Sample Code

Streaming an AR experience

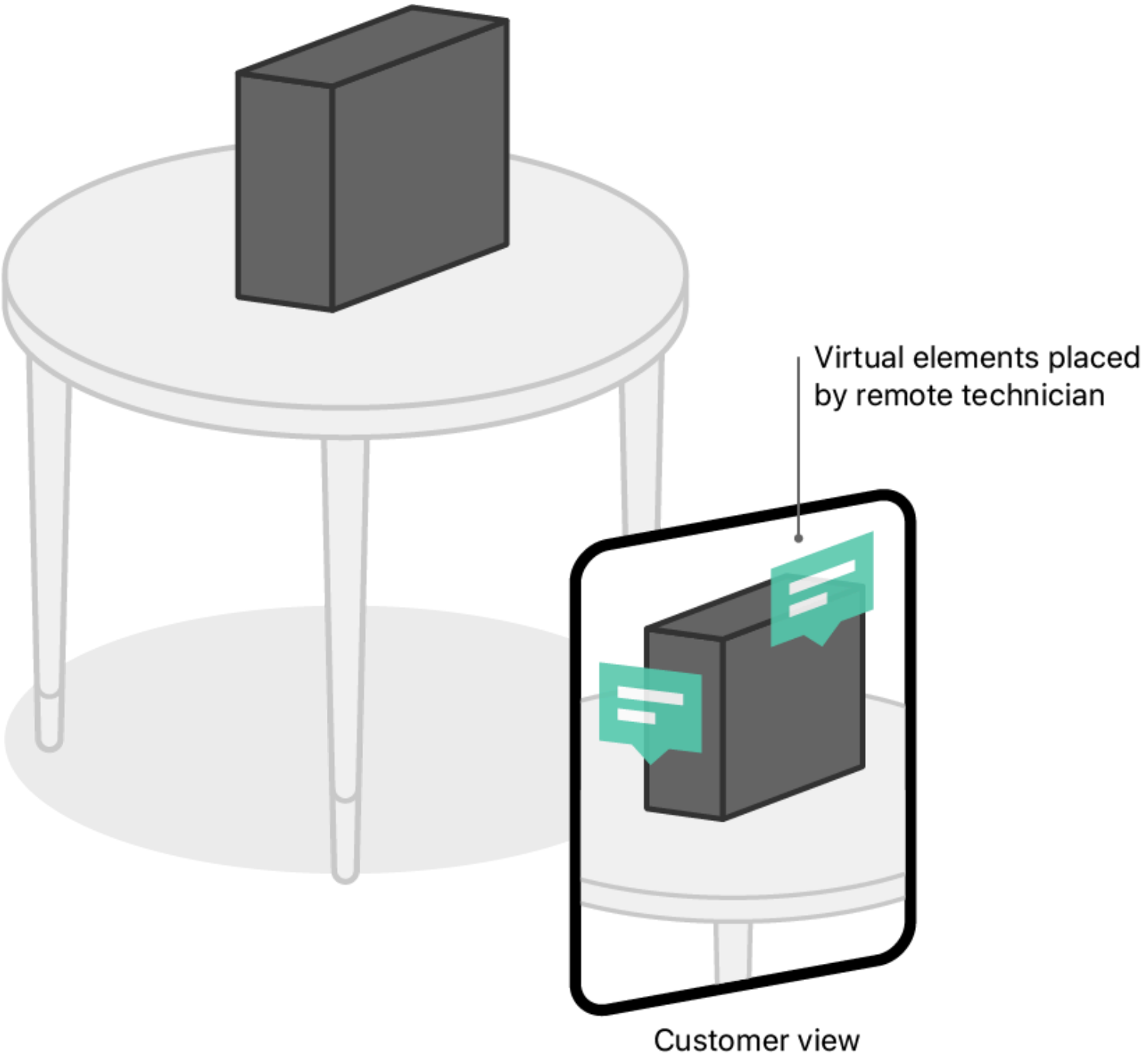
Control an AR experience remotely by transferring sensor and user input over the network.

Download

iOS 13.0+ | iPadOS 13.0+ | Xcode 16.0+

Overview

The sample app, AR Stream, shares the augmented camera feed with a peer device, and enables it to take control by interacting with the remote AR experience. For example, a user shares their screen depicting their physical environment with a computer technician who assists the user with troubleshooting a hardware issue. As the user views a broken device resting on a table from different angles, the remote technician interacts with the experience by augmenting the user's camera feed with textual annotations that describe the necessary steps to repair the device.



To enable the remote user to see the user's physical environment, AR Stream shares device sensor information across the network. By compressing camera frames with Video Toolbox, the app provides the peer with good visibility of the user's view by displaying the remote experience at a high frame rate.

AR Stream also sends mathematical details about the user's real-world pose to the remote user to process the peer's touch input. The sample app sends the session's inverse view and inverse projection matrices to the remote device so it can calculate a location in the user's environment where the remote user taps. To indicate when the remote user taps the screen, AR Stream places a helpful virtual indicator at the tap location.

Display a camera feed and monitor the session

AR Stream displays the device's camera feed by configuring a window with a view controller that displays an `ARView` (see the sample project's `Main.storyboard` file). By default,`ARView` runs a session with a world-tracking configuration `ARWorldTrackingConfiguration`. To receive notifications of the view's session events, the project's view controller (see `ViewController` in the sample project) assigns itself as the session delegate.

```
arView.session.delegate = self
```

Capture frames

To show the user's physical environment to the remote user, AR Stream uses ReplayKit to open a screen-recording session with `RPScreenRecorder`.

```
RPScreenRecorder.shared().startCapture {
```

The screen recording captures the contents of the app's main window, which includes any augmentations that RealityKit may add to the camera feed. In the `startCapture(handler: completionHandler:)` closure, the sample project passes the captured screen (`sample Buffer`) to the `compressAndSend` function for eventual transmission over the network. The sample project also passes in the session's current frame to conform the screen captures to the camera-image size.

```
if type == .video {
    guard let currentFrame = arView.session.currentFrame else { return }
    videoProcessor.compressAndSend(sampleBuffer, arFrame: currentFrame) {
```

> **Note**
>
> Although `ARView` provides a `snapshot(saveToHDR:completion:)` function to capture the contents of the view, ReplayKit's screen recording is more conducive to real-time capture.

Compress and send frames to the peer

The sample project's `VideoProcessor` class implements the `compressAndSend` function, which uses `VTCompressionSession` to compress the captured video frames.

```
VTCompressionSessionEncodeFrame(compressionSession,
    imageBuffer: imageBuffer,
    presentationTimeStamp: presentationTimeStamp,
    duration: .invalid,
    frameProperties: nil,
    infoFlagsOut: nil) {
```

To ensure timely compression for the real-time streaming use case of the app, the video processor enables the compression session's `kVTCompressionPropertyKey_RealTime` option.

```
VTSessionSetProperty(compressionSession, key: kVTCompressionPropertyKey_RealTime,
    value: kCFBooleanTrue)
```

After the `VTCompressionSession` finishes encoding a frame, the app creates a `VideoFrameData` instance using the compressed frame and the inverse view and projection matrices from the corresponding `ARFrame`.

```
let videoFrameData = VideoFrameData(sampleBuffer: sampleBuffer, arFrame: arFrame)
```

The project serializes and encodes the `VideoFrameData` as JSON data, and passes the data to its `sendHandler`.

```
do {
    let data = try JSONEncoder().encode(videoFrameData)
    // Invoke the caller's handler to send the data.
    sendHandler(data)
} catch {
    fatalError("Failed to encode videoFrameData as JSON with error: "
        + error.localizedDescription)
}
```

The screen-recording closure defines the send handler to contain code that uses Multipeer Connectivity to transmit the video data over the local network.

```
multipeerSession.sendToAllPeers(data, reliably: true)
```

Receive and decompress peer frames

When the app receives `VideoFrameData` from another device, it decodes the JSON data.

```
func receivedData(_ data: Data, from peer: MCPeerID) {
    // Try to decode the received data and handle it appropriately.
    if let videoFrameData = try? JSONDecoder().decode(VideoFrameData.self,
        from: data) {
```

To house the transmitted video frame, AR Stream reconstructs a sample buffer.

```
let sampleBuffer = videoFrameData.makeSampleBuffer()
```

The system can display only uncompressed data, so the video processor decompresses the video frame using `VTDecompressionSession` within its `decompress` function.

```
VTDecompressionSessionDecodeFrame(decompressionSession,
    sampleBuffer: sampleBuffer,
    flags: [],
    infoFlagsOut: nil) {
```

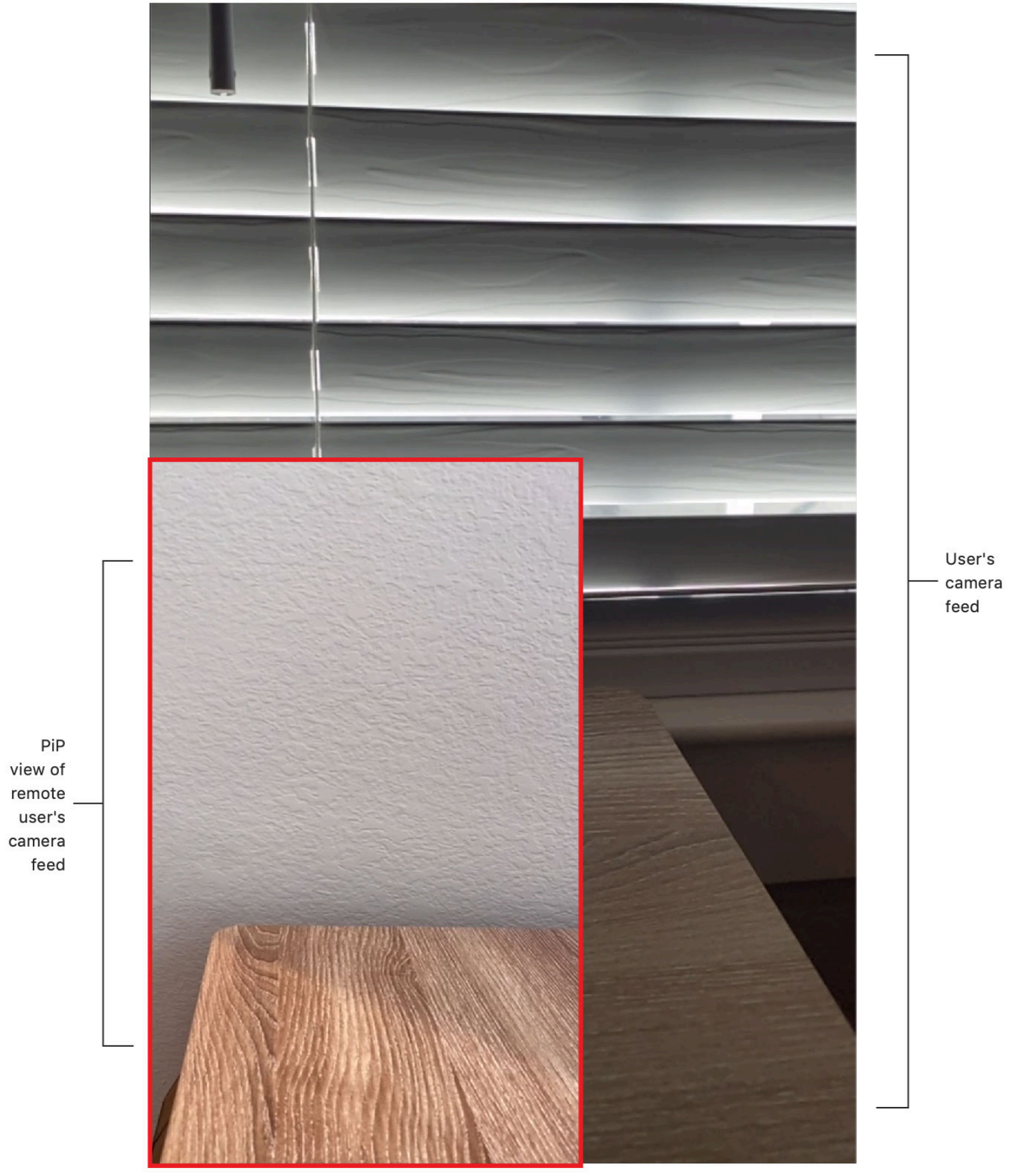
AR Stream draws the video frame to the screen using its renderer object (see `Renderer` in the sample project). The renderer enqueues the frame data for imminent display.

```
// Update the PipView aspect ratio to match the camera-image dimensions.
let width = CGFloat(CVPixelBufferGetWidth(imageBuffer))
let height = CGFloat(CVPixelBufferGetHeight(imageBuffer))
overlayViewController?.setPipViewConstraints(width: width, height: height)

overlayViewController?.renderer.enqueueFrame(
    pixelBuffer: imageBuffer,
    presentationTimeStamp: presentationTimeStamp,
    inverseProjectionMatrix: videoFrameData.inverseProjectionMatrix,
    inverseViewMatrix: videoFrameData.inverseViewMatrix)
```

Display the remote user's camera feed

AR Stream defines an `MTKView` subclass, `OverlayViewController`, that displays the remote user's camera feed on top of the `ARView` by placing a *picture-in-picture* (PiP) view at the bottom left of the screen.



The sample project's `AppDelegate` configures the PiP view in a secondary window. Because ReplayKit's screen recording captures only the main window, the PiP view displays only the remote user's camera feed.

```
overlayWindow = UIWindow(windowScene: windowScene)
```

```
let storyBoard = UIStoryboard(name: "Main", bundle: nil)
let overlayViewController = storyBoard.instantiateViewController(
    identifier: "OverlayViewController")
overlayWindow.rootViewController = overlayViewController
overlayWindow.makeKeyAndVisible()

// Make sure the overlayWindow is always above the main window.
overlayWindow.windowLevel = window.windowLevel + 1
```

Send gestures to the peer

When the remote user taps the PiP view, the project responds by recording the tap location.

```
@objc
func tapped(_ sender: UITapGestureRecognizer) {
    guard let view = sender.view else { return }
    let location = sender.location(in: view)
```

The sample project uses the inverse matrices that the user sends to enable the remote user to interact with the user's AR experience.

```
guard let inverseProjectionMatrix = renderer.lastDrawnInverseProjectionMatrix,
    let inverseViewMatrix = renderer.lastDrawnInverseViewMatrix else {
    return
}
```

The project converts the tap location and inverse matrices into a ray cast that describes the location and direction in the user's `ARSession` world coordinate system (see the `makeRay` function in the sample project).

```
let rayQuery = makeRay(from: location,
    viewportSize: view.frame.size,
    inverseProjectionMatrix: simd_float4x4(inverseProjectionMatrix),
    inverseViewMatrix: simd_float4x4(inverseViewMatrix))
```

Then, the sample project encodes the ray cast as JSON data and sends it to the connected peer.

```
let data = try JSONEncoder().encode(rayQuery)
multipeerSession?.sendToAllPeers(data, reliably: true)
```

Handle peer gestures

In the project's `ViewController`, the `receivedData` function receives a `Ray` object when the remote user taps the PiP view.

```
} else if let rayQuery = try? JSONDecoder().decode(Ray.self, from: data) {
```

To hand the remote user's tap gesture to ARKit as if the user is tapping the screen, the sample project uses the `Ray` data to create an `ARTrackedRaycast`.

```
trackedRaycast = arView.session.trackedRaycast(
    ARRaycastQuery(
        origin: rayQuery.origin,
        direction: rayQuery.direction,
        allowing: .estimatedPlane,
        alignment: .any)
    ) {
```

When the tracked ray cast intersects with a surface in the user's environment, the app records the resulting location.

```
if let result = raycastResults.first {
    marker.transform.matrix = result.worldTransform
```

Display virtual content

To enable the remote user to interact with the user's AR experience, the app places a virtual ball at the location in the environment where the remote user taps.

Important

AR Stream displays a virtual ball for simplicity. An app may require different virtual content, such as an arrow that points to a precise spot, or virtual text that explains the importance of a location. For an example app that displays text at a real-world location, see Creating screen annotations for objects in an AR experience.

The project creates this visual marker using a ball-shaped `ModelEntity`.

```
let marker: AnchorEntity = {
    let entity = AnchorEntity()
    entity.addChild(ModelEntity(mesh: .generateSphere(radius: 0.05)))
    entity.isEnabled = false
    return entity
}()
```

At app launch, the marker is invisible by default as the project readies the marker for display by adding it to the scene.

```
arView.scene.addAnchor(marker)
```

When the app receives a `Ray` from the remote user and adjusts the marker's position, the project displays the marker by enabling it.

```
marker.isEnabled = true
```

See Also

Shared Experiences

{} Creating a collaborative session

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

{} Creating a multiuser AR experience

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