Sample Code

Playing custom audio with your own player

Construct an audio player to play your custom audio data, and optionally take advantage of the advanced features of AirPlay 2.

Download

iOS 17.0+ | iPadOS 17.0+ | Xcode 15.4+

Overview

This sample code project builds a robust audio player from the ground up, using the `AVSampleBufferAudioRenderer` and `AVSampleBufferRenderSynchronizer` classes to manage enqueuing and playback of audio that you provide. The player uses a playlist of playable items, and allows the user to edit the contents of the playlist while playback is in progress.

The example app also uses `AVAudioSession` to indicate that it plays long-form audio content — music, audiobooks, podcasts, or other content that a person listens to over a substantial period of time. This allows the player to benefit from AirPlay 2. When the app plays to a compatible device such as HomePod, AirPlay 2 dramatically improves playback reliability and performance, and enables advanced features such as multiroom playback.

Specify long-form audio

To use AirPlay 2 for playback to compatible output devices, configure your audio session with the `AVAudioSession.RouteSharingPolicy.longFormAudio` route-sharing policy. Typically, you do this once when your app starts up:

```
do {
    try AVAudioSession.sharedInstance().setCategory(.playback, mode: .default, polic
} catch {
```

```
    print("Failed to set audio session route sharing policy: \(error)")
}
```

The `.longFormAudio` route-sharing policy is a hint to the system that your audio content is suitable for extended listening sessions. As a side effect, it also allows your audio to benefit from AirPlay 2 for extended buffering and improved responsiveness to commands.

You can choose not to configure your audio as long form if your content isn't intended to displace playback of long-form content from apps such as Apple Music, iTunes, or Podcasts.

Provide audio content

For your custom audio player, start by deciding how you want to identify content to the player. One way is to manage a playlist, a persistent list of playable items. Alternatively, use a temporary queue of items, a single item, or a continuous stream of audio that has no distinct identity as a separate item.

This sample app demonstrates the use of a playlist, each item representing a single music track. The player has public APIs that app code can use to manipulate the contents and order of the playlist, and start and stop playback. The `SampleBufferPlayer` class implements these APIs.

Internally, your player should contain logic to enqueue buffers in advance of their scheduled playback time, and to handle transitions between items.

In this example project, a `SampleBufferSerializer` object provides the enqueuing logic, using `SampleBufferItem` objects to wrap playable items and provide audio buffers on request. `SampleBufferItem` objects use `SampleBufferSource` objects to provide the basic audio data.

The following sections offer a more detailed discussion of the classes.

Manage your playlist

Use a `SampleBufferPlayer` to manage a playlist through a private `Playlist` structure:

```
private struct Playlist {

    // Items in the playlist.
    var items: [SampleBufferItem] = []

    // The current item index, or nil if the player is in a stopped state.
    var currentIndex: Int?
```

```
}
```

The `currentItemIndex` member represents the current state of the player. It indicates which element of the `items` array is the currently playing item, and implies a corresponding player state.

- If `currentItemIndex` is `nil`, there's no current item, and the player is in a stopped state.
- If `currentItemIndex` is not `nil`, it's a valid index into the `items` array, and the player is in a playing or paused state.

A number of `SampleBufferPlayer` methods manage the `Playlist`. For example, you use an `insertItem` method to insert an item into the queue:

```
func insertItem(_ newItem: PlaylistItem, at index: Int) {
    playbackSerializer.printLog(component: .player, message: "inserting item at play
    atomicitySemaphore.wait()
    defer { atomicitySemaphore.signal() }

    playlist.items.insert(playbackSerializer.sampleBufferItem(playlistItem: newItem,

    // Adjust the current index, if necessary.
    if let currentIndex = playlist.currentIndex, index <= currentIndex {
        playlist.currentIndex = currentIndex + 1
    }

    // Let the current item continue playing.
    continueWithCurrentItems()
}
```

Important

All methods of `SampleBufferPlayer` that modify the `Playlist` must do so in a thread-safe manner. In this example implementation, a `DispatchSemaphore` guarantees that only one method modifies the state at a time. Such methods issue a semaphore `wait` on entry, then use a `defer` statement to guarantee a matching semaphore `signal` on exit.

In general, all of the public methods of `SampleBufferPlayer` end up invoking one of these helper methods: `restartWithItems(fromIndex:atOffset:)` or `continueWithCurrentItems()`.

The `restartWithItems(fromIndex:atOffset:)` method forces playback of the currently playing item (if any) to stop before playback restarts with a new list of items:

```
private func restartWithItems(fromIndex proposedIndex: Int?, atOffset offset: CMTime
    // Stop the player if there's no current item.
    guard let currentIndex = proposedIndex,
        (0 ..< playlist.items.count).contains(currentIndex) else { stopCurrentItems(

    // Start playing the requested items.
    playlist.currentIndex = currentIndex
    let playbackItems = Array(playlist.items[currentIndex ..< playlist.items.count])

    playbackSerializer.restartQueue(with: playbackItems, atOffset: offset)
}
```

The `continueWithCurrentItems()` method allows the currently playing item to continue to play, followed by a new list of items that plays after the current item finishes:

```
private func continueWithCurrentItems() {
    // Stop the player if there's nothing to play.
    guard let currentIndex = playlist.currentIndex else { stopCurrentItems(); return

    // Continue playing with a list of items to play starting from the current item.
    let playbackItems = Array(playlist.items[currentIndex ..< playlist.items.count])

    playbackSerializer.continueQueue(with: playbackItems)
}
```

Both methods begin by checking that the player state isn't "stopped." They then construct a queue of items to play — which may consist of fewer items than the entire playlist — and pass the queue to a corresponding method in the `SampleBufferSerializer` class. This transfer of control takes place on a serial `DispatchQueue`, so that the `SampleBufferSerializer` object handles one action at a time.

Schedule playback

After the `SampleBufferSerializer` object receives a queue of items to play, it proceeds with the dual tasks of translating the items into a sequence of sample buffers containing audio data and enqueuing the buffers for rendering. The `SampleBufferSerializer` causes an `AVSampleBufferRenderSynchronizer` object to play audio at the correct time, and an `AVSampleBufferAudioRenderer` object to render enqueued audio sample buffers in time for playback.

> **Important**
>
> `SampleBufferSerializer` invokes all of its internal methods via a serial `DispatchQueue`. This ensures that changes to instance properties during each method invocation are thread-safe.

The most important `SampleBufferSerializer` methods are the two methods it uses to accept control from the `SampleBufferPlayer` object: `restartPlayback(with:atOffset:)` and `continuePlayback(with:)`. Both methods take, as their first parameter, a queue of items to play in order. The methods differ in the way they handle the item that was previously playing, if any.

The `restartPlayback(with:atOffset:)` method stops any current playback, which means that the audio renderer can simply flush all enqueued buffers, and restart enqueuing from the first provided item.

By contrast, `continuePlayback(with:)` attempts to let the current playback continue, and allow enqueued buffers to remain enqueued, as far as possible. It examines its new list of items and finds ones that match previously scheduled items. It can then do a partial flush of the audio renderer, starting from the playback time of the first nonmatching item, and resume enqueuing sample buffers from that point.

The time-based partial flush uses asynchronous API with a completion handler. Upon completion, enqueuing actually restarts in an additional method, `finishContinuePlayback(with:didFlush:)`. This division into a pair of methods is an implementation detail.

Enqueue buffers

As it converts its list of items into a sequence of sample buffers, the `SampleBufferSerializer` needs to enqueue the buffers to an `AVSampleBufferAudioRenderer` object. Control of enqueuing relies on the `requestMediaDataWhenReady(on:using:)` method of `AVSampleBufferAudioRenderer`, which takes a closure parameter that the renderer invokes whenever it's ready for more sample buffers.

For a single playlist item, the sequence of events is straightforward, but when there are multiple playlist items, the `SampleBufferSerializer` may need to enqueue sample buffers from a second (or subsequent) item before the first item finishes playing. That means, usually, that enqueuing of sample buffers takes place well in advance of the playback time of those buffers. This sequence follows these steps:

1. The serializer queues sample buffers as early as possible, when requested by the `AVSampleBufferAudioRenderer`.

2. After enqueuing the last sample buffer of a playlist item, the serializer places a boundary observer on the `AVSampleBufferRenderSynchronizer` timeline, at the expected ending playback time of that item.

3. When the boundary observer fires, playback of that item is complete. The serializer discards the item, removes the boundary observer, updates the current item in the `SampleBufferPlayer`, and generates a notification that it uses to update the current item display in the UI. It also places a periodic observer on the timeline, which fires every 0.1 seconds, to generate future timing notifications for updating the playback time display in the UI.

4. This process of placing boundary observers, and removing them when they fire, repeats as each item finishes enqueuing its buffers.

In the implementation, the important methods are `provideMediaData()`, which enqueues sample buffers and places boundary observers, and `updateCurrentPlayerItem(at:)`, which the boundary observer invokes to handle the transition between items.

Retrieve sample buffers

A `SampleBufferItem` object provides the `SampleBufferSerializer` with a sequence of audio sample buffers for a playback item:

```
// Try to read from a sample buffer source.
let source = sampleBufferSource!
let sampleBuffer = try source.nextSampleBuffer()

// Keep track of the actual duration of this source.
endOffset = source.nextSampleOffset

return sampleBuffer
```

The `SampleBufferItem` object also manages state associated with the playback item. This includes a `uniqueID` property that identifies the item uniquely within the playlist, even when items share the same underlying `PlaylistItem`.

It also includes an `endOffset` property, which is the time — relative to the start of the item — when playback of buffers enqueued so far ends. This value is important for determining the placement of the boundary observers described in the previous section, on the `AVSampleBufferRenderSynchronizer` timeline.

A `SampleBufferItem` object keeps a reference to the source of its audio data while enqueueing the data. The audio data source is represented by a `SampleBufferSource`, which is an object that you customize for your audio data.

As soon as `SampleBufferItem` enqueues all data for the item, the item can discard its data source object, allowing the system to reclaim its resources (files and memory, for example).

Provide your data source

Ultimately, you need to provide custom data source code to fetch your custom audio data and package it into `CMSampleBuffer` objects that you can pass to the audio renderer.

In this example project, a `SampleBufferSource` object serves as the data source. It simply reads data from an audio file stored within the app bundle.

The class also contains helper methods that convert an `AVAudioBuffer` to a `CMSampleBuffer`, which is the required type for data passed to the audio renderer.

Configure logging

The player and serializer implementations contain detailed, formatted logging of their actions. The logging output can be crucial to understanding the behavior of the code during development and testing.

By default, logging suppresses messages about the enqueuing of specific sample buffers, to avoid flooding the console. If desired, you can enable those messages by setting `shouldLogEnqueuerMessages` to `true`.

See Also

Playback

{} Using voice processing

Add voice-processing capabilities to your app by using audio engine.

`class AVAudioPlayerNode`

An object for scheduling the playback of buffers or segments of audio files.