Sample Code

Creating an interactive 3D model in visionOS

Display an interactive car model using gestures in a reality view.

Download

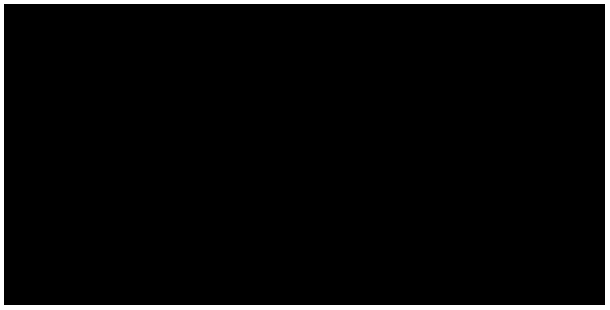
visionOS 2.0+ | Xcode 16.0+

Overview

This sample demonstrates how to create and display a 3D car model that responds to gestures within visionOS. It uses the `ModelEntity` initializer to asynchronously load a model by its filename, and it implements the `DragGesture` protocol on the `ModelEntity`, to interact with the model.

At launch, the app's main view generates the following:

- A window that contains text
- A 3D model of a car that reacts to drag and magnify gestures



Play

Set up the gestures for the entity

To make sure the car locks on the ground, the sample creates an `SIMD3` extension with the `grounded` variable to lock the y-axis to 0 during a drag gesture movement:

```
import SwiftUI

extension SIMD3 where Scalar == Float {
    /// The variable to lock the y-axis value to 0.
    var grounded: SIMD3<Scalar> {
        return .init(x: x, y: 0, z: z)
    }
}
```

The main view creates two state variables to track the initial position and scale of the entity:

```
/// The initial position of the entity.
@State var initialPosition: SIMD3<Float>? = nil

/// The initial scale of the entity.
@State var initialScale: SIMD3<Float>? = nil
```

The app creates a `translationGesture` to update the entity's positions with a person's gesture movements, allowing the person to select and move an entity around the immersive space:

```
var translationGesture: some Gesture {
    DragGesture()
        .targetedToAnyEntity()
        .onChanged({ value in
            /// The entity that the drag gesture targets.
            let rootEntity = value.entity

            // Set `initialPosition` to the initial position of the entity if it is
            if initialPosition == nil {
                initialPosition = rootEntity.position
            }

            /// The movement that converts a global world space to the scene world s
            let movement = value.convert(value.translation3D, from: .global, to: .sc

            // Apply the entity position to match the drag gesture,
            // and set the movement to stay at the ground level.
            rootEntity.position = (initialPosition ?? .zero) + movement.grounded
```

```
        })
        .onEnded({ _ in
            // Reset the `initialPosition` to `nil` when the gesture ends.
            initialPosition = nil
        })
}
```

The `initialPosition` variable resets back to `nil` when the gesture ends.

> **Note**
>
> You must use `targetedToAnyEntity()` or related methods to enable a gesture to target an entity.

The `scaleGesture` applies the scale rate by the magnification on the gesture, to smoothly scale the entity during a gesture movement:

```
var scaleGesture: some Gesture {
    MagnifyGesture()
        .targetedToAnyEntity()
        .onChanged { gesture in
            /// The entity that the magnify gesture targets.
            let rootEntity = value.entity

            // Set the `initialScale` to the initial scale of the entity if it is `n
            if initialScale == nil {
                initialScale = rootEntity.scale
            }

            /// The rate that the model will scale by.
            let scaleRate: Float = 1.0

            // Scale the entity up smoothly by the relative magnification on the ges
            rootEntity.scale = (initialScale ?? .init(repeating: scaleRate)) * Float
        }
        .onEnded { _ in
            // Reset the `initialScale` back to `nil` when the gesture ends.
            initialScale = nil
        }
}
```

The `initialScale` variable resets back to `nil` once the gesture ends.

Load the 3D car model

The `CarView` loads in a USDZ file as a `ModelEntity` instance and creates a `bounds` containing a bounding box of the outer dimensional size of the `car` entity:

```
import SwiftUI
import RealityKit

struct CarView: View {
    // ...

    var body: some View {
        RealityView { content in
            /// The name of the model.
            let fileName: String = "Huracan-EVO-RWD-Spyder-opt-22"

            /// The model that loads from the filename asynchronously.
            guard let car = try? await ModelEntity(named: fileName) else {
                assertionFailure("Failed to load model: \(fileName)")
                return
            }

            /// The visual bounds of the car.
            let bounds = car.visualBounds(relativeTo: nil)

            // ...
        }
    }
}
```

Set the collision component

The app uses the `bounds` property to generate a bounding box with `ShapeResource`, which serves as a collision bound for the `CollisionComponent`. This enables the collision component to interact with the environment:

```
import SwiftUI
import RealityKit
```

```
struct CarView: View {
    // ...

    var body: some View {
        RealityView { content in
            // ...

            /// The visual bounds of the car to show at all times.
            let bounds = car.visualBounds(relativeTo: nil)

            /// The width of the collision box by the size of the model.
            let carWidth: Float = (car.model?.mesh.bounds.max.x)!

            /// The height of the collision box by the size of the model.
            let carHeight: Float = (car.model?.mesh.bounds.max.y)!

            /// The depth of the collision box by the size of the model.
            let carDepth: Float = (car.model?.mesh.bounds.max.z)!

            /// The box around the model of the car for collisions.
            let boxShape = ShapeResource.generateBox(
                width: carWidth,
                height: carHeight,
                depth: carDepth)

            // Add the box shape as a collision component.
            car.components.set(CollisionComponent(shapes: [boxShape]))

            // ...
        }
    }
```

The app also uses the `bounds` property to set the `car` entity's spawn position. It sets this position on the ground, along the z-axis, by the radius of `bounds`:

```
import SwiftUI
import RealityKit

struct CarView: View {
    // ...
```

```
    var body: some View {
        RealityView { content in
            // ...

            // Set the spawn position of the entity on the ground.
            car.position.y -= bounds.min.y

            // Set the spawn position along the z-axis, by the radius of the visual
            car.position.z -= bounds.boundingRadius

            // Add the car model to the `RealityView`.
            content.add(car)
        }
        .gesture(translationGesture)
        .gesture(scaleGesture)
    }
}
```

Finally, the app adds the `car` entity to the `RealityView`.

Run the immersive scene

The sample structure includes an `ImmersiveSpace` entry to the scene to include in the app's environment:

```
import SwiftUI

@main
struct EntryPoint: App {
    var body: some Scene {
        WindowGroup {
            MainView()
        }
        ImmersiveSpace(id: "CarView") {
            CarView()
        }
    }
}
```

The sample's main view uses the `openImmersiveSpace` instance property to call the `ImmersiveSpace` that the app's `EntryPoint` defines:

```swift
import SwiftUI

struct MainView: View {
    /// The environment value to get the instance of the `OpenImmersiveSpaceAction`
    @Environment(\.openImmersiveSpace) var openImmersiveSpace

    var body: some View {
        // Display a line of text and
        // open a new immersive space environment.
        Text("Use gestures to move the car")
        .onAppear {
            Task {
                await openImmersiveSpace(id: "CarView")
            }
        }
    }
}
```

Related samples

{} Creating an immersive space in visionOS

Enhance your visionOS app by adding an immersive space using RealityKit.

{} Transforming RealityKit entities using gestures

Build a RealityKit component to support standard visionOS gestures on any entity.