Sample Code

Tracking accessories in volumetric windows

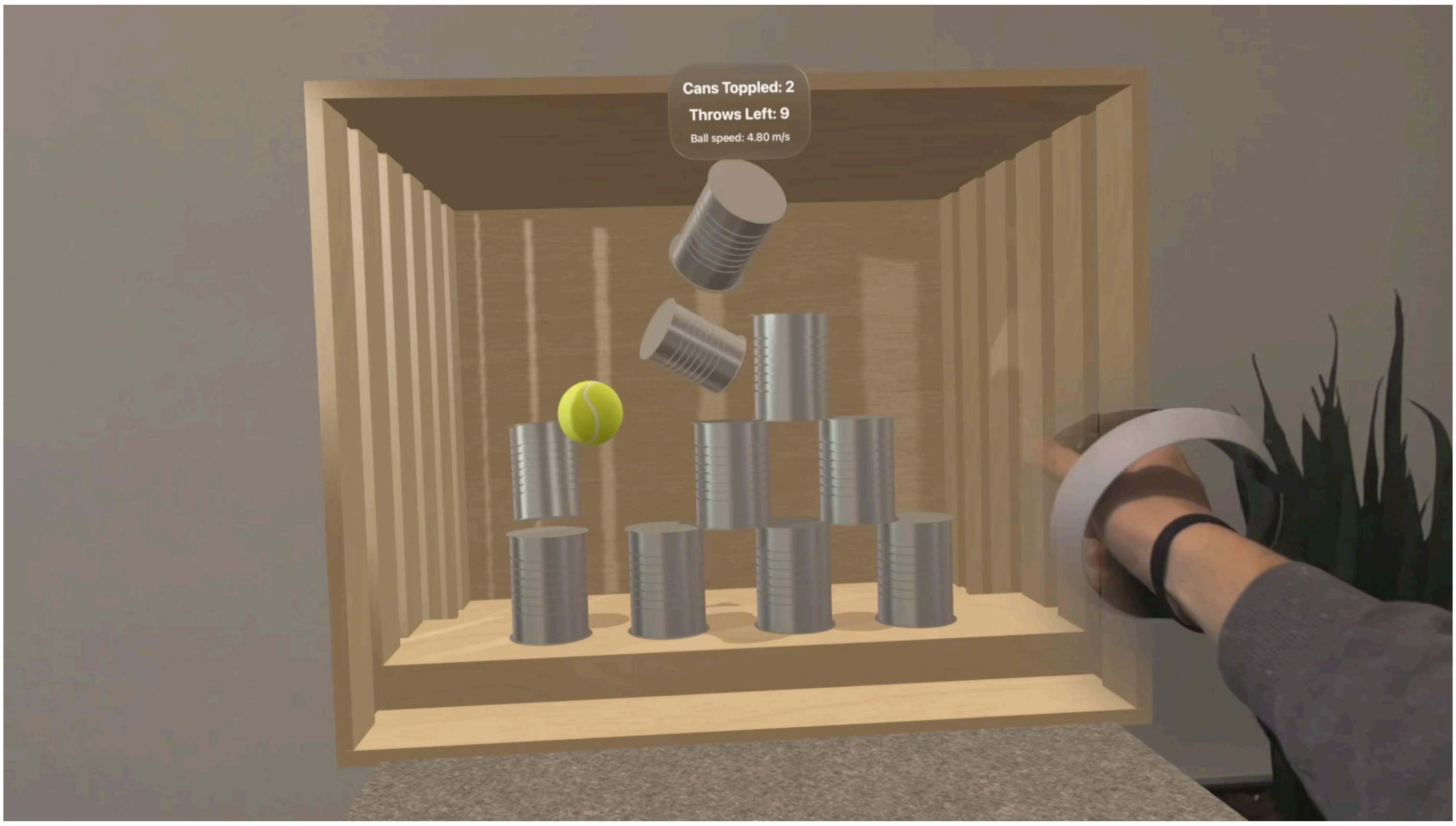
Translate the position and velocity of tracked handheld accessories to throw virtual balls at a stack of cans.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

Accessories enhance the experience of Apple Vision Pro, and offer added functionality and flexibility by supporting fine control, novel input methods, and custom experiences. You can use ARKit to locate and track accessories, converting their real-world motion into app-accessible data. Specifically, ARKit supports high-frequency tracking of an accessory's position and orientation, which it derives velocity and angular velocity information from. Additionally, ARKit provides methods to convert the real-world tracking information to the relevant coordinate spaces in your app.



Play ⊙

> **Note**
>
> Only a small subset of people may have trackable accessories.

Some experiences may require accessories, but most let people perform tasks with both controllers and hands.

This sample code project creates a carnival-like experience with a set of stacked cans in a crate. The cans have physics bodies and are subject to gravity, causing them to fall when a thrown tennis ball strikes them. The crate fills the volume that contains it, and the app tracks accessories when it locates them within the volume. The sample uses tracking information to place a virtual tennis ball model over an onscreen accessory. A quick, tossing motion throws the ball, knocking down the cans, and a rapid set of clockwise and counterclockwise rotations sets them up again.

Set up the sample

The sample adds an `NSAccessoryTrackingUsageDescription` to the information property list file with a description of how the app uses the tracking information. Additionally, a `GCSupportedGameControllers` entry with a `SpatialGamepad` type is required for the Game Controller framework to return controllers when requested. To track the controllers, the app needs to listen for accessories the system adds and removes by observing the `GCControllerDidConnect` and `GCControllerDidDisconnect` notifications of the GameController framework.

The notification object is the `GCController` that's changing connection state. The sample first checks `isSupported` to determine if accessory tracking is available. If so, the app monitors session events to update the internal state based on the data provider state.

> **Note**
>
> Simulator doesn't support accessory tracking, so you need a physical device to run the sample app.

```
if !AccessoryTrackingProvider.isSupported {
    state = .accessoryTrackingNotSupported
    return
}

// Listen for connected and disconnected controllers.
NotificationCenter.default.addObserver(forName: NSNotification.Name.GCControlle
                                       object: nil,
                                       queue: nil) { notification in
    if let controller = notification.object as? GCController {
        guard controller.productCategory == GCProductCategorySpatialController e
            return
        }

        //...
    }
}

NotificationCenter.default.addObserver(forName: NSNotification.Name.GCControlle
                                       object: nil,
                                       queue: nil) { notification in
    if let controller = notification.object as? GCController {
        if controller.productCategory == GCProductCategorySpatialController {
            //...
        }
    }
}
```

An `ARKitSession` running `AccessoryTrackingProvider` implicitly requests authorization. Your app can handle this with an `ARKitSession.Event.authorizationChanged(type:status:)` session event. If the player authorizes the tracking, the code in the `authorizationChanged` handler starts controller tracking.

```swift
private func monitorARKitSessionEvents() async {
    for await event in arkitSession.events {
        switch event {
            case .dataProviderStateChanged(_, let newState, let error):
                if newState == .stopped {
                    if let error {
                        print("An error occurred: \(error)")
                        state = .arkitSessionError
                    }
                }
        case .authorizationChanged(let type, let authorizationStatus):
            if type == .accessoryTracking {
                if authorizationStatus == .denied {
                    state = .accessoryTrackingNotAuthorized
                } else if authorizationStatus == .allowed {
                    state = .startingUp
                    // ...
                }
            }
        default:
            break
        }
    }
}
```

Track accessories

Within its tracking code, the sample requests all available controllers with `controllers()`. Create a trackable ARKit `Accessory` object from the returned `GCController` device. Passing the available accessories to the `AccessoryTrackingProvider` allows the sample to access `Anchor` updates when running in an `ARSession` object. Accessory events are available asynchronously from `anchorUpdates`. During tracking, the sample performs several operations — verifying the controllers presence within the volume, syncing the tennis ball position with the controllers, and checking for the player performing a throw or shake action.

```swift
let accessoryTracking = AccessoryTrackingProvider(accessories: accessories)

do {
    try await arkitSession.run([accessoryTracking])
    state = .inGame
    gameState = .startNewGame
} catch {
    return
}

for await update in accessoryTracking.anchorUpdates {
    process(update)
}
```

The system uses the RealityView update closure to verify that the controllers are located within the volume, and the sample generates a bounding box that the volume determines. If at least one controller is connected and located within the volume bounds, the app state updates accordingly. If all controllers exist outside the bounds, an Out of Bounds message displays on the volume's toolbar.

For each tracked accessory, the app generates a tennis ball entity, and repositions it while handling accessory-tracking anchor updates. The transform of AccessoryAnchor is relative to `WorldReferenceCoordinateSpace`. The app contains the tennis ball model within a `RealityView`, in a volume, unaligned with the world reference coordinate space. It's a complex process to convert the tracked accessory position to the placement of the tennis ball. The sample determines whether the accessory is inside the volume using the anchor's `coordinateSpace(for:correction:)` method to eliminate the complexity. The tennis ball entity doesn't render when the accessories move outside the volume.

```
let aimPoint = controllerAnchor.coordinateSpace(for: .aim, correction: .rendered)

if let realityViewFromAimPointTransform = try? realityViewOrigin.transform(from: ain
    let aimPointPosition = realityViewFromAimPointTransform.matrix.columns.3.xyz
    isInsideRealityView = realityViewEdges.contains(aimPointPosition)
}
```

Create throw and reset gestures

During anchor update processing, the sample handles the tennis ball throw and shake to reset action checking. The app triggers a throw by tracking the peak velocity of the accessory, and determining when the current velocity decreases by 0.6 m/s. The app provides the accessory's velocity as a 3D vector in the accessory anchor's local coordinate space. To obtain the correct velocity, the app transforms the vector relative to the `gameRoot` coordinate space with the `convert(value:, to:)` method. To strike the cans, the ball associated with the tossing accessory anchor sustains a velocity matching that of the anchor. If the system doesn't register a throw within 1 second, it resets the throw tracking.

```
guard let anchor = controller.anchor else { return }

let controllerSpeed = length(anchor.velocity)
controller.pendingThrow.peakSpeed = max(controller.pendingThrow.peakSpeed, controlle

if controller.pendingThrow.peakSpeed > 1.2 &&
    controllerSpeed < controller.pendingThrow.peakSpeed - 0.6 {
    // Trigger a throw if:
    // The controller's peak speed is more than 1.2 m/s.
    // The controller's speed drops more than 0.6 m/s below the peak.
    if controller.triggeredThrow == nil {
        controller.pendingThrow.anchor = anchor

        controller.triggeredThrow = controller.pendingThrow
        controller.pendingThrow = Throw()

        Task {
            // Allow the next throw after 1 second.
            try? await Task.sleep(for: .milliseconds(1000))
            controller.triggeredThrow = nil
        }
    }
}
```

The app triggers a reset by rotating the accessory quickly clockwise and counterclockwise around the z-axis. The anchor's `angularVelocity` property provides the current rate of rotation.

```
guard let anchor = controller.anchor else { return }

let controllerZAngularVelocity = anchor.angularVelocity[2]
controller.pendingShake.peakAngularVelocity = max(controller.pendingShake.peakAngula
```

Checking for positive and negative angular velocities of 90 deg/s, the sample increases the shake count on each change of direction.

```
let halfPi: Float = .pi / 2

if controllerZAngularVelocity < controller.pendingShake.peakAngularVelocity - halfPi
    abs(anchor.angularVelocity[0]) < halfPi && abs(anchor.angularVelocity[1]) < half
    // Detect a controller oscillation on the z-axis if:
    // The controller's angular velocity on the z-axis drops more than 90 deg/s belo
    // The controller's angular velocity on the other axes is less than 90 deg/s.
    let controllerPosition: SIMD3<Float> = anchor.originFromAnchorTransform.columns.

    // Reset the shake if the user moves too much.
    if let shakePrevPos = controller.pendingShake.initialPosition {
        guard length(controllerPosition - shakePrevPos) < 0.2 else {
            controller.pendingShake = Shake()
            return
        }
    }

    if controllerZAngularVelocity < -halfPi {
        if controller.pendingShake.currentDirection == .counterClockwise {
            controller.pendingShake.oscillationCount += 1
        }
        controller.pendingShake.currentDirection = .clockwise
    } else if controllerZAngularVelocity > halfPi {
        if controller.pendingShake.currentDirection == .clockwise {
            controller.pendingShake.oscillationCount += 1
        }
        controller.pendingShake.currentDirection = .counterClockwise
    }

    if controller.pendingShake.oscillationCount == 1 {
        controller.pendingShake.initialPosition = controllerPosition
    }
```

If the shake direction changes six times, the app performs the action and resets the cans into a stack, ready for the next game.

```
    if controller.triggeredShake == nil && controller.pendingShake.oscillationCount
        controller.triggeredShake = controller.pendingShake
        controller.pendingShake = Shake()

        gameState = .startNewGame

        Task {
            // Reset the triggered shake after 0.5 seconds.
            try? await Task.sleep(for: .milliseconds(500))
            controller.triggeredShake = nil
        }
    }
```

See Also

Accessory tracking

`class AccessoryTrackingProvider`

Provides the real time position of accessories in the user's environment.

`struct Accessory`

Represents an accessory to be tracked.

`struct AccessoryAnchor`

Represents a tracked accessory.

{} Tracking a handheld accessory as a virtual sculpting tool

Use a tracked accessory with Apple Vision Pro to create a virtual sculpture.