Audio Mixer Overview

An overview of the multi-platform audio-rendering API used for playing in-game sounds.

The Audio Mixer is a multi-platform audio renderer that lives in its own module. It enables feature parity across all platforms, provides backward compatibility for most legacy audio engine features, and extends Unreal Engine functionality into new domains. This document describes the structure of the Audio Mixer as a whole, and provides a point of reference for deeper discussions.

Background and Motivation

Audio Rendering

Audio rendering is the process by which sound sources are decoded and mixed together, fed to an audio hardware endpoint (called a digital-to-analog converter, or DAC), and ultimately played on one or more speakers. Audio renderers widely vary in their architecture and feature set, but for games, where interactivity and real-time performance characteristics are key, they must support real-time decoding, dynamic consuming and processing of sound parameters, real-time sample-rate conversion, and a wide variety of other audio rendering features, like per-source digital signal processing (DSP) effects, spatialization, submixing, and post-mix DSP effects such as reverb.

Platform-Level Features: Audio-Rendering APIs

Typically, each hardware platform provides at least one full-featured, high-level, audio-rendering, C++ API. Some platforms provide several options. These APIs often provide platform-specific codecs and platform-specific encoder and decoder APIs. Many platforms provide hardware decoders to improve runtime performance. In addition to codecs, platform audio APIs provide all the other features that an audio engine might need, including volume control, pitch control (real-time sample-rate conversion), spatialization, and DSP processing.

Game-Level Features: Gameplay APIs

Game engines write additional functionality on top of these platform-level features. For example, features might hook into a scripting engine (such as Blueprint) or game-specific components and systems (such as audio components, ambient Actors, and audio volumes), or do a lot of work to determine which sounds to actually play (such as Sound Concurrency) and with what parameters (such as Sound Classes, Sound Mixes, and Sound Cues).

The Problems with Platform-Specific Audio Rendering APIs

This paradigm works well when there is a small number of target platforms to support, and when there is a long lead time to ramp up new platforms. However, in the case of Unreal Engine where we support a large number of platforms, juggling differences between each platform-specific API, hunting platform-specific bugs, and striving for platform feature parity can easily dominate the audio engine development time. This is at the cost of writing new runtime features or development tools.

Since feature parity in this paradigm is not possible, feature support matrices need to be authored and maintained to communicate with quality assurance teams and sound designers regarding which features work on which platforms. Related to this, documentation and support is difficult, as not every feature will work on every platform. Mixing a game which is intended to ship on multiple platforms is difficult when not all features work on all platforms, and existing features sound different on different platforms.

Writing new audio rendering-level features needs to be implemented uniquely for every platform—or increase the platform disparity problem. Similarly, any optimization or bug fix

either requires duplication, or is isolated to specific platforms.

In most cases, new lower-level features, optimizations, and bug fixes are simply not possible. If a performance issue or known bug exists with a platform API, there is often no recourse to fix it without working with the owner of the platform. Bug fixes often consist of hacks or workarounds. If new features are even possible, they're often relegated to a narrow-platform plugin API, and won't work across platforms.

Implementing the platform-specific audio API on new platforms takes a significant amount of work, and requires a larger development lead time. In some cases, a brand new platform may not even have a full-featured audio API available. In those cases, establishing audio rendering on that platform would be impossible until the platform invested resources into writing their own audio rendering API.

One exciting avenue for any game engine is the ability for third parties to add new features and extensions to Unreal Engine. With a complex constellation of platform APIs and all the various ways they themselves might support plugin extensions,creating a common API that wraps everything is all but impossible.

The Audio Mixer Is the Solution

The solution to all of these issues is a single multi-platform, audio-rendering API—what we call the Audio Mixer. With the Audio Mixer, since there is one common code base for all platforms, feature parity is much more easily achieved. Development time is optimized as programmers are able to implement a new feature once and can expect it to work on all platforms. Testing, documentation, and sound designer implementation and mixing is also simplified. Apart from a subset of unavoidable and specific cases, things will sound the same and behave the same across platforms.

Supporting new platforms is fast—it often takes just a few days and a couple of hundred lines of code. Writing an interface for audio engine plugins becomes not only very doable but a primary avenue for innovation. In addition, fixing bugs and optimizing code benefit all platforms.

There will still remain platform-specific challenges—each platform has different CPU and memory capabilities, or different hardware support options. However, in cases of CPU or memory deficiencies, the Audio Mixer can employ features that automatically reduce CPU load (via feature disablement, quality reductions, and so on) or memory load (memory

pruning at cook, automatic downsampling and down-quality encoding). Sound designers and audio programmers can tune the Audio Mixer to hit a given performance and memory target for a given platform.

Why It's Called the Audio Mixer

Another word for audio rendering is audio mixing. Since the word rendering is used almost exclusively in Unreal Engine to refer to graphics, we decided to call the new audio renderer the **Audio Mixer**. However, this decision has sometimes caused confusion within the audio community and others due to the ambiguity of the word mixing. It can be confused with the operation of volume (loudness) mixing.

The Audio Mixer is essentially doing the same thing that a hardware mixing console does—it adds together sound sources after processing them through a variety of parameters and effects processors, including through submixing and master-effects processors.

The Audio Mixer Architecture

The Platform Layer

The Audio Mixer has a minimal platform-specific API layer. The platform API wraps all the necessary details of accessing audio hardware for our various platforms, and this layer deals with querying hardware capabilities and, where needed, hardware state changes, and sets up the multi-platform Audio Mixer to feed audio to the hardware.

Some platforms require additional code to handle a variety of platform-specific nuances like state interruptions, app suspension, device swapping, and so on; while in other cases, support was added to leverage platform-specific enhancements like hardware-accelerated decoding. In these cases, there can still be a good deal of platform-specific code required. Note that this platform-specific layer is also dealing with some of our remaining platform-specific features in Unreal Engine, like creating runtime decoders for our various codecs.

Buffer Generation

Different platforms have different methods of feeding audio to its hardware. Some platform APIs are designed to push audio to the hardware into a queue, and the client app is responsible for keeping audio queued up. Other APIs are callback-based, and will call into client code when the hardware needs more audio. In both cases, the Audio Mixer employs a multi-buffer scheme that generates future audio buffers while hardware is rendering the current buffer (such as the buffer you are listening to).

If the Audio Mixer takes too long to render the next buffer of audio, and the buffer queue starves (for push APIs), or if there is no audio available when the API calls back (for callback APIs), there will be an audible gap in the playback. This is called an underrun (sometimes called starvation), and should be avoided at all costs as it always sounds very bad. An underrun will cause a sudden discontinuity in the audio stream, and is perceived as a noise-burst—either as a pop for a short underrun, a stutter for short and persistent underruns, or a major drop-out of audio for long-running underruns. The sudden amplitude change can even cause damage to audio receivers or speakers. In most cases, underruns are due to CPU saturation—the Audio Mixer is doing too much work in the time it has. In other cases, it might indicate a blockage in the task graph (for async decoding or synthesis), or some other problem. It's also sometimes difficult to distinguish a click or pop in the audio from an underrun or vice versa.

On the other end of things, it's possible to generate too much audio up front. In these cases, game events will lag further and further from real time, and this is perceived as latency. In extreme cases, this is called an overrun, and is also something to avoid. Real-time audio engines need to find the balance between underrunning and overrunning. Underrunning is avoided at all costs, while overrunning, which is required to some degree to prevent underrunning, needs to be below the threshold of perception.

Finding the right balance can be tricky, and is often platform dependent. To that end, the Audio Mixer is architected to allow sound designers (or audio programmers) to choose, per platform, the size of the audio buffers to render per audio buffer render, and how many buffers to render ahead of the currently audible buffer.

The Audio Mixer Threading Model

The Audio Mixer creates its own thread to do the actual audio rendering. We call this the **audio render thread**. This thread is distinct from the **audio thread**, and is the thread that does all the work for DSP source generation and mixing.

The audio thread is the thread that does the work to determine what sounds to play and what their parameters are (processing sound cues, sound classes, sound mixes, attenuation,and so on). The audio thread is largely identical between the Audio Mixer and the legacy platform-specific audio engines. The audio thread currently has some complexities to how it deals with UObject and garbage collection. Since the audio thread has UObjects, it is fenced and halted during garbage collection. During runtime, these UObjects are read-only. However, in the editor, since these UObjects are writable, we don't use the audio thread.

In callback-based platform APIs, there is also a hardware thread that the actual platform hardware callback is made from. This hardware-owned thread callback nearly always pops off the rendered buffer that was queued up from the audio render thread, and does very little actual work.

Finally, the audio engine utilizes async tasks from the task graph for decoding and procedural audio (for example, synthesis) generation.

Generating Sound Sources

The first stage of the audio mixer render itself is source generation. To generate sources, the Audio Mixer receives parameters sent from the game, and from the audio thread, that define what sound sources to play, along with their parameters (such as volume, pitch, and position).

Here we use the word source because the audio generated for that source can be derived either from a compressed audio asset (such as a sound file), procedurally generated (by synthesis, decoded media, VOIP, or microphone capture), or source-derived from other sources mixed together (for example, a **source bus**). A different method is used to generate the source data for each case.

Encoded Sound Sources

In the case of encoded sound sources (ogg-vorbis, opus, atrac9, xma2, adpcm, and so on), an asynchronous task is used to decode the audio into uncompressed 32-bit float buffers. This 32-bit float buffer is then sample-rate converted (SRC) from the source's sample rate to the sample rate of the Audio Mixer. This SRC process takes into account any *pitch scale* that might have also been applied to the sound source, and does the SRC only once.

For example, if a sound source is encoded at 32 kHz and has a pitch scale of 1.2 because the sound designer wants to pitch up the sound source, and if the Audio Mixer is rendering audio at 48 kHz, the SRC will use the following sample-rate ratios:

```
SampleRateRatio = (48 kHz/32 kHz) * 1.2

SampleRateRatio = 1.8
```

Copy full snippet

This means that it will pitch up the sound source by 1.8 after the audio has been decoded.

Procedural (Generated) Sound Sources

In the case of procedural sound sources, the audio is generated through an abstract interface that calls back into client code to generate the next 32-bit float buffer. Client code can feed any audio into this callback, and the audio could theoretically be from any source. In UE4, we use this to do real-time synthesis, render audio from media sources (such as play audio from video in 3D, including attenuation, occlusion, or with real-time effects), or play audio from VOIP.

This is also one of the many ways that third-party plugins can extend the audio engine. Procedural sound sources can be defined in a plugin, and source audio can be fed into the Audio Mixer from any other source, including entirely independent sound engines.

Mixed Sources

The Audio Mixer also has support for mixing sound sources together to create other sources. These mixed sources are called **source buses**. Source buses are treated like any other source, and support most of the features that other sources have, such as spatialization, distance attenuation, volume mixing, and source-effects processing. There are a number of useful applications for this feature: dynamic routing of sound sources to new spatial locations (such as radio broadcasting), spatialized effects processing (such as location-based reverb processing), and delay-type spatial effects (such as bouncing audio off specific objects in a scene).

Sound Source DSP Effects Processing

Once a sound source has generated its audio, it's output is fed through a chain of **audio DSP effects**. There are some built-in effects that all sound sources automatically have access to. These DSP effects are a high-pass filter and a low-pass filter, and are used for a variety of higher-level features.

Through the **sound-attenuation settings**, sound designers can map filter frequency cutoffs of high-pass and low-pass features to apply to individual sound sources as a function of distance from the listener.

Another feature that utilizes this per-source filter is the built-in **occlusion system**. If occlusion is enabled, when a sound is obstructed behind occluding geometry (determined via async ray traces), it will automatically apply a sound designer-defined low-pass filter. Finally, sound mixes and sound classes can apply per-source, low-pass filtering to sounds.

Beyond the automatic built-in filter effects, sounds can also feed themselves through what is called a **source effect chain**. This is an asset that defines a sequence of per-source DSP effects that are authored via a source-effect plugin API and can be anything: chorus effects, flangers, delays, advanced filters, ring modulation, bit crushing, and so on.

The Submix Graph

After source generation, the Audio Mixer then processes the **submix graph**. The output of the submix graph is the audio that is heard on audio hardware.

Sending To Submixes

A **master submix** is created in the audio engine when the audio engine is initialized. By default, all sound sources are automatically registered to this submix as its base submix if no other submix is specified as the base submix.

A source's base submix is the submix that mixes the source output at full volume (post-attenuation, post-spatialization. Any number of send submixes can also be specified, which allows mixing sources together before sending through auxiliary DSP effects. These send submixes are analogous to auxiliary channels for DSP effects like reverb in a digital audio workstation or mixing console.

As discussed earlier, there is default routing that happens to a sound source that does not specify any submix routing. However, there are two primary means of controlling where the sound source sends its audio to submixes. The first is to specify the base submix. This submix mixes the sound at full-volume. If left blank, it will route the audio to either the EQ submix or the master submix, depending on the legacy features.

Master Submixes

In addition to the master submix, two master-effect submixes are created and added as children to this master submix. These are the master reverb submix and the master equalizer (EQ) submix. These master submixes were created primarily for backward compatibility with legacy features in Unreal Engine, and to aid in swapping existing projects to the Audio Mixer. However, they are the underlying mechanisms that support the various higher-level features in Sound Classes, Sound Mixes, Audio Volumes, and Sound Attenuations that interact with reverb and EQ.

Generating Output

The master submix is the root node of the submix graph. The final output is generated by retrieving the audio generated from this master submix.

The output of each submix object is generated by first generating the output of the submix's child submixes, then mixing this output with any sources that have been registered (or sent) to it. The send amount of the registered sound sources are a gain/attenuation on the sources mixed together in the submix.

This mixed buffer is mixed to the output channel configuration for the hardware. For example, for stereo output, this would be a 2-channel audio buffer. For 7.1 audio, it would be an 8-channel audio buffer.

This mixed, interleaved audio is then fed through the submix's DSP effect chain to generate a final output.

Additional Submix Features: Analysis, Recording, and Listening

Submixes are the ideal object for performing audio analysis. Currently, there is a mechanism to register Blueprint delegates, which retrieves amplitude envelope values from the submix, per channel. This could be useful for a number of visualizations or gameplay systems that want to interact with audio. There is also a real-time FFT delegate to retrieve spectral data from submixes in Blueprint.

Submixes also have a feature to enable recording their output to either a USoundWave object or to a raw PCM (`.wav`) file to disk.

Finally, there is a C++ delegate that can be registered by other code systems to retrieve the mixed audio from any submix dynamically. This functionality can be used for a number of useful features, like internet broadcasting (for server-based streaming of a game) or third-party plugin extensions.