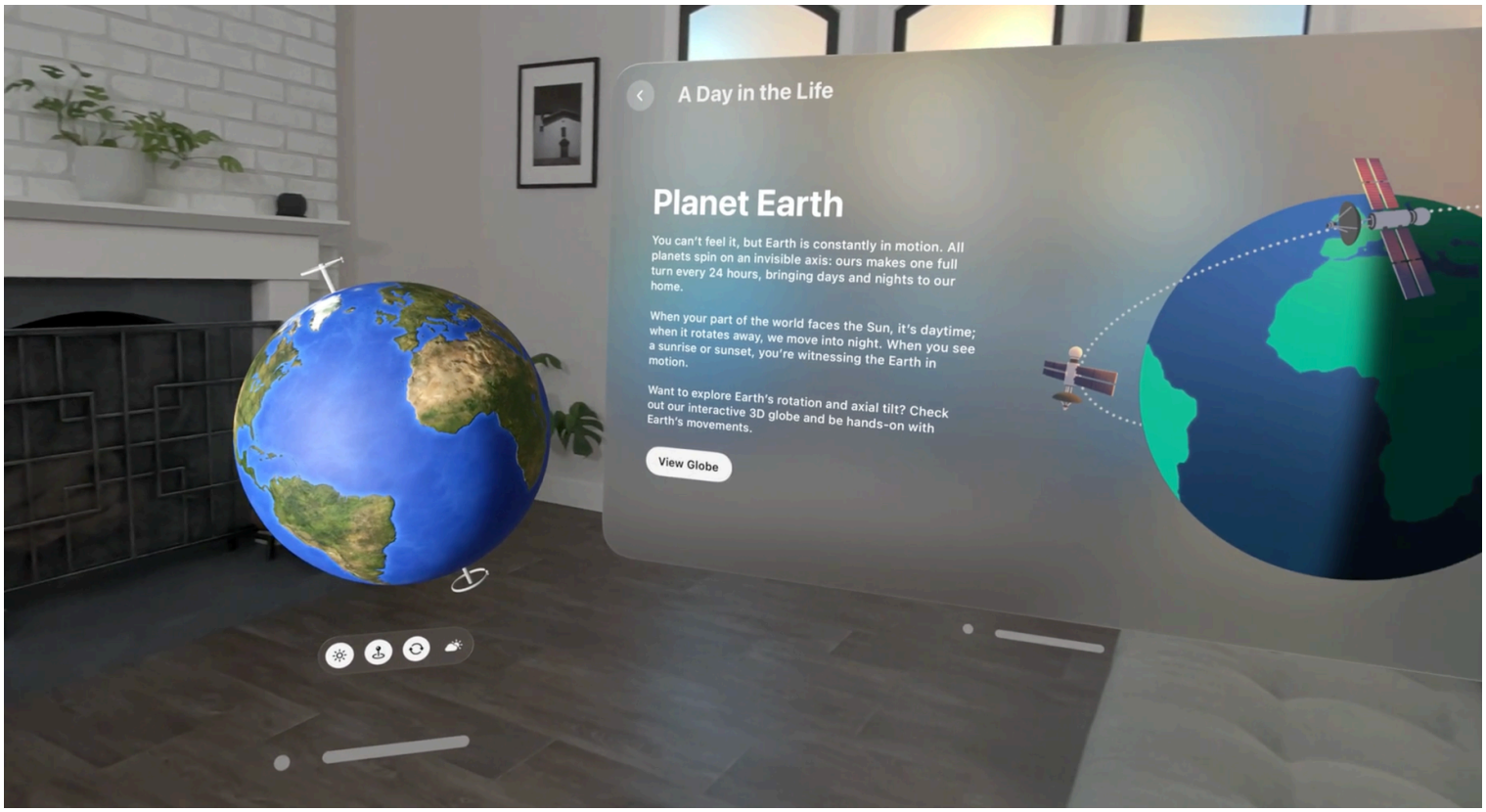
A volume is a type of window that grows in three dimensions to match the size of the content it contains. Windows and volumes both accommodate 2D and 3D content, and are alike in many ways. However, windows clip 3D content that extends too far from the window's surface, so volumes are the better choice for content that is primarily 3D.

To create a volume, add a `WindowGroup` scene to your app and set its style to `volumetric`. This style tells SwiftUI to create a window for 3D content. Include any 2D or 3D views you want in your volume. You can also add a `RealityView` to build your content using RealityKit. The following example creates a volume with a static 3D model of some balloons stored in the app's bundle:

```
struct MyApp: App {
    var body: some Scene {
        WindowGroup {
```

```
            Model3D("balloons")
        }.windowStyle(style: .volumetric)
    }
}
```

Windows and volumes are a convenient way to display bounded 2D and 3D content, but your app doesn't control the placement of that content in the person's surroundings. The system sets the initial position of each window and volume at display time. The system also adds a window bar to allow someone to reposition the window or resize it.




For more information about when to use volumes, see Human Interface Guidelines > Windows.

## Display 3D content in a person's surroundings

When you need more control over the placement of your app's content, add that content to an `ImmersiveSpace`. An immersive space offers an unbounded area for your content, and you control the size and placement of content within the space. After receiving permission from the user, you can also use ARKit with an immersive space to integrate content into their surroundings. For example, you can use ARKit scene reconstruction to obtain a mesh of furniture and nearby objects and have your content interact with that mesh.

An `ImmersiveSpace` is a scene type that you create alongside your app's other scenes. The following example shows an app that contains an immersive space and a window:

```
@main
struct MyImmersiveApp: App {
    var body: some Scene {
        WindowGroup() {
            ContentView()
        }

        ImmersiveSpace(id: "solarSystem") {
            SolarSystemView()
        }
    }
}
```

If you don't add a style modifier to your `ImmersiveSpace` declaration, the system creates that space using the `mixed` style. This style displays your content together with the passthrough content that shows the person's surroundings. Other styles let you hide passthrough to varying degrees. Use the `immersionStyle(selection:in:)` modifier to specify which styles your space supports. If you specify more than one style, you can toggle between the styles using the `selection` parameter of the modifier.

**Warning**

Be mindful of how much content you include in immersive scenes that use the `mixed` style. Content that fills a significant portion of the screen, even if that content is partially transparent, can prevent the person from seeing potential hazards in their surroundings. If you want to immerse the person in your content, configure your space with the `full` style. For more information, see, Creating fully immersive experiences in your app.

Remember to set the position of items you place in an `ImmersiveSpace`. Position SwiftUI views using modifiers, and position a RealityKit entity using its transform component. SwiftUI places the origin of a space at a person's feet initially, but can change this origin in response to other events. For example, the system might shift the origin to accommodate a SharePlay activity that displays your content with Spatial Personas. If you need to position SwiftUI views and RealityKit entities relative to one another, perform any needed coordinate conversions using the methods in the `content` parameter of `RealityView`.

To display your `ImmersiveSpace` scene, open it using the `openImmersiveSpace` action, which you obtain from the SwiftUI environment. This action runs asynchronously and uses the provided information to find and initialize your scene. The following example shows a button that opens the space with the `solarSystem` identifier:

```
Button("Show Solar System") {
    Task {
        let result = await openImmersiveSpace(id: "solarSystem")
        if case .error = result {
            print("An error occurred")
        }
    }
}
```

When an app presents an `ImmersiveSpace`, the system hides the content of other apps to prevent visual conflicts. The other apps remain hidden while your space is visible but return when you dismiss it. If your app defines multiple spaces, you must dismiss the currently visible space before displaying a different space. If you don't dismiss the visible space, the system issues a runtime warning when you try to open the other space.

# See Also

## App construction

- Creating your first visionOS app

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

- Creating fully immersive experiences in your app

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

- Drawing sharp layer-based content in visionOS

  Deliver text and vector images at multiple resolutions from custom Core Animation layers in visionOS.

- Introductory visionOS samples

  Learn the fundamentals of building apps for visionOS with beginner-friendly sample code projects.