

## Documentation

[visionOS](#) / [Introductory visionOS samples](#) / Displaying an entity that follows a person's view

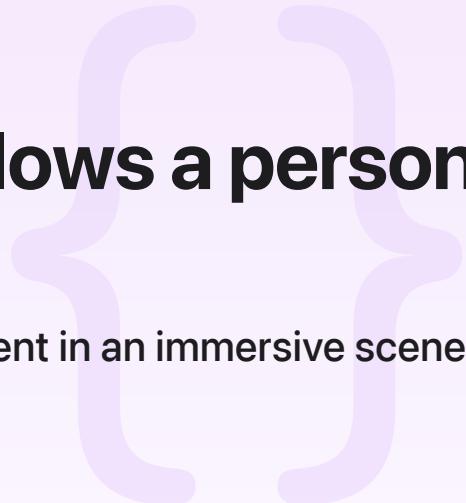
Sample Code

# Displaying an entity that follows a person's view

Create an entity that tracks and follows head movement in an immersive scene.

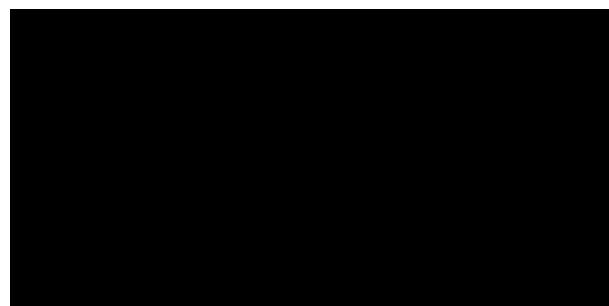
[Download](#)

visionOS 2.0+ | Xcode 16.0+



## Overview

This sample uses world-tracking data from [ARKit](#) in visionOS to create and display a 3D entity that dynamically moves in front of a person's view. As the following video shows, the floating sphere's position updates based on the person's head movement, to ensure the object stays visible and smoothly follows their view:



Play 

## Extend the floats to enable calculations

The sample adds functionality to existing class types by extending `SIMD3<Float>`, `SIMD4<Float>`, and `simd_float4x4`:

```
import Foundation
import simd
import RealityKit

/// The type alias to create a new name for `SIMD3<Float>`.
typealias Float3 = SIMD3<Float>

/// The type alias to create a new name for `SIMD4<Float>`.
typealias Float4 = SIMD4<Float>

/// The type alias to create a new name for `simd_float4x4`.
typealias Float4x4 = simd_float4x4
```

To include these data types in the extension to their associated class, the sample associates each of the entry points using a `typealias`.

The `Float3` extension includes the following methods:

- `init(_:)`, to create a `Float3` from a `Float4`
- `length()`, to calculate the total length of the `Float3`
- `normalized()`, to calculate the normalized vector of the `Float3`

```
import Foundation
import simd
import RealityKit

typealias Float3 = SIMD3<Float>

// ...

extension Float3 {
    /// The initializer of a `Float3` from a `Float4`.
    init(_ float4: Float4) {
        self.init()

        x = float4.x
        y = float4.y
        z = float4.z
    }

    // Calculate the total length by taking the square root of the product of the p
    func length() -> Float {
```

```

        sqrt(x * x + y * y + z * z)
    }

    // Calculate the normalized vector of the float.
    func normalized() -> Float3 {
        self * 1 / length()
    }
}

```

The `Float4` extension contains the `toFloat3()` method that converts a `Float4` value to `Float3`:

```

import Foundation
import simd
import RealityKit

typealias Float4 = SIMD4<Float>

// ...

extension Float4 {
    // Ignore the W value to convert a `Float4` into a `Float3`.
    func toFloat3() -> Float3 {
        Float3(self)
    }
}

```

The `Float4x4` extension includes the following methods:

- `translation()`, to get the transform information in the form of a `Float3`
- `forward()`, to get the forward-facing vector

```

import Foundation
import simd
import RealityKit

typealias Float4x4 = simd_float4x4

// ...

extension Float4x4 {

```

```

// Identify the translation value from the `float4x4` and convert to a `Float3`.
func translation() -> Float3 {
    columns.3.toFloat3()
}

// Identify the forward-facing vector and return a `Float3`.
func forward() -> Float3 {
    columns.2.toFloat3().normalized()
}

}

```

## Update the entities over time

The sample sets up a custom system and component to handle updates in real time:

```

import SwiftUI
import RealityKit

struct ClosureComponent: Component {
    /// The closure that takes the time interval since the last update.
    let closure: (TimeInterval) -> Void

    init(closure: @escaping (TimeInterval) -> Void) {
        self.closure = closure
        ClosureSystem.registerSystem()
    }
}

```

The component contains the `closure` variable to track the time. On initialization, it registers `ClosureSystem` into the reality view.

The `ClosureSystem` constructs a query using the [EntityQuery](#) to retrieve all entities with the `ClosureComponent` from the scene. Then it passes the delta time, which is the elapsed time since the last update, to the `closure` variable for each entity:

```

import SwiftUI
import RealityKit

struct ClosureSystem: System {
    /// The query to check if the entity has the `ClosureComponent`.
    static let query = EntityQuery(where: .has(ClosureComponent.self))
}

```

```

init(scene: RealityKit.Scene) {}

/// Update entities with `ClosureComponent` at each render frame.
func update(context: SceneUpdateContext) {
    for entity in context.entities(matching: Self.query, updatingSystemWhen: .rendering) {
        guard let comp = entity.components[ClosureComponent.self] else { continue }
        comp.closure(context.deltaTime)
    }
}

```

## Implement head tracking

visionOS supports [WorldTrackingProvider](#) from ARKit to get live data about a device's position. World tracking requires an [ARKitSession](#) and a device that supports world tracking. The sample uses the [HeadPositionTracker](#) to initialize the ARKit session and the [WorldTrackingProvider](#):

```

import SwiftUI
import RealityKit
import ARKit

class HeadPositionTracker: ObservableObject {
    /// The instance of the `ARKitSession` for world tracking.
    let arSession = ARKitSession()

    /// The instance of a new `WorldTrackingProvider` for world tracking.
    let worldTracking = WorldTrackingProvider()

    init() {
        Task {
            // Check whether the device supports world tracking.
            guard WorldTrackingProvider.isSupported else {
                print("WorldTrackingProvider is not supported on this device")
                return
            }
            do {
                // Attempt to start an ARKit session with the world-tracking provider.
                try await arSession.run([worldTracking])
            } catch let error as ARKitSession.Error {
                // Handle any potential ARKit session errors.
                print("Encountered an error while running providers: \(error.localizedDescription)")
            }
        }
    }
}

```

```

        } catch let error {
            // Handle any unexpected errors.
            print("Encountered an unexpected error: \(error.localizedDescription)")
        }
    }
}

```

The HeadPositionTracker contains the `originFromDeviceTransform()` method to get the device's transform in real time:

```

func originFromDeviceTransform() -> SIMD_float4x4? {
    /// The anchor of the device at the current time.
    guard let deviceAnchor = worldTracking.queryDeviceAnchor(atTimestamp: CACurrentTime)
        return nil
    }

    // Return the device's transform.
    return deviceAnchor.originFromAnchorTransform
}

```

## Display the sphere that follows the view

Device tracking is accessible within immersive spaces. The sample creates a custom view that uses a reality view to place a 3D sphere in front of the device's forward direction at a set distance.

### Note

Device-tracking data isn't available in visionOS apps that only display a SwiftUI window view or a SwiftUI volumetric view.

```

import SwiftUI
import RealityKit

struct HeadPositionView: View {
    /// The tracker that contains the logic to handle real-time transformations from
    @StateObject var headTracker = HeadPositionTracker()

    var body: some View {
        RealityView(make: { content in
            /// The entity representation of the world origin.

```

```

let root = Entity()

    /// The size of the floating sphere.
let radius: Float = 0.02

    /// The material for the floating sphere.
let material = SimpleMaterial(color: .cyan, isMetallic: false)

    /// The sphere mesh entity.
let floatingSphere = ModelEntity(
    mesh: .generateSphere(radius: radius),
    materials: [material]
)

// Add the floating sphere to the root.
root.addChild(floatingSphere)

// ...
}

}

```

The view creates two entities: the `root` and the `floatingSphere`.

The view sets the `ClosureComponent` to the `root`, creates the `currentTransform` property to determine the headset's current location, and calculates a smooth target position for the floating sphere in front of the device:

```

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

        /// The distance that the content extends out from the device.
        let distance: Float = 1.0

        root.components.set(ClosureComponent(closure: { deltaTime in
            /// The current position of the device.
            guard let currentTransform = headTracker.originFromDeviceTransform() else {
                return
            }

            /// The target position in front of the device.
            let targetPosition = currentTransform.translation() - distance * current
        })
    }
}
```

```
    /// The interpolation ratio for smooth movement.  
    let ratio = Float(pow(0.96, deltaTime / (16 * 1E-3)))  
  
    /// The new position of the floating sphere.  
    let newPosition = ratio * floatingSphere.position(relativeTo: nil) + (1  
  
        // Update the position of the floating sphere.  
        floatingSphere.setPosition(newPosition, relativeTo: nil)  
    ))  
  
    // Add the root entity to the `RealityView`.  
    content.add(root)  
, update: { _ in })
```

The `setPosition()` method moves the sphere to the new position over a set rate of time, applying a smoothing effect to the sphere.

## See Also

### Integrating ARKit

- { } **Creating a 3D painting space**  
Implement a painting canvas entity, and update its mesh to represent a stroke.
- { } **Tracking and visualizing hand movement**  
Use hand-tracking anchors to display a visual representation of hand transforms in visionOS.
- { } **Applying mesh to real-world surroundings**  
Add a layer of mesh to objects in the real world, using scene reconstruction in ARKit.
- { } **Obscuring virtual items in a scene behind real-world items**  
Increase the realism of an immersive experience by adding entities with invisible materials real-world objects.