Sample Code

Synchronizing group gameplay with TabletopKit

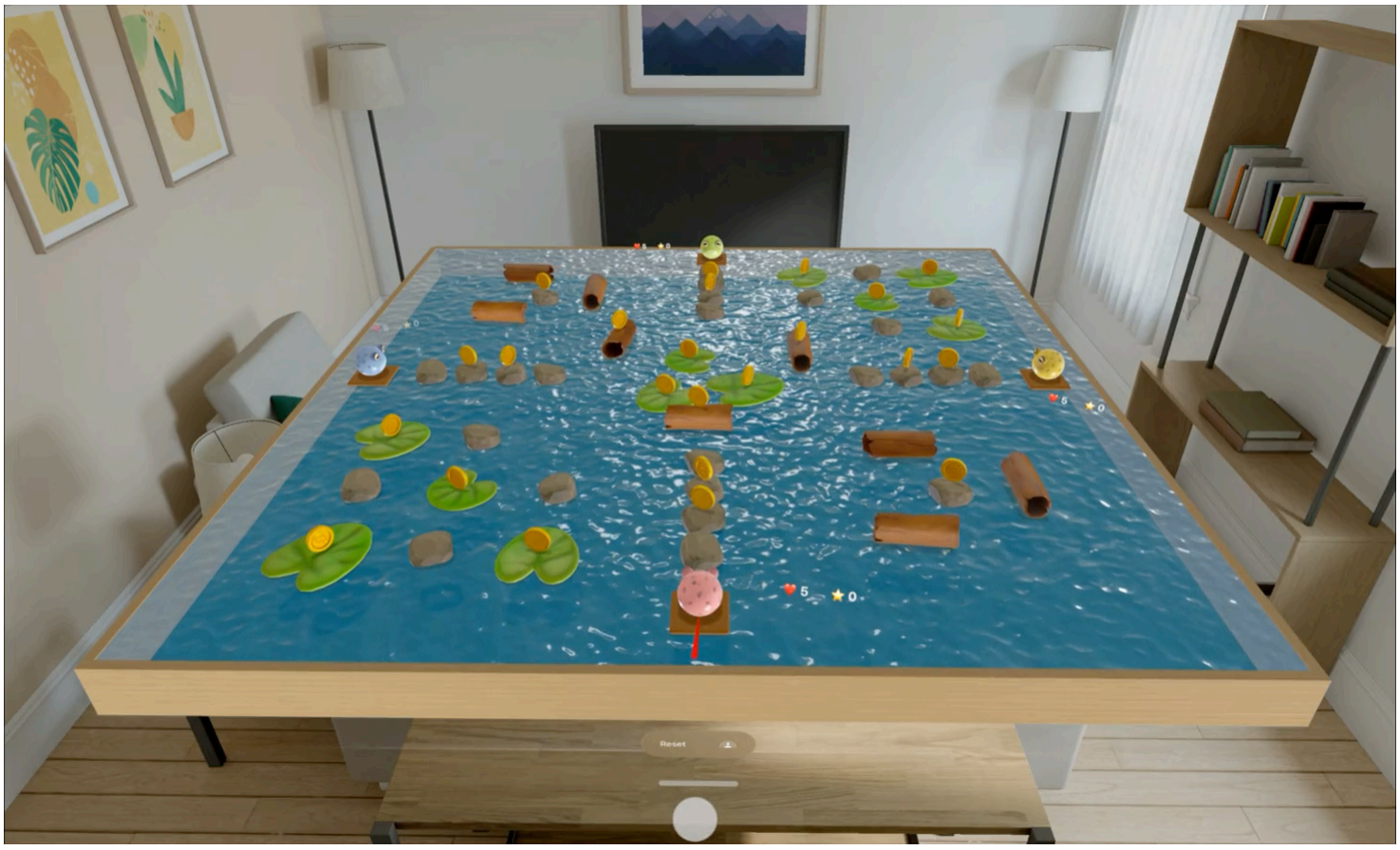
Maintain game state across multiple players in a race to capture all the coins.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

This sample code project demonstrates how to overcome the difficult task of maintaining a multiplayer game's state in real time. TabletopKit supports automatic game-state synchronization by keeping gameplay items, such as player tokens, dice, coins, and scenery updates — all with positions and actions — in sync during a multiplayer game. TabletopKit also supports many more gameplay styles than the traditional board game layout.



Play ⦿

Set up the app

The TabletopKit app object creates an instance of the `Game` class, which sets up, observes, and renders the `TabletopGame` state. `Game` equipment, objects that implement the `Equipment` protocol, represents all interactive portions of a game. The `Game` class also handles game start and reset, initiating programmatic interactions, and adding equipment reset actions. The `GameSetup` class initializes and positions all game equipment.

The `GameRenderer` class implements `TabletopGame.RenderDelegate`, which loads assets and communicates when the game needs visual updates. The `GameObserver` class implements `TabletopGame.Observer` and indicates confirmed gameplay actions. You can define custom actions to modify the game state to your needs. The `CustomAction.swift` file in this sample contains custom actions to reset the players, decrement player health, collect coins, reset coins, sink lily pads, and reset lily pads.

```
class Game {
    let tabletopGame: TabletopGame
    let renderer: GameRenderer
    let observer: GameObserver
    let setup: GameSetup

    //...
```

```
}
```

The `GameView` structure contains the `RealityView`, which hosts game content and the toolbar for player interaction, and stores the `Game` object as an environment property. The `tabletopGame(_:parent:automaticUpdate:)` modifier connects the `TabletopGame` object to the `RealityView`. The modifier returns the appropriate delegate for the game equipment ID that passes into the update closure. A `Task` creates `GroupActivityManager`, which connects multiple players and provides the `TabletopGame` object.

```
var body: some View {
    GeometryReader3D { proxy3D in
        RealityView { (content: inout RealityViewContent) in
            content.entities.append(volumetricRoot)
            // Set the root at the base of the volume.
            let frame = content.convert(proxy3D.frame(in: .local), from: .local, to:
            volumetricRoot.transform.translation.y = frame.min.y
            volumetricRoot.addChild(game.renderer.root)
        }
    }.toolbar() {
        GameToolbar(game: game)
    }.tabletopGame(game.tabletopGame, parent: game.renderer.root) { value in
        var delegate: GameInteraction?

        if let _ = game.tabletopGame.equipment(of: Log.self, matching: value.startir
            delegate = LogInteraction(game: game)
        } else if let _ = game.tabletopGame.equipment(of: LilyPad.self, matching: va
            delegate = LilyPadInteraction(game: game)
        } else if let _ = game.tabletopGame.equipment(of: Player.self, matching: val
            delegate = PlayerInteraction(game: game)
        } else {
            delegate = GameInteraction(game: game)
        }

        return delegate!
    }.task {
        activityManager = .init(tabletopGame: game.tabletopGame)
    }
}
```

The player joins other players by tapping the SharePlay button in the volume's toolbar. This instantiates an `Activity` object that implements the `GroupActivity` protocol. `Activity` initializes an observable `GroupActivityManager` awaiting `Activity` sessions that coordinate

with the provided `TabletopGame`. TabletopKit handles all player connections and synchronization. The app only needs to register its equipment and actions, and then handle their updates.

```
struct Activity: GroupActivity {
    var metadata: GroupActivityMetadata {
        var metadata = GroupActivityMetadata()
        metadata.type = .generic
        metadata.title = "TabletopKitSample"
        return metadata
    }
}

class GroupActivityManager: Observable {
    var tabletopGame: TabletopGame
    var sessionTask = Task<Void, Never> {}

    init(tabletopGame: TabletopGame) {
        self.tabletopGame = tabletopGame
        sessionTask = Task { @MainActor in
            for await session in Activity.sessions() {
                tabletopGame.coordinateWithSession(session)
            }
        }
    }

    deinit {
        tabletopGame.detachNetworkCoordinator()
    }
}
```

Define gameplay

The game begins when the player taps the Start Game button in the toolbar. This sets the `Game.gameStarted` state to `true` and initiates `Log` entity interactions. Objects that implement the `TabletopInteraction.Delegate` protocol handle all gameplay event updates.

```
class GameInteraction: TabletopInteraction.Delegate {
    let game: Game

    init(game: Game) {
```

```
        self.game = game
    }

    func update(interaction: TabletopKit.TabletopInteraction) {

    }
}
```

`PlayerInteraction` handles player input by implementing `TabletopInteraction.Delegate`. When the player gazes at a gameplay object and interacts using direct or indirect gestures, the `PlayerInteraction` receives a `TabletopInteraction` object in its `update(interaction:)` method. TabletopKit supplies an active gesture for the interaction. The sample splits this functionality into two parts — updating based on user gestures and updating due to automatic programmatic interactions.

```
class PlayerInteraction: GameInteraction {
    override func update(interaction: TabletopInteraction) {
        // A gesture interaction to aim the jump.
        if interaction.value.gesture != nil || interaction.value.startingEquipmentID
            updateGestureInteraction(interaction: interaction)
            return
        }

        // A programmatic interaction for the jump after the player releases the aim
        updateProgrammaticInteraction(interaction: interaction)
    }
}
```

`PlayerInteraction` calls into `updateGestureInteraction` to handle gesture events. On gesture start, `updateGestureInteraction` sets the `.aimingSightID` for the controlled equipment on the interaction. This allows TabletopKit to manipulate the appropriate equipment entity onscreen.

```
func updateGestureInteraction(interaction: TabletopInteraction) {
    guard let gesture = interaction.value.gesture else { return }
    if gesture.phase == .started {
        guard let player = game.tabletopGame.equipment(matching: interaction.value.s
        interaction.setControlledEquipment(matching: .aimingSightID(for: player.seat
        return
    }
    //...
```

On gesture update, `updateGestureInteraction` modifies the local slingshot visuals, matching user input.

```
    //...
    if gesture.phase == .update {
        // Update the slingshot visuals while the player is still dragging.
        game.tabletopGame.withCurrentSnapshot { snapshot in
            guard let (playerEquip, _) = snapshot.equipment(of: Player.self, matchir
            let aimX = interaction.value.pose.position.x
            let aimZ = interaction.value.pose.position.z
            let root = game.renderer.root
            Task { @MainActor in
                playerEquip.updateAimingVisuals(dragPosition: .init(x: aimX, z: aimZ
            }
        }
        return
    }
    //...
```

On gesture end, `updateGestureInteraction` calculates the change in gesture position, calls `startInteraction(onEquipmentID:)` for the interaction's equipment, and adds it to the programmatic player interaction dictionary of `Game`. When gesture handling is complete, TabletopKit moves the player's piece.

```
    //...
    if gesture.phase == .ended {
        // When the player releases the aim, hide the aiming visuals and start a pro
        game.tabletopGame.withCurrentSnapshot { snapshot in
            if let (playerEquip, _) = snapshot.equipment(of: Player.self, matching:
                Task { @MainActor in
                    playerEquip.hideAimingVisuals()
                }
            }

            guard let interactionIdentifier = game.tabletopGame.startInteraction(onE
                return
            }
            guard let (playerEquip, _) = snapshot.equipment(of: Player.self, matchir

            let targetX = interaction.value.pose.position.x
            let targetZ = interaction.value.pose.position.z
            let root = game.renderer.root
```

```
            Task { @MainActor in
                game.programmaticPlayerInteractions[interactionIdentifier] = playerE
                    dragPosition: .init(x: targetX, z: targetZ),
                    root: root
                )
                playerEquip.playJumpAudio()
            }
        }
        return
    }
}
```

Next, `PlayerInteraction.updateProgrammaticInteraction` handles the programmatic interactions of automated equipment. At the beginning of the interaction, it provides the set of available interaction destinations — the stones, lily pads, and logs.

```
func updateProgrammaticInteraction(interaction: TabletopInteraction) {
    if interaction.value.phase == .started {
        interaction.setConfiguration(.init(allowedDestinations: .restricted(.allStor
        return
    }
    //...
```

During the interaction, the app finds a target programmatic player interaction for the provided interaction ID and sets the target's position.

```
    //...
    if interaction.value.phase == .update {
        guard let targetPose = game.programmaticPlayerInteractions[interaction.value
        let oldPose = interaction.value.pose

        if abs(oldPose.position.x - targetPose.position.x) < 1e-3 && abs(oldPose.pos
            if interaction.value.proposedDestination == nil {
                sinkPlayer(interaction: interaction, targetPose: targetPose)
                return
            }
            interaction.setPose(targetPose)
            interaction.end()
            return
        }
        movePlayer(interaction: interaction, targetPose: targetPose)
```

```
        return
    }
    //
```

At the end of the interaction, `updateProgrammaticInteraction` calls into `endJump`. Adding a `MoveEquipmentAction` object to the active `TabletopInteraction` moves the player. If the target lands on a lily pad, the sample initiates the sinking animation by calling `startInteraction(onEquipmentID:)` with the lily pad's equipment ID. If the target lands on an allowed destination with a coin, `endJump` adds a `CollectCoin` action to the active `TabletopInteraction`. The app then removes the interaction from the `programmaticPlayerInteractions` dictionary.

```
    //...
    if interaction.value.phase == .ended {
        endJump(interaction: interaction)
    }
}

func endJump(interaction: TabletopInteraction) {
    if let proposedDestination = interaction.value.proposedDestination {
        // Move the player to the proposed destination.
        interaction.addAction(.moveEquipment(matching: interaction.value.controlledE
                                             childOf: proposedDestination.equipmentI

        // If the destination is a lily pad, sink it.
        if game.tabletopGame.equipment(of: LilyPad.self, matching: proposedDestinati
            _ = game.tabletopGame.startInteraction(onEquipmentID: proposedDestinatio
        }

        // If the destination contains an uncollected coin, collect it.
        game.tabletopGame.withCurrentSnapshot { snapshot in
            if let childId = snapshot.equipmentIDs(childrenOf: proposedDestination.e
               let coinState = snapshot.state(matching: childId) as? CoinState {
                if !coinState.collected {
                    interaction.addAction(CollectCoin(playerId: interaction.value.co
                }
            }
        }
        game.programmaticPlayerInteractions.removeValue(forKey: interaction.value.id
        return
    }
    //...
```

Finally, if the landing site isn't a valid location, `endJump` returns the player to their starting position and decrements their health.

```
    // If the player doesn't land on a valid destination, return them to their start
    let player = game.tabletopGame.equipment(of: Player.self, matching: interaction.
    interaction.addAction(.moveEquipment(matching: player.id, childOf: .bankID(for:
    interaction.addAction(DecrementHealth(playerId: interaction.value.controlledEqui
    game.programmaticPlayerInteractions.removeValue(forKey: interaction.value.id)
}
```

The `GameObserver` class is responsible for reacting to confirmed game actions. The game decrements a player's health when the player lands in the water, or if they fail to jump again quickly after landing on a lily pad before it sinks. Within `actionWasConfirmed()`, the game resets the player's state if the newly provided `TableSnapshot` matches a `ResetPlayer` action. The actions can have other possible matches, like `DecrementHealth` where the game decrements the player's health, or `CollectCoin` where the player collects a coin at the player's new location. The game ends when the players collect all of the coins, or when all of the players run out of lives.

```
class GameObserver: TabletopGame.Observer {
    func actionWasConfirmed(_ action: some TabletopAction, oldSnapshot: TableSnapsho

        guard let game else {
            return
        }

        if let resetPlayerAction = ResetPlayer(from: action) {
            let (equip, state) = newSnapshot.equipment(of: Player.self, matching: [r
            game.playerStats[equip.seat].health = state.health
            game.playerStats[equip.seat].coinsCount = state.coinsCount

            return
        }

        if let decrementHealthAction = DecrementHealth(from: action) {
            let (equip, state) = newSnapshot.equipment(of: Player.self, matching: [d
            game.playerStats[equip.seat].health = state.health

            // Freeze the player when their health equals `0`.
            if action.playerID == game.tabletopGame.localPlayer.id && state.health =
                game.tabletopGame.addAction(FreezePlayer(playerId: equip.id))
            }
```

```
            return
        }

        if let collectCoinAction = CollectCoin(from: action) {
            let (equip, playerState) = newSnapshot.equipment(of: Player.self, matchi
            game.playerStats[equip.seat].coinsCount = playerState.coinsCount

            return
        }
    }
}
```

See Also

Essentials

{} Creating tabletop games

Develop a spatial board game where multiple players interact with pieces on a table.

`class TabletopGame`

An object that manages the setup and gameplay of a tabletop game.

`struct TableSetup`

An object that represents the arrangement of seats, equipment, and counters around the game table.

`protocol Tabletop`

A protocol for the table surface in your game.

`protocol EntityTabletop`

A protocol for the table surface in your game when you render it using RealityKit.

`struct TabletopShape`

An object that represents the physical properties of the table.