Sample Code

Placing entities using head and device transform

Query and react to changes in the position and rotation of Apple Vision Pro.

Download

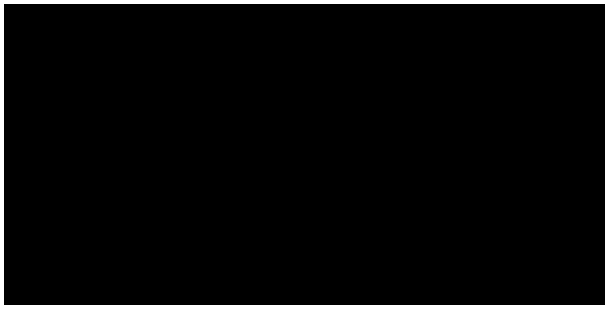
visionOS 2.0+ | Xcode 16.0+

Overview

This sample code project demonstrates how to create and display content that appears at a person's head location, and follows a person's view as they move their head in immersive spaces. It uses `AnchorEntity` and `queryDeviceAnchor(atTimestamp:)` to get the transform of the person's head and Apple Vision Pro to place content relative to them.

This sample creates the following two views and allows you to toggle between them:

- A hummingbird and a feeder directly in front of the person wearing the device.
- A hummingbird that flies to stay in the view of the person wearing the device.



Play

The sample code project uses RealityKit and ARKit, respectively, to position the entities relative to the person. You can run the sample app in either Simulator or on-device.

> **Note**
>
> See Design considerations for vision and motion and Motion in the Human Interface Guidelines for guidance on continuously head-tracked entities.

Show entities at a person's head position

To launch the hummingbird feeder at the position of the wearer's head, the sample uses `AnchorEntity` with the anchoring target of `AnchoringComponent.Target.head`. This target provides the center of the wearer's head, rather than the position of the device itself. You can only use `AnchorEntity` in an immersive space. Although it allows you to anchor content to the wearer's head, you can't access its transform because there's no authorization required. If you attempt to access the transform, the property returns the identity transform instead.

> **Note**
>
> You can get the transform of an `AnchorEntity` with a different `AnchoringComponent.Target`, such as a hand, by using a `SpatialTrackingSession` and requesting authorization from the person using the app.

The sample creates an `AnchorEntity` that anchors to the wearer's head, and sets the `AnchoringComponent.TrackingMode` to `once` to stop tracking after the initial anchor. The head-positioned entity root contains both the feeder entity and the hummingbird entity, which the sample loads from Reality Composer Pro. The app adds the root entity as a subentity of the head anchor to track it. The sample then offsets the feeder from the center of the wearer's head by setting the position.

```
func startHeadPositionMode(content: RealityViewContent) {
    // Reset the rotation so it aligns with the feeder.
    hummingbird.transform.rotation = simd_quatf()

    // Create an anchor for the head and set the tracking mode to `.once`.
    let headAnchor = AnchorEntity(.head)
    headAnchor.anchoring.trackingMode = .once
    headAnchor.name = "headAnchor"
    // Add the `AnchorEntity` to the scene.
    headAnchorRoot.addChild(headAnchor)

    // Add the feeder as a subentity of the root containing the head-positioned enti
    headPositionedEntitiesRoot.addChild(feeder)
```

```
    // Add the hummingbird to the root containing the head-positioned entities and s
    headPositionedEntitiesRoot.addChild(hummingbird)
    hummingbird.setPosition([0, 0, -0.15], relativeTo: headPositionedEntitiesRoot)

    // Add the head-positioned entities to the anchor, and set the position to be in
    headAnchor.addChild(headPositionedEntitiesRoot)
    headPositionedEntitiesRoot.setPosition([0, 0, -0.6], relativeTo: headAnchor)
}
```

Move entities relative to device transform

This sample contains a hummingbird that reacts to the wearer while they move around. It achieves this by creating a `System` and using `queryDeviceAnchor` to update the entities in the scene with each scene update.

You can only use `queryDeviceAnchor` in an immersive space, but it doesn't require authorization.

> **Note**
>
> `queryDeviceAnchor` gives you the transform of the device, not the wearer's head. If you want to get the visual transform of the center of the wearer's head, use `AnchorEntity(.head)`.

The sample starts by creating a RealityKit system, which allows you to update the entities with each scene update. See Implementing systems for entities in a scene for information on creating a system class and using components to query entities. In the system, the app creates a query for entities with the `FollowComponent` `WorldTrackingProvider` and an `ARKitSession` as follows:

```
public struct FollowSystem: System {
    static let query = EntityQuery(where: .has(FollowComponent.self))
    private let arkitSession = ARKitSession()
    private let worldTrackingProvider = WorldTrackingProvider()
    //...
}
```

Then, the sample starts the session by using the `ARKitSession` to run the `WorldTrackingProvider`.

```
public struct FollowSystem: System {
    //...

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

    func startSession() {
        Task {
            do {
                try await arkitSession.run([worldTrackingProvider])
            } catch {
                print("Error: \(error)")
            }
        }
    }

    //...
}
```

The sample adds a custom `Component` named `FollowComponent` to the root entity of the hummingbird entity, and then uses it to query the entities in the scene to apply the movement to.

Important

Make sure to register both the system and the component.

The following example shows how to query the device anchor and move the entity accordingly:

```
public struct FollowSystem: System {
    //...

    public func update(context: SceneUpdateContext) {
        // Check whether the world-tracking provider is running.
        guard worldTrackingProvider.state == .running else { return }

        // Query the device anchor at the current time.
        guard let deviceAnchor = worldTrackingProvider.queryDeviceAnchor(atTimestamp

        // Find the transform of the device.
        let deviceTransform = Transform(matrix: deviceAnchor.originFromAnchorTransfo
```

```
        // Iterate through each entity in the scene containing `FollowComponent`.
        let entities = context.entities(matching: Self.query, updatingSystemWhen: .r

        for entity in entities {
            // Move the entity to the device's transform.
            entity.move(to: deviceTransform, relativeTo: entity.parent, duration: 1.
        }
    }
}
```

The sample keeps the hummingbird at the top right of the wearer's field of vision by setting the hummingbird's position relative to its root entity and offsetting it on the y and z axes.

```
func startFollowMode() {
    //...

    // Set the hummingbird as a subentity of its root, and move it to the top-right
    followRoot.addChild(hummingbird)
    hummingbird.setPosition([0.4, 0.2, -1], relativeTo: followRoot)
}
```

See Also

RealityKit and Reality Composer Pro

- Reality Composer Pro

  Build, create, and design 3D content for your RealityKit apps.

- {} Petite Asteroids: Building a volumetric visionOS game

  Use the latest RealityKit APIs to create a beautiful video game for visionOS.

- {} BOT-anist

  Build a multiplatform app that uses windows, volumes, and animations to create a robot botanist's greenhouse.

- {} Swift Splash

  Use RealityKit to create an interactive ride in visionOS.

- {} Diorama

Design scenes for your visionOS app using Reality Composer Pro.

{} Building an immersive media viewing experience

Add a deeper level of immersion to media playback in your app with RealityKit and Reality Composer Pro.

{} Enabling video reflections in an immersive environment

Create a more immersive experience by adding video reflections in a custom environment.

{} Combining 2D and 3D views in an immersive app

Use attachments to place 2D content relative to 3D content in your visionOS app.

Understanding the modular architecture of RealityKit

Learn how everything fits together in RealityKit.

Using transforms to move, scale, and rotate entities

Learn how to use Transforms to move, scale, and rotate entities in RealityKit.

Capturing screenshots and video from Apple Vision Pro for 2D viewing

Create screenshots and record high-quality video of your visionOS app and its surroundings for app previews.

Implementing object tracking in your visionOS app

Create engaging interactions by training models to recognize and track real-world objects in your app.