Sample Code

Displaying a 3D environment through a portal

Implement a portal window that displays a 3D environment and simulates entering a portal by using RealityKit.

Download

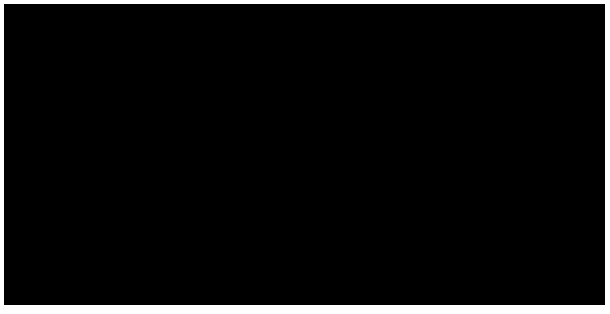
visionOS 2.0+ | Xcode 16.0+

Overview

This sample code project demonstrates how to view a 3D environment through a flat plane with the portal component, and transition into the portal by creating an immersive space. To create a portal, you first need to define an entity and set the `WorldComponent` to store the content of the portal. Next, you create a separate entity to represent the portal's structure, such as the flat plane in the sample. Finally, you combine the two entities, assigning the portal content entity to `PortalComponent` and setting it to the portal structure entity to create the portal.

At launch, the sample app creates the following:

- A portal that displays a box environment.
- A vertical stack that contains a line of text and a button.
- An immersive space that is ready to open.



Play

Set up the 3D environment

The sample creates the 3D environment in two scenarios:

- When creating the portal
- When loading the immersive space

The app uses the `createEnvironment(on:)` method to create the box environment and apply image-based lighting.

```
@MainActor func createEnvironment(on root: Entity) async throws {
    do {
        /// The root entity for the box environment.
        let assetRoot = try await Entity(named: "CornellBox.usda")

        // Convert the image-based lighting file into a URL, and load it as an envi
        guard let iblURL = Bundle.main.url(forResource: "TeapotIBL", withExtension:
            fatalError("Failed to load the Image-Based Lighting file.")
        }
        let iblEnv = try await EnvironmentResource(fromImage: iblURL)

        // ...
    }
}
```

The method starts by loading the box environment from the `.usda` file. Then, it loads an image-based lighting file as an `EnvironmentResource` that contains the background and lighting information as a scene.

To apply image-based lighting to the box environment, the app creates an `ImageBasedLightComponent` and sets it to a placeholder `Entity`. This allows the app to later assign the `ImageBasedLightReceiverComponent` to the box environment to enable the environment with image-based lighting.

```
@MainActor func createEnvironment(on root: Entity) async throws {
    do {
        // ...

        /// The entity to perform image-based lighting on the environment.
        let iblEntity = Entity()
```

```
        /// The image-based lighting component that contains background and lighting
        var iblComp = ImageBasedLightComponent(source: .single(iblEnv))
        iblComp.inheritsRotation = true

        // Add the image-based lighting component to the entity.
        iblEntity.components.set(iblComp)

        // Set up image-based lighting for the box environment.
        assetRoot.components.set(ImageBasedLightReceiverComponent(imageBasedLight: i

        // Add the image-based lighting entity to the box environment.
        assetRoot.addChild(iblEntity)

        // Add the box environment to `root`.
        root.addChild(assetRoot)
    }
}
```

Finally, the app adds the placeholder entity containing the image-based lighting component to the box environment, and then adds the box environment to the root entity.

Create a portal window

To create a portal, the app defines the mesh of the portal by creating a flat plane to act as a window shape for it.

```
/// The root entity for other entities within the scene.
private let root = Entity()

/// A plane entity representing a portal.
private let portalPlane = ModelEntity(
    mesh: .generatePlane(width: 1.0, height: 1.0),
    materials: [PortalMaterial()]
)
```

The `createPortal()` method sets up the portal and adds it to the `root`. First, it creates a `world` entity to store the content within the portal and sets it with `WorldComponent` to separate the `world` entity from the default world, enabling it to be visible only through a portal. Then, it creates the 3D box environment using `createEnvironment(on:)`, and passes in the `world` entity.

```
@MainActor func createPortal() async {
    // Create the entity that stores the content within the portal.
    let world = Entity()

    // Shrink the portal world and update the position
    // to make it fit into the portal view.
    world.scale *= 0.5
    world.position.y -= 0.5
    world.position.z -= 0.5

    // Allow the entity to be visible only through a portal.
    world.components.set(WorldComponent())

    do {
        // Create the box environment and add it to the root.
        try await createEnvironment(on: world)
        root.addChild(world)

        // Set up the portal to show the content in the `world`.
        portalPlane.components.set(PortalComponent(target: world))
        root.addChild(portalPlane)
    } catch {
        fatalError("Failed to create environment: \(error)")
    }
}
```

Finally, it sets the `portalPlane` entity with the `PortalComponent` to transform it into a portal that renders the 3D environment within the `world` entity.

Enter the 3D environment

In the sample app, a person can enter the portal by tapping a button. To achieve this, the app creates an immersive space and loads the 3D box environment in the scene.

```
struct ImmersiveView: View {
    var body: some View {
        RealityView { content in
            // Create the box environment on the root entity.
            let root = Entity()
            do {
                try await createEnvironment(on: root)
```

```
            } catch {
                print("Failed to load environment: \(error.localizedDescription)")
            }

            content.add(root)
        }
    }
}
```

To handle the enter and exit actions of the portal, the sample creates a button to enter or exit the immersive space based on the current state. The button uses `openImmersiveSpace` and `dismissImmersiveSpace` to open and close the immersive space. Because there are multiple paths to close the immersive space and there may be multiple paths to open the immersive space, your app can use the view life-cycle updates to keep track of the current state of the immersive space. Use the `onAppear(perform:)` and `onDisappear(perform:)` on the root view of the immersive space to respond to these changes.

```
ImmersiveSpace(id: appModel.immersiveSpaceID) {
    ImmersiveView()
        .onAppear {
            appModel.immersiveSpaceState = .open
        }
        .onDisappear {
            appModel.immersiveSpaceState = .closed
        }
}
```