Sample Code

Performing offline audio processing

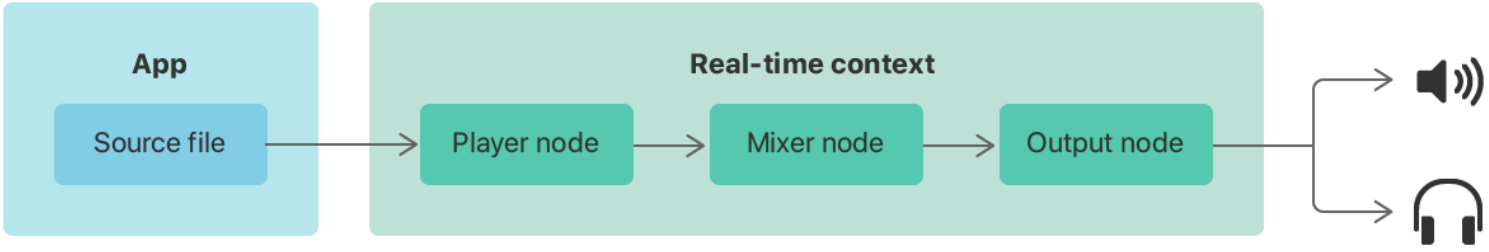
Add offline audio processing features to your app by enabling offline manual rendering mode.

Download

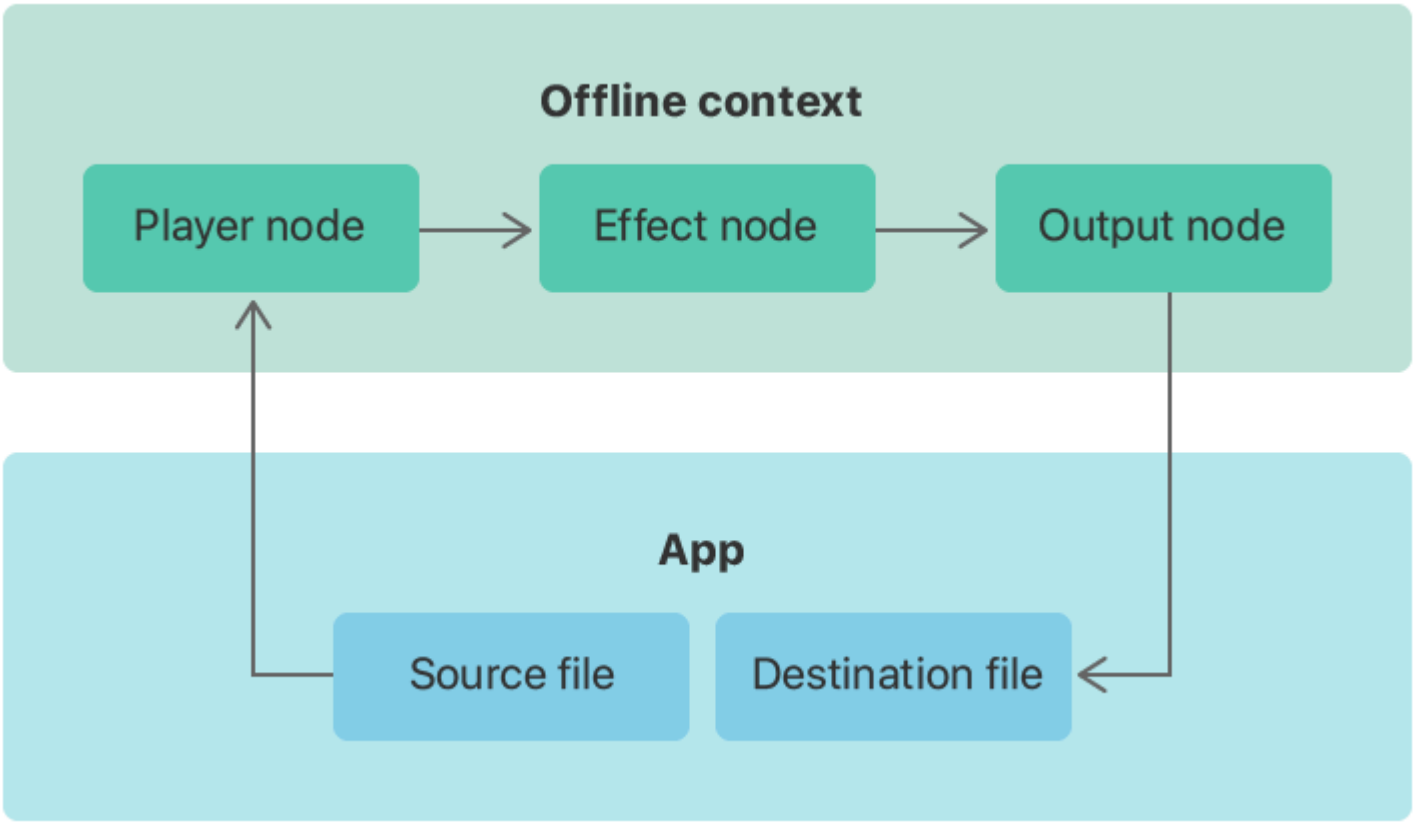
Xcode 15.4+

Overview

You commonly use `AVAudioEngine` to add advanced real-time audio playback features to your app. In a real-time scenario, the audio hardware drives the engine's I/O and renders the data to the output hardware, such as the device's built-in speaker or connected headphones.



You can also use `AVAudioEngine` to perform *offline* audio processing by enabling the engine's offline manual rendering mode. In this mode, the engine's input and output nodes are disconnected from the audio hardware and the rendering is driven by your app. You use offline manual rendering mode to perform advanced postprocessing tasks, such as applying effects or performing audio analysis, usually much faster than you can do in real time.



This sample playground shows you how to enable the audio engine's manual rendering mode and drive the rendering process from your app.

Prepare the Source Audio

The sample loads its audio data from a file contained within the playground. It creates an `AVAudioFile` object to wrap the on-disk file and retrieves its input format, which is used to configure later stages of the audio pipeline.

```
let sourceFile: AVAudioFile
let format: AVAudioFormat
do {
    let sourceFileURL = Bundle.main.url(forResource: "Rhythm", withExtension: "caf")
    sourceFile = try AVAudioFile(forReading: sourceFileURL)
    format = sourceFile.processingFormat
} catch {
    fatalError("Unable to load the source audio file: \(error.localizedDescription).
}
```

Create and Configure the Audio Engine

The sample creates the engine and configures it to perform the desired processing. It creates a player node to drive the input, feeds its output into a reverb node, and connects the output of the

reverb node to the main mixer node (which is implicitly connected to the output node). Finally, it schedules the audio file for playback.

```
let engine = AVAudioEngine()
let player = AVAudioPlayerNode()
let reverb = AVAudioUnitReverb()

engine.attach(player)
engine.attach(reverb)

// Set the desired reverb parameters.
reverb.loadFactoryPreset(.mediumHall)
reverb.wetDryMix = 50

// Connect the nodes.
engine.connect(player, to: reverb, format: format)
engine.connect(reverb, to: engine.mainMixerNode, format: format)

// Schedule the source file.
player.scheduleFile(sourceFile, at: nil)
```

Enable Offline Manual Rendering Mode

If you started the engine at this point, you'd hear the audio playing through the active output device. To change this default behavior, you need to explicitly configure the engine to use manual rendering mode. The sample enables this mode by calling the `enableManualRenderingMode(_:format:maximumFrameCount:)` method, passing it the offline rendering mode value, the audio format, and the maximum number of frames to render in a given pass of the render thread.

```
do {
    // The maximum number of frames the engine renders in any single render call.
    let maxFrames: AVAudioFrameCount = 4096
    try engine.enableManualRenderingMode(.offline, format: format,
                                         maximumFrameCount: maxFrames)
} catch {
    fatalError("Enabling manual rendering mode failed: \(error).")
}
```

With the configuration complete, the sample starts the engine and tells the player to play.

```
do {
    try engine.start()
    player.play()
} catch {
    fatalError("Unable to start audio engine: \(error).")
}
```

Prepare the Output Destinations

Unlike in a real-time playback scenario, the output from the audio engine isn't rendered to the output hardware, but is instead rendered to an output buffer and ultimately written to a file on disk. The sample creates the buffer object to render into, and an output file in your `~/Documents` directory to save the processed audio.

```
// The output buffer to which the engine renders the processed data.
let buffer = AVAudioPCMBuffer(pcmFormat: engine.manualRenderingFormat,
                              frameCapacity: engine.manualRenderingMaximumFrameCount

let outputFile: AVAudioFile
do {
    let documentsURL = FileManager.default.urls(for: .documentDirectory, in: .userDo
    let outputURL = documentsURL.appendingPathComponent("Rhythm-processed.caf")
    outputFile = try AVAudioFile(forWriting: outputURL, settings: sourceFile.fileFor
} catch {
    fatalError("Unable to open output audio file: \(error).")
}
```

Manually Render the Audio

The sample sequentially loops over the full duration of the input audio file, and in each pass, asks the engine to render the next batch of frames to the output buffer. If the audio engine successfully renders the data, the sample writes the buffer to the output file on disk. When the sample finishes processing the audio data, it stops the player and engine.

```
while engine.manualRenderingSampleTime < sourceFile.length {
    do {
        let frameCount = sourceFile.length - engine.manualRenderingSampleTime
        let framesToRender = min(AVAudioFrameCount(frameCount), buffer.frameCapacity

        let status = try engine.renderOffline(framesToRender, to: buffer)
```

```
        switch status {

        case .success:
            // The data rendered successfully. Write it to the output file.
            try outputFile.write(from: buffer)

        case .insufficientDataFromInputNode:
            // Applicable only when using the input node as one of the sources.
            break

        case .cannotDoInCurrentContext:
            // The engine couldn't render in the current render call.
            // Retry in the next iteration.
            break

        case .error:
            // An error occurred while rendering the audio.
            fatalError("The manual rendering failed.")
        }
    } catch {
        fatalError("The manual rendering failed: \(error).")
    }
}

// Stop the player node and engine.
player.stop()
engine.stop()
```

See Also

Rendering

{} Building a signal generator

Generate audio signals using an audio source node and a custom render callback.

class AVAudioSourceNode

An object that supplies audio data.

class AVAudioSinkNode

An object that receives audio data.