Sample Code

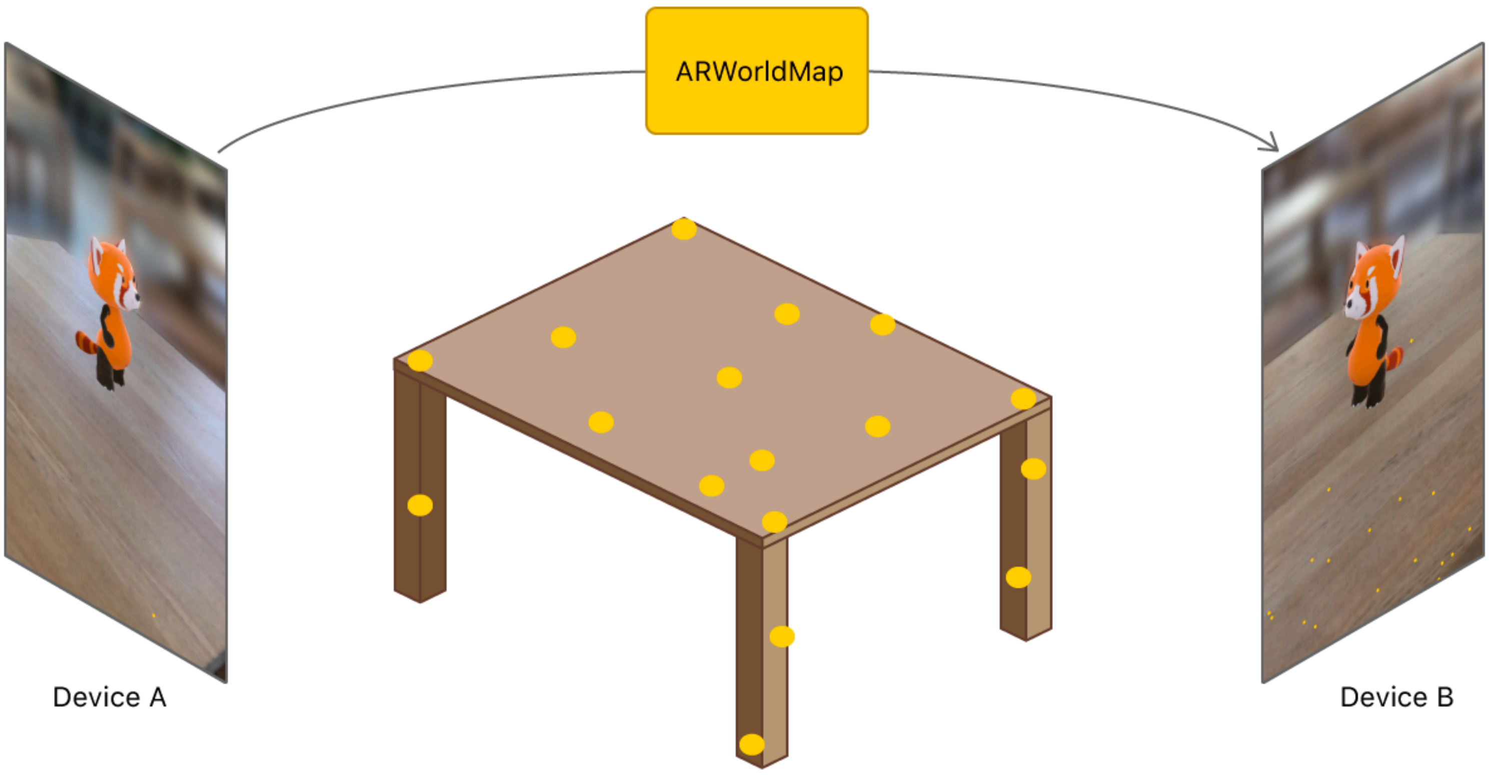
Creating a multiuser AR experience

Enable nearby devices to share an AR experience by using a host-guest multiuser strategy.

Download

iOS 12.0+ | iPadOS 12.0+ | Xcode 16.1+

Overview



This sample app demonstrates a simple shared AR experience for two or more iOS 12 devices. Before exploring the code, try building and running the app to familiarize yourself with the user experience it demonstrates:

1. Run the app on one device. You can look around the local environment, and tap to place a virtual 3D character on real-world surfaces. (Tap again to place multiple copies of the character.)
2. Run the app on a second device. On both device screens, a message indicates that they have automatically joined a shared session.
3. Tap the Send World Map button on one device. Make sure the other device is in an area that the first device visited before sending the map, or has a similar view of the surrounding environment.
4. The other device displays a message indicating that it has received the map and is attempting to use it. When that process succeeds, both devices show virtual content at the same real-world positions, and tapping on either device places virtual content visible to both.

Follow the steps below to see how this app uses the `ARWorldMap` class to save and restore ARKit's spatial mapping state, and the Multipeer Connectivity framework to send world maps between nearby devices.

Getting started

Requires Xcode 10.0, iOS 12.0 and two or more iOS devices with A9 or later processors.

Run the AR Session and Place AR Content

This app extends the basic workflow for building an ARKit app. (For details, see Tracking and visualizing planes.) It defines an `ARWorldTrackingConfiguration` with plane detection enabled, then runs that configuration in the `ARSession` attached to the `ARSCNView` that displays the AR experience.

When `UITapGestureRecognizer` detects a tap on the screen, the `handleSceneTap` method uses ARKit hit-testing to find a 3D point on a real-world surface, then places an `ARAnchor` marking that position. When ARKit calls the delegate method `renderer(_:didAdd:for:)`, the app loads a 3D model for `ARSCNView` to display at the anchor's position.

Connect to peer devices

The sample `MultipeerSession` class provides a simple abstraction around the Multipeer Connectivity features this app uses. After the main view controller creates a `MultipeerSession` instance (at app launch), it starts running an `MCNearbyServiceAdvertiser` to broadcast the device's ability to join multipeer sessions and an `MCNearbyServiceBrowser` to find other devices:

```
session = MCSession(peer: myPeerID, securityIdentity: nil, encryptionPreference: .re
session.delegate = self

serviceAdvertiser = MCNearbyServiceAdvertiser(peer: myPeerID, discoveryInfo: nil, se
serviceAdvertiser.delegate = self
serviceAdvertiser.startAdvertisingPeer()

serviceBrowser = MCNearbyServiceBrowser(peer: myPeerID, serviceType: MultipeerSessio
serviceBrowser.delegate = self
serviceBrowser.startBrowsingForPeers()
```

When the `MCNearbyServiceBrowser` finds another device, it calls the `browser(_:foundPeer:withDiscoveryInfo:)` delegate method. To invite that other device to a shared session, call the browser's `invitePeer(_:to:withContext:timeout:)` method:

```
public func browser(_ browser: MCNearbyServiceBrowser, foundPeer peerID: MCPeerID, w
    // Invite the new peer to the session.
    browser.invitePeer(peerID, to: session, withContext: nil, timeout: 10)
}
```

When the other device receives that invitation, `MCNearbyServiceAdvertiser` calls the `advertiser(_:didReceiveInvitationFromPeer:withContext:invitationHandler:)` delegate method. To accept the invitation, call the provided `invitationHandler`:

```
func advertiser(_ advertiser: MCNearbyServiceAdvertiser,
                didReceiveInvitationFromPeer peerID: MCPeerID,
                withContext context: Data?,
                invitationHandler: @escaping (Bool, MCSession?) -> Void) {
    // Call handler to accept invitation and join the session.
    invitationHandler(true, self.session)
}
```

Important

This app automatically joins the first nearby session it finds. Depending on the kind of shared AR experience you want to create, you may want to more precisely control broadcasting, invitation, and acceptance behavior. See the Multipeer Connectivity documentation for details.

In a multipeer session, all participants are by definition equal peers; there is no explicit separation of devices into host and guest roles. However, you may wish to define such roles for your own AR experience. For example, a multiplayer game design might require a host role to arbitrate gameplay. If you need to separate peers by role, you can choose a way to do so that fits the design of your app. For example:

- Have the user choose whether to act as a host or guest before starting a session. The host uses `MCNearbyServiceAdvertiser` to broadcast availability, and guests use `MCNearbyServiceBrowser` to find a host to join.
- Join a session as peers, then negotiate between peers to nominate a host. (This approach can be helpful for designs that need a host role but also allow peers to join or leave at any time.)

Capture and send the ar world map

An `ARWorldMap` object contains a snapshot of all the spatial mapping information that ARKit uses to locate the user's device in real-world space. Reliably sharing a map to another device requires two key steps: finding a good time to capture a map, and capturing and sending it.

ARKit provides a `worldMappingStatus` value that indicates whether it's currently a good time to capture a world map (or if it's better to wait until ARKit has mapped more of the local environment). This app uses that value to provide visual feedback on its Send World Map button:

```
switch frame.worldMappingStatus {
case .notAvailable, .limited:
    sendMapButton.isEnabled = false
case .extending:
    sendMapButton.isEnabled = !multipeerSession.connectedPeers.isEmpty
case .mapped:
    sendMapButton.isEnabled = !multipeerSession.connectedPeers.isEmpty
@unknown default:
    sendMapButton.isEnabled = false
}
mappingStatusLabel.text = frame.worldMappingStatus.description
```

When the user taps the Send World Map button, the app calls `getCurrentWorldMap(completionHandler:)` to capture the map from the running ARSession, then serializes it to a `Data` object with `NSKeyedArchiver` and sends it to other devices in the multipeer session:

```
sceneView.session.getCurrentWorldMap { worldMap, error in
    guard let map = worldMap
        else { print("Error: \(error!.localizedDescription)"); return }
    guard let data = try? NSKeyedArchiver.archivedData(withRootObject: map, requirir
        else { fatalError("can't encode map") }
    self.multipeerSession.sendToAllPeers(data)
}
```

Receive and relocalize to the shared map

When a device receives data sent by another participant in the multipeer session, the `session(_:didReceive:fromPeer:)`delegate method provides that data. To make use of it, the app uses `NSKeyedArchiver` to deserialize an `ARWorldMap` object, then creates and runs a new `ARWorldTrackingConfiguration` using that map as the `initialWorldMap`:

```
if let worldMap = try NSKeyedUnarchiver.unarchivedObject(ofClass: ARWorldMap.self, f
    // Run the session with the received world map.
    let configuration = ARWorldTrackingConfiguration()
    configuration.planeDetection = .horizontal
    configuration.initialWorldMap = worldMap
    sceneView.session.run(configuration, options: [.resetTracking, .removeExistingAr

    // Remember who provided the map for showing UI feedback.
    mapProvider = peer
}
```

ARKit then attempts to *relocalize* to the new world map—that is, to reconcile the received spatial-mapping information with what it senses of the local environment. For best results:

1. Thoroughly scan the local environment on the sending device before sharing a world map.
2. Place the receiving device next to the sending device, so that both see the same view of the environment.

Share AR content and user actions

Sharing the world map also shares all existing anchors. In this app, this means that as soon as a receiving device relocalizes to the world map, it shows all the 3D characters that were placed by the sending device before it captured and sent a world map. However, recording and transmitting a world map and relocalizing to a world map are time-consuming, bandwidth-intensive operations, so you should take those steps only once, when a new device joins a session.

To create an ongoing shared AR experience, where each user's actions affect the AR scene visible to other users, after each device relocalizes to the same world map you should share only the information needed to recreate each user action. For example, in this app the user can tap to place a virtual 3D character in the scene. That character is static, so all that is needed to place the character on another participating device is the character's position and orientation in world space.

This app communicates virtual character positions by sharing ARAnchor objects between peers. When one user taps in the scene, the app creates an anchor and adds it to the local ARSession, then serializes that ARAnchor using NSKeyedArchiver and sends it to other devices in the multipeer session:

```
// Place an anchor for a virtual character. The model appears in renderer(_:didAdd:f
let anchor = ARAnchor(name: "panda", transform: hitTestResult.worldTransform)
sceneView.session.add(anchor: anchor)

// Send the anchor info to peers, so they can place the same content.
guard let data = try? NSKeyedArchiver.archivedData(withRootObject: anchor, requiring
    else { fatalError("can't encode anchor") }
self.multipeerSession.sendToAllPeers(data)
```

When other peers receive data from the multipeer session, they test for whether that data contains an archived ARAnchor; if so, they decode it and add it to their session:

```
if let anchor = try NSKeyedUnarchiver.unarchivedObject(ofClass: ARAnchor.self, from:
    // Add anchor to the session, ARSCNView delegate adds visible content.
    sceneView.session.add(anchor: anchor)
}
```

This is just one strategy for adding dynamic features to a shared AR experience—many other strategies are possible. Choose one that fits the user interaction, rendering, and networking requirements of your app. For example, a game where users throw projectiles in the AR world space might define custom data types with attributes like initial position and velocity, then use Swift's Codable protocols to serialize that information to a binary representation for sending over the network.

See Also

Shared Experiences

{} Streaming an AR experience

Control an AR experience remotely by transferring sensor and user input over the network.

{} Creating a collaborative session

Enable nearby devices to share an AR experience by using a peer-to-peer multiuser strategy.

`class ARParticipantAnchor`

An anchor for another user in multiuser augmented reality experiences.

`class CollaborationData`

An object that holds information that a user has collected about the physical environment.