Sample Code

Drawing content in a group session

Invite your friends to draw on a shared canvas while on a FaceTime call.

Download

iOS 17.0+ | iPadOS 17.0+ | Xcode 15.0+

Overview

Note

This sample code project is associated with WWDC23 session 10241: Share files with SharePlay, and WWDC21 session 10187: Build custom experiences with Group Activities.

Group Activities allows you to build shared experiences across devices with SharePlay. Although its focus is on creating media experiences, you can design your own custom app experiences to share with multiple devices. The real-time interactions you unlock, and the instant reaction people who use your app get, can lead to some truly special moments.

The sample app, DrawTogether, builds on some of the concepts introduced in WWDC21 session 10225: Coordinate media experiences with Group Activities. It allows you to draw together while on a FaceTime call. The entire screen is the canvas, and everyone draws with a random color.

There are two steps to adopt a `GroupActivity`: activity creation and session management. These are covered in detail in WWDC21 session 10225: Coordinate media experiences with Group Activities and in Supporting Coordinated Media Playback.

These steps change slightly when building a custom experience, starting with activity creation. Activity creation consists of configuring the activity, and then activating the activity. Only the configuration part is different for a custom activity compared to a media activity.

Configure the sample code project

- Build the sample with Xcode 13 or later, and Swift 5.5 or later.
- This sample runs on physical devices with iOS 15 or later.

To see the custom drawing experience together, install the app on two or more devices with unique Apple IDs, and start a FaceTime call between the devices. Tap the bottom-left icon, and respond to the system prompt to start a shared DrawTogether experience. On the other device, join the group session. The app gives each user a random color to draw with. When the user draws on the screen, the app propagates the drawing to all the other devices in the group session.

Configure a custom activity

To make drawing a group experience, the sample defines a `DrawTogether` structure that adopts the `GroupActivity` protocol. This protocol defines a shareable experience in the app. The `GroupActivity` protocol has two properties that the app implements: `activityIdentifier` and `metadata`. The app relies on the default implementation for `activityIdentifier`, and the `metadata` property is required. To make this activity a custom activity, the sample code sets the `type` on the `metadata` to `generic`. This is crucial for a custom activity, and is the only difference between configuring a custom activity and a media activity.

```
struct DrawTogether: GroupActivity {
    var metadata: GroupActivityMetadata {
        var metadata = GroupActivityMetadata()
        metadata.title = NSLocalizedString("Draw Together", comment: "Title of group
        metadata.type = .generic
        return metadata
    }
}
```

Activate an activity

After configuring the activity, the app needs to activate it at the appropriate moment. The sample provides a button for activation. First, the sample checks to make sure a `GroupSession` doesn't exist already. Next, it calls the `GroupStateObserver` `isEligibleForGroupSession` method to make sure a FaceTime call is active and the system can create group sessions. Then, in the action closure, the sample calls its `canvas.startSharing()` method to activate the activity.

```
if canvas.groupSession == nil && groupStateObserver.isEligibleForGroupSession {
    Button {
        canvas.startSharing()
```

```
    } label: {
        Image(systemName: "person.2.fill")
    }
    .buttonStyle(.borderedProminent)
}
```

In its `canvas.startSharing()` method, the sample creates a new instance of the custom activity, and calls the `activate()` method on it. That's all that's necessary to activate the activity.

```
func startSharing() {
    Task {
        do {
            _ = try await DrawTogether().activate()
        } catch {
            print("Failed to activate DrawTogether activity: \(error)")
        }
    }
}
```

Configure the session for sending and receiving custom data

The sample uses `GroupSessionMessenger` to configure the session for sending and receiving its custom drawing data. The app creates a `GroupSessionMessenger` from the `GroupSession`. It also adds the `messenger` property to its `Canvas` to hold the `messenger` object.

```
func configureGroupSession(_ groupSession: GroupSession<DrawTogether>) {
    strokes = []

    self.groupSession = groupSession
    let messenger = GroupSessionMessenger(session: groupSession)
    self.messenger = messenger
```

When using `GroupSessionMessenger`, the sample code defines the type of data to exchange between participants. The app shares the strokes themselves. The sample defines the `UpsertStrokeMessage` structure to represent a stroke with three properties: an identifier, a color, and a coordinate point. The sample also specifies that the `UpsertStrokeMessage` structure conforms to the `Codable` protocol. `GroupSessionMessenger` automatically handles the serialization and deserialization of the message data if the messages are `Codable`.

```
struct UpsertStrokeMessage: Codable {
    let id: UUID
    let color: Stroke.Color
    let point: CGPoint
}
```

The second step in configuring the session is to call the `GroupSessionMessenger messages(of:)` method to receive the `UpsertStrokeMessages` data. The sample specifies the `UpsertStrokeMessage` type when calling the `messages` method. This method returns an async sequence that provides a tuple containing messages of that type and the context surrounding the message, such as which participant sends the message.

```
for await (message, _) in messenger.messages(of: UpsertStrokeMessage.self) {
    handle(message)
}
```

The third step for configuring the session is to send data using the `GroupSessionMessenger send(_:to:)` method. The app sends an `UpsertStrokeMessage` to all participants within the group.

```
try? await messenger.send(UpsertStrokeMessage(id: stroke.id, color: stroke.color, po
```

Receive custom data

The sample creates a detached task to receive the `UpsertStrokeMessages` from the async sequence, then calls its `handle` method to process the message.

```
var task = Task {
    for await (message, _) in messenger.messages(of: UpsertStrokeMessage.self) {
        handle(message)
    }
}
tasks.insert(task)
```

The `handle` method checks the `stroke` identifier to see if one exists already — and if so, adds the point to it. Otherwise, it creates a new `stroke`, adds the point to it, and appends the `stroke` to an array of `strokes`.

```
func handle(_ message: UpsertStrokeMessage) {
    if let stroke = strokes.first(where: { $0.id == message.id }) {
        stroke.points.append(message.point)
    } else {
        let stroke = Stroke(id: message.id, color: message.color)
        stroke.points.append(message.point)
        strokes.append(stroke)
    }
}
```

Send custom data

The sample calls its `addPointToActiveStroke` method to send the messages using the `GroupSessionMessenger` `send(_:to:)` method.

```
func addPointToActiveStroke(_ point: CGPoint) {
    let stroke: Stroke
    if let activeStroke = activeStroke {
        stroke = activeStroke
    } else {
        stroke = Stroke(color: strokeColor)
        activeStroke = stroke
    }

    stroke.points.append(point)

    if let messenger = messenger {
        Task {
            try? await messenger.send(UpsertStrokeMessage(id: stroke.id, color: stro
        }
    }
}
```

Handle late joiners

Late joiners are devices that join an activity session after the session starts. To ensure a proper experience, the app gives late joiners the most recent information so all the devices use the same data.

When a new device calls `join()` on the `GroupSession`, every other device in the `GroupSession` receives an update of the `GroupSession` `activeParticipants` property. Devices

that observe the update then send their own catch-up data (in this case, the existing drawing canvas) to the newly joined device.

The app defines its catch-up data in the `CanvasMessage` structure. This structure contains all of the existing strokes and a variable `pointCount`, a heuristic that calculates which message is the most up-to-date.

```
struct CanvasMessage: Codable {
    let strokes: [Stroke]
    let pointCount: Int
}
```

The sample defines a message handler in its `configureGroupSession` method to receive this message using the `GroupSessionMessenger` `messages(of:)` method. The handler calls the samples `handle()` method to process the message.

```
task = Task {
    for await (message, _) in messenger.messages(of: CanvasMessage.self) {
        handle(message)
    }
}
```

The sample's `handle()` method guards against the `pointCount` heuristic to only accept catch-up messages that are newer than any currently saved messages. Then the sample overrides the `canvas strokes` with the catch-up message's `strokes`.

```
func handle(_ message: CanvasMessage) {
    guard message.pointCount > self.pointCount else { return }
    self.strokes = message.strokes
}
```

Next, the sample listens for `activeParticipants` changes to determine whether there are any new participants to communicate with. The sample's `configureGroupSession` method handler obtains the delta between the new `activeParticipants` and the previous `active Participants`. This ensures that the sample only sends catch-up messages to the newly joined participants. Finally, the sample forms and sends the message. The message contains the current `canvas` state and sends it only to the `newParticipants`.

```
groupSession.$activeParticipants
    .sink { activeParticipants in
```

```
        let newParticipants = activeParticipants.subtracting(groupSession.activePart

        Task {
            try? await messenger.send(CanvasMessage(strokes: self.strokes, pointCour
        }
    }
```

Change activities

The Group Activities framework provides two ways to change activities: create a `GroupSession`, or update the activity for everyone in the existing `GroupSession`.

The sample app resets for each new drawing canvas. The sample triggers a new session when the user taps the Reset button. In this case, the sample calls its `Canvas` `reset` method to reset the local state and create a `GroupSession`.

```
Button {
    canvas.reset()
} label: {
    Image(systemName: "trash.fill")
}
```

The sample's `reset` method removes the existing `GroupSession`, which allows for a clean transition to a new `canvas` in the new session. The `reset` method cancels any tasks for the `GroupSession`. It also checks for an existing `GroupSession`, and if one exists, it calls the `leave()` method to leave the current activity. It then calls the `activate()` method to start the activity immediately and create a session for the app. Thereafter, the sample waits for the system to deliver a `GroupSession` object asynchronously through the `GroupActivity.Sessions` method of the `GroupActivity`.

```
func reset() {
    // Clear the local drawing canvas.
    strokes = []

    // Tear down the existing groupSession.
    messenger = nil
    tasks.forEach { $0.cancel() }
    tasks = []
    subscriptions = []
    if groupSession != nil {
        groupSession?.leave()
        groupSession = nil
```

```
        self.startSharing()
    }
}

func startSharing() {
    Task {
        do {
            _ = try await DrawTogether().activate()
        } catch {
            print("Failed to activate DrawTogether activity: \(error)")
        }
    }
}
```

Start SharePlay experiences

The sample's `ControlBar` view contains a button to allow users to share the canvas with their friends. The sample dynamically displays the button only when it's helpful to the user. The sample uses the `GroupStateObserver` `isEligibleForGroupSession` property to determine whether a FaceTime call is active and the system can create group sessions. The `ControlBar` only shows the button if the system is eligible for a group session, and not in a group session already.

```
struct ControlBar: View {
    @ObservedObject var canvas: Canvas
    @StateObject var groupStateObserver = GroupStateObserver()

    var body: some View {
        HStack {
            if canvas.groupSession == nil && groupStateObserver.isEligibleForGroupSe
                Button {
                    canvas.startSharing()
                } label: {
                    Image(systemName: "person.2.fill")
                }
                .buttonStyle(.borderedProminent)
            }

            Spacer()

            Button {
                canvas.reset()
```

```
            } label: {
                Image(systemName: "trash.fill")
            }
        }
        .buttonStyle(.bordered)
        .controlSize(.large)
    }
}
```