Sample Code

Finding devices with precision

Leverage the spatial awareness of ARKit and Apple Ultra Wideband Chips in your app to guide users to a nearby device.

Download

iOS 16.0+ | iPadOS 16.0+ | Xcode 15.0+

Overview

This sample code project demonstrates how to use ARKit to find devices equipped with Apple Ultra Wideband (UWB) chips using a technique known as *ranging*. Ranging estimates the distance between devices using parameters such as signal strength and packet timing.

> **Note**
>
> To use Extended Distance Measurement (EDM) capabilities, you'll need two iPhone 15 devices or later. In Xcode, set the deployment target for this sample to iOS 17.

To use the Nearby Interaction framework in your app, it needs to implement the following three processes to use precision device finding:

1. Check for availability of UWB and specific versions according to iOS releases if your app needs capabilities such as EDM.
2. Start ARKit, begin ranging for peer devices, and respond to changes in their status — this is how your app discovers compatible nearby devices and responds to changes in their availability.
3. Implement a distance quality estimator function, if your app needs to take advantage of the EDM capabilities of the second-generation UWB chip. This is a function that uses data the Nearby Interaction framework returns to make decisions about signal quality to estimate the

distance and direction between devices. This sample code project includes one possible implementation of a distance quality estimator function and its use to guide the people using the appropriate UI.

Configure the sample code project

To run the app:

1. Install the app on two iPhone 15 devices or later.
2. Rest one iPhone in portrait orientation as a stationary device.
3. Run the app. If available, tap "Discuss jetpacks with another visitor" on both devices to run ranging with EDM capabilities (available on devices with second-generation UWB chips). Otherwise, tap "Go to next Exhibit" on both devices to run first-generation UWB ranging.
4. On the first run, accept the request prompts for local network and camera access.
5. Wait a few moments for the apps to discover each other, then accept the Nearby Interaction access prompt.
6. To initialize ARKit, move the nonstationary device from side to side.
7. With the stationary device in the immediate vicinity, move the active device up and down until the app displays an arrow.
8. Move the active device until the arrow points upward in the direction of the stationary device.
9. On the active device, the app's blurred background fades and several virtual spheres display above the stationary device as it appears within the active device's AR view.

> Note
>
> This sample app doesn't support Apple Watch and, because Camera Assistance uses ARKit, the app doesn't support Simulator.

Check availability

Because there are two types of UWB chips, always check to ensure that the peer devices support the mode the app is interested in, as shown in the example below:

```
if #available(iOS 17.0, watchOS 10.0, *) {
    guard NISession.deviceCapabilities.supportsExtendedDistanceMeasurement else {
        NSLog("This device isn't capable of finding visitors.")
        return
    }

    guard token.deviceCapabilities.supportsExtendedDistanceMeasurement else {
        NSLog("Peer device \(peer.displayName) isn't capable of finding visitors.")
        return
    }

    config.isExtendedDistanceMeasurementEnabled = true
    NSLog("The Nearby Interaction session uses extended distance measurement.")

} else {
    NSLog("This version of iOS isn't capable of finding visitors.")
}
```

Start ARKit and ranging for peer devices

Before the app can start ranging for potential peer devices, it needs to setup ARKit and the Multipeer Connectivity frameworks, as shown in this example:

```
func startup() {
    // The initial view.
    Task { @MainActor in
        self.updateViewState(with: nil, quality: .unknown, nearbyObject: nil, worldT
    }

    // Create the interaction session.
    session = NISession()
    session?.delegateQueue = sessionQueue

    // Set a delegate.
    session?.delegate = self

    // Because this is a new session, reset the token-shared flag.
    sharedTokenWithPeer = false
    connectedPeer = nil

    // Start multipeer connectivity (MPC) to discover peers.
    startupMPC()
}
```

Respond to changes in peer devices

After the app initializes the Multipeer Connectivity framework, the Nearby Interaction framework provides a protocol for tracking changes to peer devices and provides information necessary to interact with peers it discovers in the environment.

Use `session(_:didUpdate:)` to track changes to peer connectivity, as shown in the sample below:

```
func session(_ session: NISession, didUpdate nearbyObjects: [NINearbyObject]) {
    guard let peerToken = peerDiscoveryToken else {
        fatalError("don't have peer token")
    }

    // Find the right peer.
    let peerObj = nearbyObjects.first { (obj) -> Bool in
        return obj.discoveryToken == peerToken
    }

    guard let nearbyObjectUpdate = peerObj else {
        return
    }

    // When the session is ranging with its peer, the data connection might
    // drop; after which which you don't need to keep it.
    // Tear down the MPC session after the app initially started ranging with the pe
    // After the current ranging session stops and is invalidated, the app
    // restarts a new MPC data connection for a new peer.
    if mpc != nil {
        tearDownMpc()
    }

    // Update and compute with updated `nearbyObject`.
    currentNearbyObject = nearbyObjectUpdate
    computeViewState(with: convergenceContext, nearbyObject: nearbyObjectUpdate)
}
```

When peers drop out of range or their session validity (user permissions) changes, the framework announces these changes using `session(_:didInvalidateWith:)` and the app responds accordingly, as shown here:

```
func session(_ session: NISession, didInvalidateWith error: Error) {
    // If the app doesn't have approval for Nearby Interaction, present
    // an option to open the Settings app where the they can update the access.
    if #available(iOS 17.0, watchOS 10.0, *) {
        switch error {
        case NIError.userDidNotAllow,
            NIError.invalidARConfiguration,
            NIError.incompatiblePeerDevice,
            NIError.activeSessionsLimitExceeded,
            NIError.activeExtendedDistanceSessionsLimitExceeded:
            return
        default:
            break
        }
    } else {
        switch error {
        case NIError.userDidNotAllow,
            NIError.invalidARConfiguration,
            NIError.activeSessionsLimitExceeded:
            return
        default:
            break
        }
    }

    // Recreate a valid session in other failure cases.
    startup()
}
```

When the app sets `isCameraAssistanceEnabled` to `true`, the framework provides coaching suggestions through `session(_:didUpdateAlgorithmConvergence:for:)`, as shown here:

```
func session(_ session: NISession, didUpdateAlgorithmConvergence convergence: NIAlgo
    guard let peerToken = peerDiscoveryToken else {
        fatalError("Don't have peer token.")
    }

    guard let nearbyObject = object, nearbyObject.discoveryToken == peerToken else {
        return
    }
    // Update and compute with updated algorithm `convergence` and `nearbyObject`.
    currentNearbyObject = nearbyObject
    convergenceContext = convergence
    computeViewState(with: convergence, nearbyObject: currentNearbyObject)
}
```

Define a distance quality estimator

Finally, the key to using EDM is by creating a distance quality estimator function that the app uses to determine which peers to interact with. The specific criteria for such estimators are specific to your app's use case; the following example is one such implementation:

```
class MeasurementQualityEstimator {

    // Define the criteria that qualify a peer with "good" characteristics:
    // these include:

    // A time window, in seconds.
    let freshnessWindow = TimeInterval(floatLiteral: 2.0)
    // A minimum number of samples in that time window.
    let minSamples: Int = 8
    // A maximim distance, in meters.
    let maxDistance: Float = 50
    // A minimum distance, in meters.
    let closeDistance: Float = 10

    // A buffer to hold the individual quality measurements.
    private var measurements: [TimedNIObject] = []

    // An enumeration that defines levels of peer quality.
    enum MeasurementQuality {
        // The peer fails to meet any of the measurement quality criteria.
        case unknown

        // The extended distance measurements indicate the peer iPhone or device
        // satisfies the criteria for "good" quality and falls inside the
        // minimum and maximum acceptable distance.
        case good

        // The extended distance measurements indicate the current device
        // satisfies the criteria for being "close" to the peer iPhone or device.
        case close
    }

    // A structure that captures the range of a peer at a specific time.
    struct TimedNIObject {
        let time: TimeInterval
        let distance: Float
    }

    func estimateQuality(update: NINearbyObject?) -> MeasurementQuality {
        let timeNow = NSDate().timeIntervalSinceReferenceDate
        if let distance = update?.distance {
            if let lastMeasureMent = measurements.last {
                if lastMeasureMent.distance != distance {
                    // Before adding a new measurement to buffers, check
                    // if the reported distance is unique.
                    measurements.append(TimedNIObject(time: timeNow, distance: dista
                }
            } else {
                // If the buffer is empty, unconditionally add the new measurement.
                measurements.append(TimedNIObject(time: timeNow, distance: distance)
            }
        }
        let validTimestamp = timeNow - freshnessWindow
        measurements.removeAll { $0.time < validTimestamp }
        if measurements.count > minSamples, let lastDistance = measurements.last?.di
            if lastDistance <= closeDistance { return .close }
            return lastDistance < maxDistance ? .good : .unknown
        }
        return .unknown
    }
}
```

See Also

Camera assistance

`class NIAlgorithmConvergence`

An object that provides the state and reason for user coaching recommendations.

`enum NIAlgorithmConvergenceStatus`

The possible states of Camera Assistance.

Algorithm Convergence Status

The possible Objective-C states of Camera Assistance.