Sample Code

Tracking and visualizing hand movement

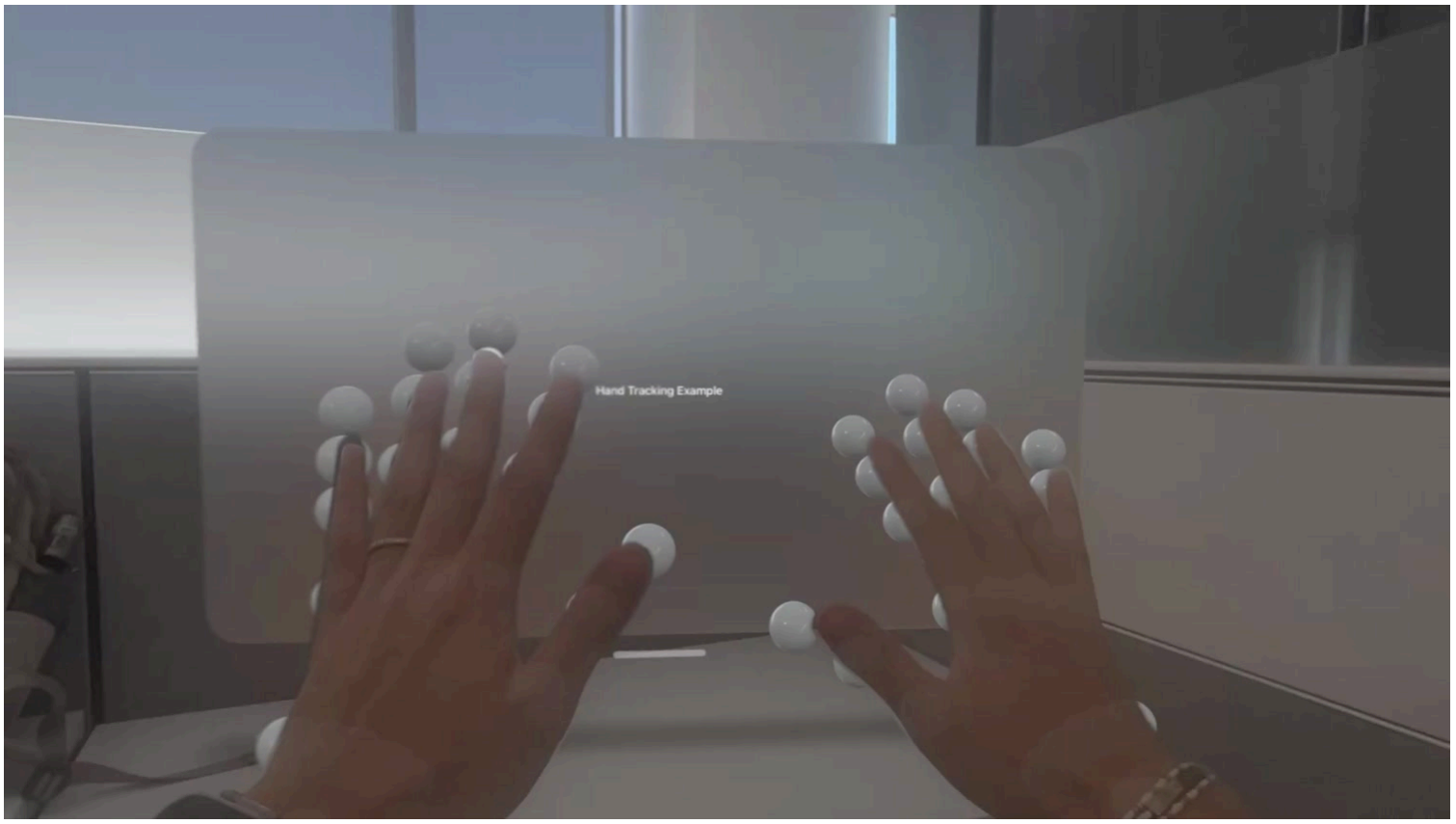
Use hand-tracking anchors to display a visual representation of hand transforms in visionOS.

Download

visionOS 2.0+ | Xcode 16.0+

Overview

This sample demonstrates tracking hand transforms in visionOS with the `HandTrackingProvider` class, a type within ARKit. As the following video shows, the app displays a series of white spheres that affix themselves to each anchor point on a person's hands and remain attached as the hands move:



Play ⊙

The app achieves this effect by:

1. Creating a hand entity
2. Placing white spheres on each hand-tracking anchor
3. Tracking the person's hand to update the hand entity's position and orientation

Define the hand-tracking anchors

`HandSkeleton.JointName` contains a total of 26 hand-tracking anchors for each hand. To identify these anchors, the sample creates a series of `Finger` and `Bone` enumerations:

```
enum Finger: Int, CaseIterable {
    case forearm
    case thumb
    case index
    case middle
    case ring
    case little
}

enum Bone: Int, CaseIterable {
    case arm
```

```
    case wrist
    case metacarpal
    case knuckle
    case intermediateBase
    case intermediateTip
    case tip
}
```

The sample also defines a `Hand` structure that stores data for joints in a hand:

```
struct Hand {
    /// The collection of joints in a hand.
    static let joints: [(HandSkeleton.JointName, Finger, Bone)] = [
        // Define the thumb bones.
        (.thumbKnuckle, .thumb, .knuckle),
        (.thumbIntermediateBase, .thumb, .intermediateBase),
        (.thumbIntermediateTip, .thumb, .intermediateTip),
        (.thumbTip, .thumb, .tip),

        // ...

        // Define wrist and arm bones.
        (.forearmWrist, .forearm, .wrist),
        (.forearmArm, .forearm, .arm)
    ]
}
```

Set up the hand tracker

The sample creates the `HandTrackingComponent` to track the entities of the left and right hand. It differentiates the left- and right-hand entities by configuring the component with a `chirality` value and registering it with the app's `HandTrackingSystem` singleton instance:

```
struct HandTrackingComponent: Component {
    /// The chirality for the hand this component tracks.
    let chirality: AnchoringComponent.Target.Chirality

    /// A lookup that maps each joint name to the entity that represents it.
    var fingers: [HandSkeleton.JointName: Entity] = [:]

    /// Creates a new hand-tracking component.
```

```
    init(chirality: AnchoringComponent.Target.Chirality) {
        self.chirality = chirality
        HandTrackingSystem.registerSystem()
    }
}
```

The sample implements a custom system that tracks a person's hands with a `HandTrackingProvider` and stores them in the `latestLeftHand` and `latestRightHand` instances:

```
struct HandTrackingSystem: System {
    /// The ARKit session for hand tracking.
    private let arSession = ARKitSession()

    /// The provider instance for hand tracking.
    private let handTracking = HandTrackingProvider()

    /// The most recent anchor that the provider detects on the left hand.
    @State var latestLeftHand: HandAnchor?

    /// The most recent anchor that the provider detects on the right hand.
    @State var latestRightHand: HandAnchor?

    init(scene: RealityKit.Scene) {
        Task { await Self.runSession() }
    }

    // ...
}
```

On initialization, the `HandTrackingSystem` starts a task that calls the `runSession()` method.

> **Tip**
>
> If you want to later cancel `runSession()`, keep a reference to the `Task` that encapsulates it.

The `runSession()` method starts an `ARKitSession` with the `HandTrackingProvider`:

```
func runSession() {
    Task {
        do {
            // Attempt to run the ARKit session with the hand-tracking provider.
```

```
            try await arSession.run([handTracking])
        } catch let error as ARKitSession.Error {
            print("The app has encountered an error while running providers: \(error
        } catch let error {
            print("The app has encountered an unexpected error: \(error.localizedDes
        }

        // Start collecting each hand-tracking anchor.
        for await anchorUpdate in handTracking.anchorUpdates {
            // Check if the anchor is on the left or right hand.
            switch anchorUpdate.anchor.chirality {
            case .left:
                self.latestLeftHand = anchorUpdate.anchor
            case .right:
                self.latestRightHand = anchorUpdate.anchor
            }
        }
    }
}
```

When the `handTracking.anchorUpdates` stream yields new hand data from ARKit, this async method updates the `latestLeftHand` and `latestRightHand` anchors accordingly.

Visualize the joints of a hand entity

The system also has an `addJoints(to:handComponent:)` method that adds a sphere entity to each anchor of a hand entity. The method starts by creating the sphere entity with the `radius` and `material` properties, then it adds the sphere entity to the hand entity and updates the `fingers` collection for each anchor in the `Hand` structure:

```
func addJoints(to handEntity: Entity, handComponent: inout HandTrackingComponent) {
    /// The size of the sphere mesh.
    let radius: Float = 0.01

    /// The material to apply to the sphere entity.
    let material = SimpleMaterial(color: .white, isMetallic: false)

    /// The sphere entity that represents a hand-tracking anchor.
    let sphereEntity = ModelEntity(
        mesh: .generateSphere(radius: radius),
        materials: [material]
```

```
    )

    // For each anchor, create a sphere and attach it to the fingers.
    for bone in Hand.joints {
        let newJoint = sphereEntity.clone(recursive: false)
        handEntity.addChild(newJoint)
        handComponent.fingers[bone.0] = newJoint
    }

    // Apply the updated hand component back to the hand entity.
    handEntity.components.set(handComponent)
```

The `System` protocol has an `update(context:)` method that the app calls for each scene update, to update its hand entities.

> **Note**
>
> The `update(context:)` instance method is required to create a custom system.

The app finds the hand entities for each scene update with an `EntityQuery` instance that retrieves entities with a hand-tracking component.

```
struct HandTrackingSystem: System {
    /// The query this system uses to find all entities with the hand-tracking compo
    static let query = EntityQuery(where: .has(HandTrackingComponent.self))

    func update(context: SceneUpdateContext) {
        let handEntities = context.entities(matching: Self.query, updatingSystemWher

        // ...
    }
}
```

The system's `update(context:)` method:

1. Sets up hand-tracking anchors for each hand entity with the `addJoints(to:handComponent:)` method
2. Determines the hand's anchor based on the chirality of the entity's hand component
3. Updates the transform of the joints entity to match the transform of a person's hand joint that ARKit detects

```
for entity in handEntities {
    guard var handComponent = entity.components[HandTrackingComponent.self] else { c

    // Set up the finger joint entities if you haven't yet.
    if handComponent.fingers.isEmpty {
        self.addJoints(to: entity, handComponent: &handComponent)
    }

    // Get the hand anchor for the component, depending on its chirality.
    guard let handAnchor: HandAnchor = switch handComponent.chirality {
        case .left: Self.latestLeftHand
        case .right: Self.latestRightHand
        default: nil
    } else { continue }

    // Iterate through all of the anchors on the hand skeleton.
    if let handSkeleton = handAnchor.handSkeleton {
        for (jointName, jointEntity) in handComponent.fingers {
            /// The current transform of the person's hand joint.
            let anchorFromJointTransform = handSkeleton.joint(jointName).anchorFromJ

            // Update the joint entity to match the transform of the person's hand j
            jointEntity.setTransformMatrix(
                handAnchor.originFromAnchorTransform * anchorFromJointTransform,
                relativeTo: nil
            )
        }
    }
}
```

Create the left- and right-hand entities

The app adds hand-tracking entities to the content of the RealityKit view by calling `makeHandEntities(in:)`:

```
struct HandTrackingView: View {
    /// The main body of the view.
    var body: some View {
        RealityView { content in
            content.add(makeHandEntities())
        }
```

```
    }

    @MainActor
    func makeHandEntities(in content: any RealityViewContentProtocol) {
        // Add the left hand.
        let leftHand = Entity()
        leftHand.components.set(HandTrackingComponent(chirality: .left))
        content.add(leftHand)

        // Add the right hand.
        let rightHand = Entity()
        rightHand.components.set(HandTrackingComponent(chirality: .right))
        content.add(rightHand)
    }
}
```

The `makeHandEntities(in:)` method creates the left- and right-hand entities, and adds a `HandTrackingComponent` instance, each with the chirality case that correlates to its hand.

See Also

Integrating ARKit

{} Creating a 3D painting space

Implement a painting canvas entity, and update its mesh to represent a stroke.

{} Displaying an entity that follows a person's view

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

{} 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.