Building Collaborative AR Experiences

Kuen-han Lin, ARKit Engineering
David Paschich, Tools Foundation
Collaborative Session

ARAnchor Best Practice for Multi-User AR

SwiftStrike — A New Multiplayer AR Experience
Collaborative Session
Saving and Loading Map in ARKit 2

Recap

ARWorldMap data
- Map of sparse 3D landmarks
- List of named ARAnchors
Multi-User Experiences in ARKit 2

Recap
Multi-User Experiences in ARKit 2

Recap
Multi-User Experiences in ARKit 2

Recap
Multi-User Experiences in ARKit 2

Recap
Collaborative Session

Live multi-user AR experience

Continuously share ARAnchors and map data

No host user
Different Coordinates in Collaborative Session
Different Coordinates in Collaborative Session
Different Coordinates in Collaborative Session

User 2
External Map

User 1
External Map

Anchor 1

User 1 World

Anchor 2

User 2 World
Different Coordinates in Collaborative Session

User 1 External Map

User 2 External Map

Anchor 1

User 1 World

Anchor 2

User 2 World
Different Coordinates in Collaborative Session
Different Coordinates in Collaborative Session

User 2
External Map

User 1
External Map

Anchor 1

User 1
World

Anchor 2

User 2
World
Different Coordinates in Collaborative Session
Different Coordinates in Collaborative Session
Collaborative Session Flow

Set up communication

User 1
Set up Network

ARSession Starts
Collaboration Data

Localize User 2 Map
Receive User 2 ARAnchor

User 2
Set up Network

ARSession Starts
Collaboration Data

Localize User 1 Map
Receive User 1 ARAnchor
Collaborative Session Flow
Transmit collaboration data

User 1
- Set up Network
- ARSession Starts
- Collaboration Data
- Localize User 2 Map
- Receive User 2 ARAnchor

User 2
- Set up Network
- ARSession Starts
- Collaboration Data
- Localize User 1 Map
- Receive User 1 ARAnchor
// Collaborative Session

// Create world tracking configuration
let config = ARWorldTrackingConfiguration()

// Enable collaborative session
config.isCollaborationEnabled = true

// Run the configuration
session.run(config)
// Collaborative Session

// Create world tracking configuration
let config = ARWorldTrackingConfiguration()

// Enable collaborative session
config.isCollaborationEnabled = true

// Run the configuration
session.run(config)
// Collaborative Session

// Create world tracking configuration
let config = ARWorldTrackingConfiguration()

// Enable collaborative session
config.isCollaborationEnabled = true

// Run the configuration
session.run(config)
// Collaborative Session

// Create world tracking configuration
let config = ARWorldTrackingConfiguration()

// Enable collaborative session
config.isCollaborationEnabled = true

// Run the configuration
session.run(config)
func session(_ session: ARSession, didOutputCollaborationData data: ARSession.CollaborationData) {
    // Transmit Data representation of the data to all other participants using MPC
    do {
        try self.mpcSession.send(data.data, toPeers: self.peerIds, with: .reliable)
    } catch {
        // Re-transmit the data if failed
    }
}

func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
    // Pass the received data to ARSession
    self.arSession.update(data: ARSession.CollaborationData(data: data))
}
// ARSession delegate function to output ARCollaborationData

func session(_ session: ARSession,
didOutputCollaborationData data: ARSession.CollaborationData) {
    // Transmit Data representation of the data to all other participants using MPC
    do {
        try self.mpcSession.send(data.data, toPeers: self.peerIds, with: .reliable)
    } catch {
        // Re-transmit the data if failed
    }
}

// MPC delegate function when receiving collaboration data from other users
func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
    // Pass the received data to ARSession
    self.arSession.update(data: ARSession.CollaborationData(data: data))
}
// ARSession delegate function to output ARCollaborationData
func session(_ session: ARSession,
    didOutputCollaborationData data: ARSession.CollaborationData) {
    // Transmit Data representation of the data to all other participants using MPC
do {
        try self.mpcSession.send(data.data, toPeers: self.peerIds, with: .reliable)
    } catch {
        // Re-transmit the data if failed
    }
}

// MPC delegate function when receiving collaboration data from other users
func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
    // Pass the received data to ARSession
    self.arSession.update(data: ARSession.CollaborationData(data: data))
}
Collaborative Session Flow

Start shared AR experience

User 1

Set up Network → ARSession Starts → Collaboration Data → Localize User 2 Map → Receive User 2 ARAnchor

User 2

Set up Network → ARSession Starts → Collaboration Data → Localize User 1 Map → Receive User 1 ARAnchor
Collaborative Session Flow
Start shared AR experience

User 1
Set up Network → ARSession Starts → Collaboration Data → Localize User 2 Map → Receive User 2 ARAnchor

User 2
Set up Network → ARSession Starts → Collaboration Data → Localize User 1 Map → Receive User 1 ARAnchor
ARAnchor in Collaborative Session

ARAnchor

- Synchronized lifetime
- Session identifier
- Subclass ARAnchors are not shared
// ARAnchor in collaborative session

// ARSession delegate function when an ARAnchor is added.
func session(_ session: ARSession, didAdd anchors: [ARAnchor]) {
    for anchor in anchors {
        // Use session identifier to determine creator of the ARAnchor
        if anchor.sessionIdentifier == session.identifier {
            // Self-placed ARAnchor
        } else {
            // ARAnchor from another participant
        }
    }
}
// ARAnchor in collaborative session

// ARSession delegate function when an ARAnchor is added.
func session(_ session: ARSession, didAdd anchors: [ARAnchor]) {
    for anchor in anchors {
        // Use session identifier to determine creator of the ARAnchor
        if anchor.sessionIdentifier == session.identifier {
            // Self-placed ARAnchor
        } else {
            // ARAnchor from another participant
        }
    }
}
func session(_ session: ARSession, didRemove anchors: [ARAnchor]) {
    for anchor in anchors {
        // Use session identifier to determine creator of the ARAnchor
        if anchor.sessionIdentifier == session.identifier {
            // Self-placed ARAnchor
        } else {
            // ARAnchor from another participant
        }
    }
}
// ARSession delegate function when an ARAnchor is removed.

func session(_ session: ARSession, didRemove anchors: [ARAnchor]) {
    for anchor in anchors {
        // Use session identifier to determine creator of the ARAnchor
        if anchor.sessionIdentifier == session.identifier {
            // Self-placed ARAnchor
        } else {
            // ARAnchor from another participant
        }
    }
}
ARParticipantAnchor

- Position of other user
- Update at high frame rate
- Created after localizing other user’s map
Practical Advice for Localizing Other Users

Shared AR experience starts after seeing same area
Practical Advice for Localizing Other Users

Shared AR experience starts after seeing same area

Approach other user to have same camera perspective
Practical Advice for Localizing Other Users

Shared AR experience starts after seeing same area

Approach other user to have same camera perspective

Stay in map-tracking status

ARFrame.WorldMappingStatus.mapped
ARAnchor Best Practice for Multi-User AR
Behind the Scenes — ARWorldMap

Internal tracking data
- Map of sparse 3D landmarks
- Camera poses
Behind the Scenes — ARWorldMap

Internal tracking data
• Map of sparse 3D landmarks
• Camera poses
Behind the Scenes — ARWorldMap

Internal tracking data
- Map of sparse 3D landmarks
- Camera poses

Landmarks are grouped based on camera view
AR Anchor

- Global position against world
Behind the Scenes — ARAnchor

ARAnchor
- Global position against world
- Relative position against map
Best Practice for Using ARAnchor

Respond to anchor update delegate
Best Practice for Using ARAnchor

Respond to anchor update delegate
Best Practice for Using ARAnchor

Respond to anchor update delegate

Place 3D content near the ARAnchor
Best Practice for Using ARAnchor

Respond to anchor update delegate

Place 3D content near the ARAnchor
Best Practice for Using ARAnchor

- Respond to anchor update delegate
- Place 3D content near the ARAnchor
- Multiple anchors for independent 3D content
Best Practice for Using ARAnchor

Respond to anchor update delegate

Place 3D content near the ARAnchor

Multiple anchors for independent 3D content
SwiftStrike — A New Multiplayer AR Experience

David Paschich, Tools Foundation
SwiftStrike — A New Multiplayer AR Experience

Inspired by SwiftShot

All-new experience for 2019

Built with RealityKit and ARKit 3

SwiftStrike Tabletop available now
RealityKit networking
Physics simulation
SwiftStrike game design
RealityKit Networking

Based on entity-component architecture

All state synchronized, including physics!

Custom components for app/game logic

Uses MultipeerConnectivity as network layer

Create network session, hand to ARView
Different Roles

First device acts as "host"
Controls game state, physics simulation
Other devices participate
Custom Components in RealityKit

Define your own components to store application state

Register components before instantiating ARView

Implement `Codable` to enable synchronization
Use Case — Starting the Game

Match object tracks whether enough players are ready to start the game

State maintained on host, synced to clients

Component maintains full log
// Custom component for game start

struct MatchStateComponent: Component, Codable {
    struct Transition: Codable {
        var date: Date
        var state: MatchOutput
    }
    var transitions = [Transition]()
}

// Registering the component, in application(_:didFinishLaunchingWithOptions:)
MatchStateComponent.self.registerComponent()

// On client
class MatchObserver {
    var matchOutputEvents: AnyPublisher<MatchOutput, Never>
}
// Custom component for game start

struct MatchStateComponent: Component, Codable {
    struct Transition: Codable {
        var date: Date
        var state: MatchOutput
    }
    var transitions = [Transition]()}

// Registering the component, in application(_:didFinishLaunchingWithOptions:)
MatchStateComponent.self.registerComponent()

// On client
class MatchObserver {
    var matchOutputEvents: AnyPublisher<MatchOutput, Never>
// Custom component for game start

struct MatchStateComponent: Component, Codable {
    struct Transition: Codable {
        var date: Date
        var state: MatchOutput
    }
    var transitions = [Transition]()
}

// Registering the component, in application(_:didFinishLaunchingWithOptions:)
MatchStateComponent.self.registerComponent()

// On client
class MatchObserver {
    var matchOutputEvents: AnyPublisher<MatchOutput, Never>
}
// Custom component for game start

struct MatchStateComponent: Codable {
    struct Transition: Codable {
        var date: Date
        var state: MatchOutput
    }
    var transitions = [Transition]()
}

// Registering the component, in application(_:didFinishLaunchingWithOptions:)
MatchStateComponent.self.registerComponent()

// On client
class MatchObserver {
    var matchOutputEvents: AnyPublisher<MatchOutput, Never>
}
<table>
<thead>
<tr>
<th>Ray</th>
<th>GamePhysicsSmoothComponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CameraInfo</td>
<td>SlingshotComponent</td>
</tr>
<tr>
<td>GameVelocity</td>
<td>GameBoardDescription</td>
</tr>
<tr>
<td>HitCatapult</td>
<td>GameBoardLocation</td>
</tr>
<tr>
<td>SlingData</td>
<td>LevelConfiguration</td>
</tr>
<tr>
<td>GrabInfo</td>
<td>LevelInfo</td>
</tr>
<tr>
<td>LeverMove</td>
<td>LeverMove</td>
</tr>
<tr>
<td>PhysicsSyncData</td>
<td>BitStreamEncodable</td>
</tr>
<tr>
<td>CollisionSoundData</td>
<td>BitStreamDecodable</td>
</tr>
<tr>
<td>PhysicsSyncSceneData</td>
<td>WriteableBitStream</td>
</tr>
<tr>
<td>PhysicsSyncSceneDataDelegate</td>
<td>ReadableBitStream</td>
</tr>
</tbody>
</table>
Ray
CameraInfo
GameVelocity
HitCatapult
SlingData
GrabInfo
LeverMove
PhysicsSyncData
CollisionSoundData
PhysicsSyncSceneData
PhysicsSyncSceneDataDelegate
GamePhysicsSmoothComponent
SlingshotComponent
GameBoardDescription
GameBoardLocation
LevelConfiguration
LevelInfo
LeverMove
BitStreamEncodable
BitStreamDecodable
WriteableBitStream
ReadableBitStream
Physics

Physics state synchronization handled by RealityKit

Configure physics properties via components
• Rigid bodies
• Collision masks
• Mass, friction, restitution ("bounciness")

Host device owns simulation

Client devices interpolate between updates
Bowling Pin
Bowling Pin
Bowling Pin
Bowling Pin
SwiftStrike Game Design

Designing for People Occlusion

On-site experience

Control mechanism
People Occlusion

SwiftStrike designed with this in mind
Opens up new experiences
SwiftStrike On-Site Experience

Full scale experience

Floor features designed for quick localization
SwiftStrike Control Mechanism

Device is your controller

Faster movement means larger push

Physics body in the scene
• Ball bounces off people
• Pins and ball bounce off each other
• Pins don’t bounce off people
Ownership of Player Paddle
Ownership of Player Paddle

AnchorEntity ——— Host
Ownership of Player Paddle

- Client
- PlayerLocationEntity
- AnchorEntity
- Host
Ownership of Player Paddle
Ownership of Player Paddle

Client

AnchorEntity

PlayerLocationEntity

PaddleEntity

Host

Host
Ownership of Player Paddle

Client

AnchorEntity

PlayerLocationEntity

PaddleEntity

Host

Host
Summary

Collaborative Sessions enable new experiences

Use ARAnchors to attach content to the real world

SwiftStrike Tabletop available now!
More Information

developer.apple.com/wwdc19/610

<table>
<thead>
<tr>
<th>ARKit Lab</th>
<th>Thursday, 3:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealityKit and Reality Composer Lab</td>
<td>Thursday, 3:00</td>
</tr>
</tbody>
</table>