Audio FX Toolkit

Audio effect prototyping for Unity

Release 1.0

Overview:

The Audio FX Toolkit is a collection of classes and example scripts that are meant to be used as building blocks for user-designed audio effects. This guide will give you an overview of how to best utilize these scripts.

API Documentation:

In depth Doxygen generated API documentation on the DSP can be found here: https://blackwoodsounddesign.github.io/AudioFXToolKitdocs/index.html

Essential Background:

Audio files, from a computer's perspective, are a collection of floating-point numbers ranging from -1 to 1. These numbers correspond to speaker vibration and create sound when sent to a digital to audio converter (DAC).

To create audio effects, code is pulled from the practice of digital signal processing (DSP for short). This code acts individually on each floating point number from the .wav, .mp3, .oog, etc. Writing this can be a daunting task! Luckily, a lot of low-level DSP is included with this package, and unless you're doing something very specialized, you should find the code you need here.

File Structure Overview:

Inside of /AudioFXToolkit there are three folders:

/AudioFXToolkitDSP
/DemoResources
/Example_Usage_Scripts

/AudioFXToolkitDSP holds the raw DSP code that makes the audio effects function. Each audio effect is a separate class and can be easily transferred to other applications, environments, and languages (C, C++, etc.). None of these function on their own. The classes are meant to be implemented as objects. While this may seem counterintuitive, this lets these classes be expanded upon and reused.

/DemoResources holds the resources in the included demo: audio files, GUI scripts, the demo *Scene*.

/Example_Usage_Scripts holds scripts that showcase easy to complex implementations of the AudioFXToolkitDSP.

Effect Scripting Overview:

The Audio Effects Toolkit runs on a collection of classes in the AudioFXToolkitDSP namespace. The classes included range from low-level filters to high-level audio effects. These classes are meant to be used in the "process block" found in the OnAudioFilterRead() method. This is useful for quick prototyping of audio effect ideas and implementations. *click OnAudioFilterRead() to be taken to its official Unity documentation.

The implementation of these classes is a simple process. A good example script to start with is the SimpleFilter_Example.cs. Here is its breakdown:

This section sets up the variables to control the audio effect.

```
using UnityEngine;

public class SimpleFilter_Example : MonoBehaviour
{
   [Header ("Lowpass Filter")]
   [Range(0.0f, 20000f)]          ← This creates a slider in the inspector to control the variable directly below it.

   public float filterF;          ← A variable to store the cutoff frequency.
```

To use the classes, create a new object as below. This can be thought of as a guitar pedal, a Max/MSP object, etc. This will be used in the process block of OnAudioFilterRead().

```
AudioFXToolkitDSP.SimpleFilter simpleFilterL = new AudioFXToolkitDSP.SimpleFilter();
                                                ↑ Creates a filter for the left channel
AudioFXToolkitDSP.SimpleFilter simpleFilterR = new AudioFXToolkitDSP.SimpleFilter();
                                                ↑ Creates a filter for the right channel
```

Filters need the sample rate of the audio to function properly. This is required in the SetFilterParameters().

```
private void Awake()
{
   sample_rate = AudioSettings.outputSampleRate;     ← Gets the sample rate from the AudioSettings object.
}
```

Lastly, the process block. The process block is where each audio sample is individually processed to create the audio effect. This is a structure in all audio programming, even stand-alone applications. To do this in Unity, use the OnAudioFilterRead() method. This adds a custom filter into the audio DSP chain.

```
private void OnAudioFilterRead(float[] data, int channels) ← These parameters are automatic to this Unity method
  {
    //makes sure the audio is stereo, if not, return
     if (channels != 2)
       return;

    int dataLen = data.Length;

    int n = 0;

    simpleFilterL.SetFilterParameters(filterF, sample_rate);
    simpleFilterR.SetFilterParameters(filterF, sample_rate);

    //process block, this is interleaved
    while (n < dataLen)
    {
       //pull out the left and right channels
       int channeliter = n % channels;

       if (channeliter == 0)
          data[n] = simpleFilterL .Filter(data[n]);
       else
          data[n] = simpleFilterR.Filter(data[n]);

       n++; ← This increases the iterator by 1 for each sample filtered through the AudioFXToolkitDSP.SimpleFilter.
  }
}
```

The data float array holds the individual audio samples. channels is the number of audio channels (ie. stereo is 2, mono is 1, quad is 4, 5.1 is 6, etc.)

dataLen is the length of the data float array. This is used to iterate over the audio in data.

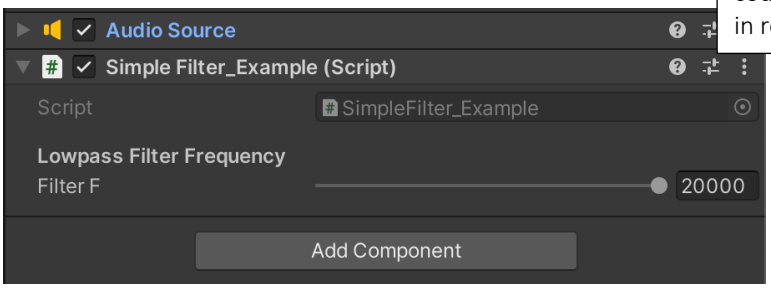
n is the iterator for the data array

SetFilterParameters() can be called outside of OnAudioFilterRead() and it is best practice to do so.

The channeliter modulo goes between 0 and 1. 0 is the left channel and 1 is the right. This pulls the left and right channels out from each other individually.

simpleFilterL.Filter(data[n]) passes the audio through the simple filter class in AudioFXToolkitDSP. It returns the filtered float value, and we set it the output (data[n]) to this new and improved value.

The inspector view of the above code. The filter cutoff is adjustable in real time with this slider.



For a more complicated implementation, reference the Demo_DSPChain.cs script in /AudioFXToolkit/DemoResources. The structure is largely the same, although the audio is passed through a waterfall of DSP Modules to create a more involved effect.

Quick Class Reference:

These classes are the backbone of the DSP. Most of the classes have example implementations here: /Example_Usage_Scripts. Use them in combination. Get creative!

AllpassFilter.cs
: An allpass filter built around a single circular delay line. It can be thought of as an input "smearer." This can be used to create reverbs, phasors, and other interesting effects.

BandPassFilter.cs
: A biquad based Bandpass filter. This filters out/attenuates the high and low end surrounding a desired frequency.

BiquadFilter.cs
: This filter is used as a building block throughout the various filters.

DelayLine.cs
: A mono circular delay line. This class contains a tap delay, sample delay, and a feedback delay. All of the delays use a linear interpolation to achieve fractional delay. This means the value of the delay can be in-between two samples.

Distortion.cs
: Two simple distortion algorithms: Clip and Soft.

FeedbackCombFilter.cs
: A feedback comb filter built around a single circular delay line. In this case, the delay line will feedback onto itself. This is useful for simulating spaces, reverbs, phasors, and other interesting effects.

FeedforwardCombFilter.cs
: A feedforward comb filter built around a single circular delay line delay line utilizing only the input (vs feedback utilizing the output). This is useful for simulating spaces, reverbs, phasors, and other interesting effects.

HighpassFilter.cs
: A biquad based highpass filter. This filters out/attenuates low end signals.

HighShelfFilter.cs
: A biquad based highshelf filter. This boosts/attenuates the high end of the signal.

LowPassFilter.cs
: A biquad based Lowpass filter. This filters out/attenuates high signals above the cutoff frequency.

LowShelfFilter.cs
: A biquad based lowshelf filter. This boosts/attenuates the low end of the signal.

PeakNotchFilter.cs
: A biquad based peaknotch filter. This boosts a configurable part of the spectrum. This can be used to create multiband EQs, etc.

RMS.cs

This returns the root-mean-square of a desired signal. Generally, RMS can be thought of as a good way of measuring loudness. This can be used for compressors, limiters, multiband dynamics, etc.

SimpleFilter.cs

This is a simple one pole filter, set up to attenuate the high end of the signal. Commonly this is referred to as a lowpass filter.

SimpleReverb.cs

The simplest reverb worth anything is this Schroeder verb. Generally, algorithmic reverbs are done by using this method, a feedback delay network, or a combination of.

Tremolo.cs

A classic tremolo effect. This is done be modulating the input audio signal by an oscillator. This is known as Amplitude modulation.

Wavetable.cs

This class can be set up to create any kind of repetitive signal but is mostly used to create oscillators. These signals can be used as modulators in phasors, tremolos, stored signals in ffts, etc.

Extending the DSP:

After implementing a number of these algorithms and then making your own combination therein, you may want to reuse the algorithm repeatedly. This makes your effect a good candidate for being moved into its own class. A good template for how to do this is the SimpleVerb.cs class. This uses three AllpassFilters and four FeedbackCombFilters. For each SimpleVerb object created, it does all of the allocation and delay times under the hood. You can easily create an effect based on these classes similarly.

Launching a Title with a Custom Audio Effect:

Generally, if you have something working and you want to implement it in your game permanently, it's best practice to create an audio plugin using the UnityAudioSDK. The Audio FX Toolkit classes will get you up and running to quickly prototype and test your ideas, but they can slow down performance if they are used in production. In development, the tradeoff is that you can edit these classes directly in the IDE linked to Unity. This saves you the trouble of having to compile a new binary for each effect update and iteration.

Feature Requests and Support:

Feature requests, connections, and bug reports can be asked here:
https://github.com/Blackwoodsounddesign/AudioFXToolKitdocs/discussions