4-22-2016

Audio Software (VST Plugin) Development with Practical Application

Zachary Hummel
Western Michigan University, zphummel@gmail.com

Audio Software (VST Plugin) Development with Practical Application

Zachary Paul Hummel

Honors Thesis submitted in partial fulfillment of the requirements of the Lee Honors College, Western Michigan University

Thesis Committee:

Dr. Richard Johnson, Chair

Matthew Layher

Objective:

The objective of this project was to increase my own understanding of digital signal processing (DSP) algorithms, with a focus on applications related to digital audio. There are two primary phases to complete in the pursuit of this objective. The first is to study and design DSP algorithms that accomplish desirable manipulation of an incoming audio signal. The second phase is to implement these algorithms in small pieces of software known as "plugins" for use in DAWs, allowing the user to have access to the functionality of these algorithms within the confines of an intuitive graphical user interface (GUI).

Method:

The DSP algorithms that I used in this project (whether original or derived from an outside source) will be documented in mathematical notation wherever possible to establish a consistent frame of reference. Where mathematical notation is not possible or not especially helpful for understanding, source code or a visual reference will be provided instead.

I decided to implement this project as a set of Virtual Studio Technology (VST) plugins. VST is a format created by Steinberg GmbH ("3rd Party Developer"). It consists of a set of C++ libraries known as the VST SDK, which allow a user to create audio plugins that can be used inside of most major DAWs. The VST SDK is available free of charge, making it an ideal choice for this project, which had no budget for purchasing software. I considered other plugin formats such as Apple's Audio Units (AU) and Avid Audio Extensions (AAX) when first starting this project, but AU plugins can only be used on Mac systems, and developing AAX plugins requires a developer's license from Avid (the resultant plugins can also only be used in Avid's Pro Tools software) ("AAX Connectivity Toolkit"). Overall, the VST SDK proved to be by far the most useful format available free of charge for developing plugins.

Unfortunately the VST SDK is quite difficult to use in its native state due to its complexity. To make things easier for myself, I implemented JUCE, an extremely useful tool designed to "[d]evelop powerful, cross-platform audio, interactive, embedded or graphic applications" ("Features"). In practice, JUCE generates the necessary APIs for programming a VST plugin, and the user fills in the empty functions to describe the plugin's desired behavior. In this way, JUCE can drastically reduce the complexity of interfacing directly with the VST SDK. JUCE is available to developers free of charge. It does not contain its own compiler, however, so it must be coupled with an external program such as Microsoft Visual Studio. In my case setting Visual Studio 2012 as the desired compiler in the JUCE project settings worked seamlessly. Projects then gain the ability to "Open in Visual Studio 2012," where they can be debugged and compiled. JUCE contains an excellent drag-and-drop GUI creation system, which

also builds its own API for the user to fill in with the GUI's behavior and to pass data to and from the guts of the plugin (what JUCE refers to as the "Processor").

I chose to build all of the VST plugins for this project as 32-bit, stereo-only versions to simplify the project setup. It would be trivial to release them as 64-bit builds, as JUCE contains the ability to create different release builds from the same or similar source code. Some minor modifications would be required in order to release mono versions, although for most DAWs this is just a formality, since stereo-only versions are often usable even on mono audio tracks. This project in its entirety has been uploaded to GitHub at https://www.github.com/BurningCircus as open-source software.

Background:

Audio engineers around the world routinely utilize small pieces of software, known as "plugins," to help shape the sound of the media they are working with. These plugins are inserted into a digital audio workstation (DAW) and contain digital signal processing (DSP) algorithms that can affect audio in a number of ways limited only by the creativity of the software developer and the speed of the computational system. Common implementations of plugins include equalizers, compressors and limiters, gain, phase rotation or polarity inversion, reverb and delay, and emulation of the non-linearities of analog audio equipment (Shambro).

The DSP algorithms that form the heart of audio plugins draw their roots from the mathematical modeling used by electrical engineers to create analog filters on circuit boards for use in audio processing hardware (Smith). Digital processing, however, offers substantial improvements over the abilities of analog circuits, especially in the time domain. While an analog circuit must take in an input voltage and produce an output voltage in real time with no noticeable latency, a digital processor can look ahead in the digital audio stream and/or buffer input samples in order to predict what processing must be done in advance. This enables software plugins to avoid many technical compromises inherent in the analog domain, and is why I chose to develop software instead of hardware audio processing devices.

While audio may be synthesized digitally, much of the source material used by audio engineers in media creation comes from an acoustic or analog electronic source. Since audio in its natural state is composed of pressure variations in a medium and does not inherently contain any digital information, some attention must be paid to the process used to extract digital information from the source signal.

For an acoustic source, the sound is captured by one or more transducers (most often microphones or pickups) and converted in real time into a small AC voltage in an electrical circuit which

is analogous (although not identical) in amplitude, frequency, and phase content to the source. Analog synthesizers and playback devices also generate a small AC voltage, but require different transducers (or no transducers at all) to do so. This voltage is most often present at or near "line level," a nominal reference level that equates to roughly $1.25V_{RMS}$ (Heiduska).

Specialized hardware components called analog-to-digital converters (ADCs) are used in practice to accept these line level electrical signals and convert them into digital information using a process called sampling. Sampling involves the ADC determining the electrical voltage at its input and converting this value to a corresponding digital value, represented in the digital system as a signed integer. This process occurs many times per second at a rate determined by the sampling frequency f_S. Common sampling frequencies are multiples of 44.1kHz or 48kHz. Digital values are represented most frequently as 16- or 24-bit integers, where the maximum integer value corresponds to the maximum amplitude which can be represented by the digital system. 16- and 24-bit systems correspond to 2^{16} and 2^{24} possible amplitude values, respectively. Because integers cannot contain fractional amplitude information, the incoming analog signal must be quantized to the closest possible integer amplitude representation in the digital domain, thereby introducing a small amount of error. Thus it can be seen that a larger bit depth per sample increases the number of possible amplitude representation values on an exponential curve, and that higher bit depths which can more accurately represent incoming signals cause less error to be introduced in the digital signal. In practice, a higher bit depth corresponds to a lower noise floor generated at the ADC stage, which is a highly desirable characteristic of digital audio files. The primary disadvantage to higher bit depths is the increase in required disk space to store the recorded digital audio. Due to the steadily declining costs of digital storage, most audio engineers perceive the benefits of high bit depths as outweighing the disadvantages, and thus choose to work at the highest bit depth that their ADCs allow (Park 13-38).

Audio samples pass through a few layers of software before reaching a plugin where they can be manipulated. Most often, audio is recorded in a pulse code modulation (PCM) format such as a .WAV or .AIFF file on a digital system and played back by pulling blocks of samples (known as "buffers") from the file and passing the resultant samples to the playback system individually at a rate defined as the time interval $t_S = 1\ \mathrm{second}/f_S$. Plugins are implemented as part of a real time signal processing chain before the playback system, which means that they are passed buffers from the host (usually a DAW) as the input, and must output a processed version of the input buffer with the same number of samples within a set amount of time. Computational processing is not instantaneous, but the time allotted to complete the processing is substantial in relation to the amount of time required by most systems. It is defined as

t_S multiplied by the number of samples in the buffer. It is important to note that t_S varies according to the sampling rate f_S. It can be clearly seen that the higher the sampling rate, the lower the allotted time interval for a plugin to complete its processing. This limits the amount of processing that can be applied to incoming audio and still be able to generate output at a pace fast enough to maintain a real-time stream of buffers back to the host. In practice, these limitations are rarely reached except when attempting the most complex of signal processing tasks. Such processes are often implemented in non-real time, allowing them to be applied to an existing audio file without the burden of outputting buffers fast enough to keep up with a playback system. For the sake of this project, all algorithms discussed will be implemented in real time.

The amplitude information that plugins receive from the host system is also formatted differently than the data that is output from the ADC and stored as PCM audio files. For each audio sample, rather than an integer value in two's complement form where a sample falls in the range $2^{B-1} - 1 \leq x[n] \leq 2^{B-1}$ (for bit depth *B* and input sample $x[n]$), plugins receive all amplitudes as floating point values such that $-1.0 \leq x[n] \leq 1.0$ (techopedia). This has some significant advantages for simplifying signal processing. For instance, the developer is relieved of the responsibility to quantize every sample to the nearest integer value before outputting it (since a floating point value can contain fractional amplitudes). Basic operations such as polarity inversion are dramatically simplified as well. In fact, inverting the polarity of an input signal becomes as simple as outputting $-x[n]$. There are some additional advantages with the floating point representation as related to trigonometric functions (some of which have an identical output range of $-1.0 \leq f(x) \leq 1.0$) which will be addressed later. There are some minor disadvantages for intricate bitwise operations which require additional computation, but in general the floating point representation of amplitude values is easier to conceptualize and work with.

One of the dangers involved in using floating-point values to represent sample amplitudes is that the range of values is a very small ratio of the maximum value that a floating-point number can represent in a digital system. Sample values in software can (and will) contain values of much greater magnitude than the maximum allowable amplitude of a sample (1.0 or -1.0). If these large sample values reach a playback system, they can cause extremely loud pops and noise artifacts that can be damaging to equipment, especially speakers. Care must be taken when coding to avoid logical errors that may cause large amplitude increases to occur.

One important signal processing topic to consider is the way that gain values are interpreted in a digital system as opposed to the way that gain is usually understood by humans. In a digital system such as a DAW, gain values are most often represented as floating point values. The gain value reflects the

ratio of the output signal amplitude $y[n]$ and input signal amplitude $x[n]$ such that $g = \frac{y[n]}{x[n]}$ for gain value g (note that gain is a ratio and therefore has no units). We can rearrange this equation to obtain $y[n] = g \cdot x[n]$, from which this relationship is most clear. It is evident that for values of g less than 1, the output amplitude will be lower than the input amplitude, whereas for values of g greater than 1 the output amplitude will be higher than the input amplitude. When g is equal to 1, the system is said to have "unity gain," meaning the output is the same amplitude as the input.

Humans perceive volume on a logarithmic scale, which means that as a sound gets louder, it has to become exponentially more powerful to retain the same perceived volume increase. The exact relationship is usually expressed in terms of the decibel (often abbreviated dB), and is defined as $dB = 10 \cdot log_{10}(\frac{P_m}{P_{ref}})$, where P_m is the measured power (in this case, the output of a system) and P_{ref} is a reference value (in this case, the input of the same system). Note that when the ratio $\frac{P_m}{P_{ref}}$ is 1, the gain of a system is said to be 0dB; in other words, the system has unity gain, since the power at the input is equal to the power at the output ("Decibels").

In order to calculate the gain of a system in decibels, then, one needs to know the ratio of the power between the output and the input. This is defined as the square of the ratio of input and output amplitudes, which we have previously defined as gain g (informit). Therefore, for a given system, we can easily convert between linearly scaled gain and logarithmically scaled gain in decibels with the equation $g_{dB} = 10 \cdot log_{10}(g^2)$, which solved for gain yields the equation $g = \sqrt{10^{g_{dB}/20}}$ ("Decibels"). This becomes useful when implementing a GUI; users can set gain values using the more intuitive decibel scale and the software uses these equations to automatically scale this to the linear gain required for amplitude calculations.

This is not to be confused with digital level metering in DAWs and other digital systems, which utilize a version of the decibel scale with a fixed reference power P_{ref} equal to the maximum amplitude value that can be represented by the digital system without distortion. This scale is most often labelled dBFS (decibels full scale) in order to avoid confusion. A signal at 0dBFS thus represents the maximum possible amplitude for the digital system, and all other possible amplitudes are represented relative to this point as negative values where $-\infty$dBFS implies a gain of 0 (Mellor).

The Plugins

Gain

This is one of the simplest useful plugins that can be designed. The purpose of creating such a simple plugin is twofold. First, it was an exercise in using the JUCE interface for writing the software, and secondly, it solidified my understanding of the mathematics behind gain, which is an extremely common DSP operation that forms a fundamental part of other processes and effects.

The actual processing done by the plugin is as simple as multiplying incoming samples by a scalar gain value. For the sake of utility, a polarity inversion option was included. Implementing this is as simple as multiplying all incoming samples by -1 if polarity inversion is desired. The application of this plugin is actually quite useful, and similar plugins are part of stock packages included with many DAWs. Gain can be inserted anywhere in the digital signal path, allowing adjustment of gain staging between plugins or a gain boost on quiet tracks with minimal processing expenditure. It can also be used to quickly check the phase relationship between two sources using the polarity inversion button.

The GUI for the Gain plugin consists of a fader to control gain, a button to invert polarity, and a button to bypass the effect. To increase user-friendliness, the gain fader displays its value in decibels. This is accomplished using the equations discussed previously for conversion between gain in decibels and gain in linear amplitude. These calculations are done at the GUI stage to convert units into what the software needs as quickly and simply as possible. The GUI converts the value set by the user (in dB) into the value necessary for the internal processing (in linear amplitude), which is then stored.

Panorama

The Panorama plugin is an exercise in taking a basic process, in this case gain, and creating something useful and not trivially obvious out of it. Panorama is designed to be more robust than the typical pan controls found in a DAW; it offers the user a choice between three different panning algorithms and a user-adjustable pan law, as well as a finer degree of panning precision than many DAWs.

Panning, conceptually, is a technique used in multi-channel audio mixing to “place” different sources at separate virtual locations in the sound field. Panning is used most often in a stereo speaker arrangement to place a sound source to the right or left of the perceived center point of the stereo image. This is accomplished by varying the gain of the signal that is sent to each stereo channel. For instance, to pan a sound source to the left, one would decrease the gain of the sound source being sent to the right channel. This causes the source to be perceived as coming more strongly from the left side of the stereo image (“MSP Panning Tutorial 2”).

For a mono sound source, the same signal is sent to multiple channels with different gain values for each channel to create the illusion that it is arriving from a certain direction ("MSP Panning Tutorial 2"). For stereo sound sources, the input channel is sent to the corresponding output channel (left input to left output, etc.) with gain values set according to the pan value in the same way as for mono signals. In other words, stereo panning algorithms do *not* introduce crosstalk between the two channels of audio. A stereo signal panned completely to the left will simply output the left channel information at full volume and will output nothing to the right channel, so any information unique to the right channel will be lost. These types of pan controls are often notated as "balance" controls to clarify their intent.

Mathematically, panning can be accomplished in a handful of different ways. By far the easiest way is to fade each channel linearly as the source is swept across the stereo field. To implement this, the user's pan setting is scaled to fit a [0.0-1.0] scale, where 0.0 represents left panning, 0.5 is center, and 1.0 is far right. The gain can then be calculated for the left channel as $g_l = 1.0 - p$, for pan value p. The right channel's gain is scaled opposite of the left channel, and so we arrive at the gain for the right channel as $g_r = 1.0 - g_l$. This algorithm is called the "Linear Crossfade" in the GUI for Panorama ("Tutorial 22").

Once again, however, we find that human hearing does not correlate directly with digital systems. An equal-amplitude panning algorithm such as this one, where the output amplitudes of the left and right channels added together equals the amplitude of the input signal, fails to consider that humans hear volume in terms of *power*, not amplitude. To see this, consider a mono input signal of amplitude 1.0 panned centrally in a stereo, equal-amplitude panning system. The result is left and right output channels with amplitude 0.5 each. If we convert this to decibels, we find that the ratio of output to input for each channel is $10 * \log_{10}\left((\frac{p_{output}}{p_{input}})^2\right) = 10 * \log_{10}(0.5^2) \approx -6\text{dB}$. Since there are two output channels, our output power is twice that of a single channel, and so we can add 3dB of gain (equivalent to a doubling of power) to the individual channel gain to arrive at a total system gain of -3dB. Since a centrally panned mono signal should experience no volume change, we can conclude that equal-amplitude panning is a sub-optimal algorithm ("Tutorial 22").

The most common panning algorithm (used in the vast majority of DAWs and other audio software) is notated "Equal Power" in Panorama, and, as the name suggests, is designed to maintain constant signal power from the input to the output. A simple implementation of equal power panning involves the use of the trigonometric identity $\sin(x)^2 + \cos(x)^2 = 1$ (Max Tutorial). We can multiply through by the square of the input amplitude $x[n]$ to obtain the definition of equal power panning:

$x[n]^2 * \sin(\theta)^2 + x[n]^2 * \cos(\theta)^2 = x[n]^2$. Angle θ is defined as the pan value p scaled to the range $\left[0, \frac{\pi}{2}\right]$, where 0 represents far-left panning, $\frac{\pi}{4}$ represents central panning, and $\frac{\pi}{2}$ represents far-right panning. The two terms on the left side of this equation each represent the power of one channel of output. We can take the square root of each term to obtain the output amplitude $y[n]$ of each channel. After doing so, we arrive at $y_l[n] = x[n] * |\cos(\theta)|$ and $y_r[n] = x[n] * |\sin(\theta)|$. Since the sine and cosine functions are both always positive in the range of interest, we can ignore the absolute value calculation in practice for software optimization. This operation can also be described as defining channel gains $g_l = \cos(\theta)$ and $g_r = \sin(\theta)$.

Another interesting panning algorithm attempts to create the illusion that the sound source is passing in a straight line from one speaker to the other. With equal power panning, for instance, the sound source describes an arc in space from one speaker to the other with radius equal to the distance from the listener to one speaker ("Tutorial 22"). This means that, assuming an equilateral triangle with the listener at one point and a stereo pair of speakers at the other two, we must increase gain at central panning (relative to an equal power panning algorithm) to make the sound appear closer to the listener (via the inverse distance law). This is implemented by applying equal power panning to an input signal and then applying a gain calculated based on the pan value to "oppose" the arc described by equal power panning and "pull" it into a straight line.

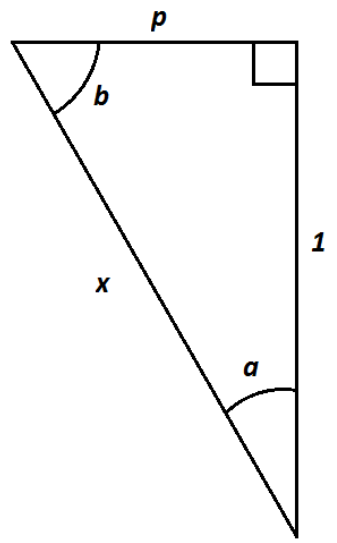

Figure 1: Half of an equilateral triangle forming the geometry of the left side of the ideal listening position.

To describe this arc, we must discuss the geometry of the listening environment. Half of an equilateral triangle representing the position of a virtual sound source in a stereo field is presented in **Figure 1**. The listener is at the intersection of x and the fixed-length leg, while the sound source is at the intersection of p and x when angle *a* is equal to $\frac{\pi}{6}$. The fixed-length leg is the distance between the listener and the center point of a straight line drawn between the two speakers (note that leg p falls, in its entirety, along this line). This leg is set to a reference distance of 1 to simplify calculations.

In order to figure out how much louder our source should be at central panning compared to an equal power panning algorithm, we can use the distance ratio of x and the fixed length leg ($\frac{x}{1} = x$) coupled with the inverse distance law to find the power

ratio of the two algorithms at central panning. To do this we check the maximum size of x, which is equivalent to the distance from the listener to either speaker. At the pan setting (far left panning), angles a and b are both $\frac{\pi}{6}$, and x is equal to $\frac{2}{\sqrt{3}}$. This distance ratio can then be used in the distance squared law to get the ratio of intensity between far left and central pan, which can be simplified $\frac{I_1}{I_2} = \frac{P_1}{4\pi} * \frac{4\pi\left(\frac{2}{\sqrt{3}}\right)^2}{P_2} = (\frac{2}{\sqrt{3}})^2 = \frac{4}{3}$ if we assume that P_1 and P_2 are equal, since we are testing the effects of distance. Plugging in this ratio for $\frac{I_1}{I_2}$, we arrive at the equation $\frac{4}{3} = \frac{P_1}{4\pi {d_1}^2} * \frac{4\pi {d_2}^2}{P_2} = \frac{P_1}{P_2} * \frac{{d_2}^2}{{d_1}^2} = \frac{P_1}{P_2} * (\frac{d_2}{d_1})^2$, from which it is evident that we can accomplish this desired intensity ratio in multiple ways. A power ratio of $\frac{4}{3}$ and a distance ratio of 1 is one solution, while a power ratio of 1 and a distance ratio $\frac{d_2}{d_1} = \frac{2}{\sqrt{3}}$ is another. Therefore, for a distance ratio of $\frac{x}{1} = \frac{2}{\sqrt{3}}$ we can conclude that the equivalent power ratio to create the same intensity ratio at the listening position for two equidistant points is $\frac{4}{3}$. If we convert this ratio to dB, we find that a source must be $10 * \log_{10}\left(\frac{4}{3}\right) = 1.25\text{dB}$ louder in the center of its pan arc to appear to move directly from speaker to speaker.

This can be implemented with the use of some key ratios from **Figure 1**: $a = \tan^{-1}(\frac{p}{\sqrt{3}})$, and x $= x = \cos^{-1}(a)$. In order to avoid possible signal clipping due to gain increase, distance ratio $\frac{1}{x}$ is used in place of x. This creates an arc which decreases in gain toward the outside edges of the pan arc, instead of increasing gain in the center. Then when we pan out to the side and scale our pan range to 0-1 (for pan value p), we can define length x as $cos^{-1}(tan^{-1}(\frac{p}{\sqrt{3}}))$, which is also the distance ratio x. Recall that we are using distance ratio $\frac{1}{x}$, which has equivalent power ratio $(\frac{1}{x})^2$. This can then be plugged into the gain equation (in dB) $g_{dB} = 10 * \log_{10}((\frac{1}{x})^2)$, which expands to yield $g_{dB} = 10 * \log_{10}(\cos(\tan^{-1}(\frac{p}{\sqrt{3}}))^2)$. Since this scaling added to an equal power panning curve results in a signal that is 1.25dB quieter at the edges of its pan arc than equal power panning alone, it is recommended that an additional gain stage be used to add 1.25dB of gain to all signals using speaker-to-speaker panning in this manner.

Unfortunately, the speaker-to-speaker panning illusion is very sensitive to the geometry of any given listening environment. If for any reason the listener is not sitting in an equilateral triangle with his/her speakers (or is wearing headphones), the illusion will be weakened or destroyed. This limits the usefulness of speaker-to-speaker panning as an effect to audio that will only be reproduced on a single

system that is properly oriented for all listeners. This might include audio for film, specialty audio installations, or audio accompaniment for a visual installation.

Most panning algorithms also include a setting called a “pan law,” which is gain reduction applied to compensate for the effects of signals adding together. When two signals at the same volume that have no relation to each other are added together, the output will be 3dB louder than either input signal. If the two signals are identical and perfectly in phase, the output will be 6dB louder than either input signal. For signals that are similar but not identical (such as stereo pairs of microphones), the output is often roughly 4.5dB louder than either input. This becomes important for panning, because the input signals are often mono or stereo pairs. For a mono signal, panning algorithms split the signal into two channels (left and right) for output. When the signal is panned centrally, both channels are playing at once, whereas for a hard-panned signal only one channel is playing. This means that the signal will be louder at central pan than when panned out to one side (“MSP Panning tutorial 2”). The majority of pan law implementations specify a -3dB gain reduction at central pan to compensate for this effect, and gain reduction decreases until unity gain at hard pan to either side. The gain reduction value of -3dB is mostly a matter of tradition, since the pan circuitry on analog consoles most often has a built-in pan law of -3dB. In Panorama, the pan law can be set by the user from 0.0-10.0dB, or disabled entirely. While the pan law is represented in positive decibels in the GUI, it represents gain reduction in decibels. This is implemented using the equation $g_{dB} = (1 - |p|) * L$ to define gain in decibels (where p is the pan value scaled to the range [-1.0, 1.0] and L is the pan law in decibels) and the equation $g = \sqrt{10^{.1 * g_{dB}}}$, seen previously, to convert decibels to linear amplitude.

Panorama is a useful utility that offers features not typically found in plugins. Most DAWs have a fixed equal-power panning algorithm, which is the standard for audio mixing. In most cases this is desirable, but for some special applications in sound design a different panning algorithm can achieve the desired result faster and with less volume automation than an equal-power algorithm. Panorama also offers some additional flexibility during mixing. Unlike a typical DAW’s pan function, which occurs after the entire plugin chain has finished processing the audio, Panorama can be placed at any location in the plugin chain. This allows some interesting options with stereo effects, since the sound can be panned before a stereo delay or reverb, for example, without having to use a stereo send to an effects bus.

Likewise, having the ability to set a custom pan law for the mix (if Panorama is used on every channel) can create the impression of a wider mix if the pan law is increased, since sounds panned out to one side will be louder than those in the center. While this can be achieved via volume control, any

panning automation would have to utilize volume automation as well to remain consistent with the imaging of the rest of the mix. Some audio engineers may find that being able to set a custom pan law is beneficial to their mixing process and helps them create better results faster than their DAW's default pan law, which is usually fixed at 3dB.

BitMangler

For the BitMangler plugin, the objective was to create a useful effects unit capable of applying distortion and filtering to incoming audio to create a low-fidelity sound. In many genres of music, as well as in sound design and film scoring, distortion is an extremely important component of the sounds that form the aesthetic of the music. Distortion as a term is hard to characterize, because much of the processing that can be applied to audio can be considered some type of distortion of the original material. In the context of BitMangler, however, we will define distortion as error that is intentionally introduced into the signal.

One of the simplest ways to introduce distortion into a signal is to clip it off at a certain amplitude threshold, which causes a loss of amplitude information and adds harmonics to the signal. A system that exhibits this behavior can be accurately modelled as a waveshaping process with a parametric transfer function $y[n] = \begin{cases} -t, & x[n] \leq -t \\ x[n], & x[n] < t \\ t, & x[n] \geq t \end{cases}$ where t represents the amplitude threshold.

This is also easily implemented (and perhaps more easily understood) in C++:

```
//Apply hard clipping (digital clipping)
void BitMangler::hardClip(float *sample) {
   float threshold = getClipThreshold();

    if(*sample > threshold) {
        *sample = threshold;
    }

    if(*sample < -threshold) {
        *sample = -threshold;
    }
}
```

This is a highly reduced version of the function actually found in the BitMangler source code, but it serves to illustrate the principle at work. The same type of clipping behavior is found when overloading a digital system such as a DAW, which simply chops off amplitude values whose magnitudes are too great to be represented accurately by the system. This style of distortion (notated as “Hard Clipping” in BitMangler) yields a characteristic sound that is often considered unpleasant or undesirable. For the purposes of a low-fidelity effect, however, it is useful because it is a recognizable distortion character which can be immediately identified as “digital.”

Waveshaping as a distortion tool can be taken much further by using a nonlinear transfer function when processing the audio samples. In particular, the hyperbolic tangent function ($\tanh(x)$) is extremely useful. Its output range is the same as the amplitude range of the audio samples [-1.0, 1.0], which means that it can be easily utilized as a waveshaping function without having to scale or otherwise modify the output. As can be seen in **Figure 2**, the hyperbolic tangent function creates a “soft clipping” curve as the input signal reaches maximum amplitude. This results in the generation of harmonics and soft-kneed compression which create a sound similar to a vacuum tube gain stage or analog tape driven into saturation. Various “magic number” scalars can be inserted into the basic transfer function in order to tailor the sound of the distortion to suit a specific objective, if desired (Burk).

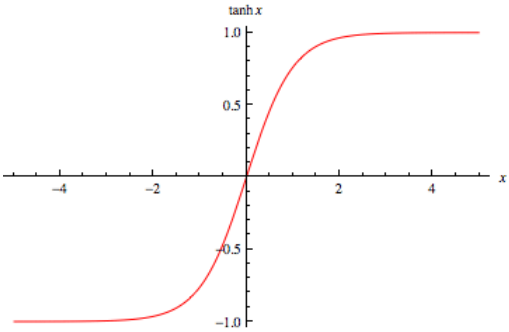

Figure 2: a graph of the transfer function y = tanh(x) (“Hyperbolic Tangent”).

The easiest way to implement distortion with waveshaping is to leave the transfer functions completely unaltered the entire time and determine the amount of distortion to add by adjusting the input gain of the signal, causing it to be clipped more or less by the transfer function (Yeh). Note that it is possible for a signal with magnitude greater than 1.0 to exist inside of a plugin without clipping the system because the floating-point integers that represent sample amplitudes can contain much higher values, but the output of the system must be scaled back to prevent unwanted clipping distortion in the host software. This distortion control is notated as “Drive” in the BitMangler GUI, and is implemented by adding a user-specified amount of gain to the signal before applying the waveshaping transfer function. BitMangler also contains an automatic gain compensation feature at the output of the clipping stage, which applies the inverse of the drive gain to return the signal to approximately the level that was present at the input to this stage. This gain compensation is by no means perfect; since a clipped signal

inherently contains a lower dynamic range than a clean signal, applying the inverse of drive gain to the clipped signal will always result in an output signal that is lower in volume than the input signal. This is a counterintuitive characteristic for a drive control, but calculating additional gain compensation to make up for the amount of gain reduction introduced by clipping is beyond the scope of this project.

Distortion can also be accomplished by introducing quantization error. The process for doing this involves reducing the bit depth of the signal and is aptly termed "bitcrushing." Recall that in a digital system, all amplitude values are quantized to the closest amplitude value that can be represented as an integer value in the desired bit depth. In a 24-bit system, there are 2^{24} possible amplitude values (almost 17 million), whereas a 16-bit system contains only 2^{16} (65,536) possible values. It is clearly evident that the greater the bit depth, the more possible amplitudes exist in the system, which in turn requires less quantization of each sample and causes less overall quantization error. In a nutshell, bitcrushing works by re-quantizing the amplitude values of incoming signals to a new set of possible amplitudes. The quantization noise introduced by bitcrushing is harmonically related to the input signal, but it is far more abrasive to the ear than the $\tanh(x)$ transfer function.

To implement re-quantization in the digital signal processing domain, we can use the amplitude range of our samples, which is [-1.0, 1.0], to our advantage. In order to quantize an incoming amplitude value to a new bit depth, we can round the result of the expression $Y[n] = \frac{x[n]+1}{2} * 2^B$ to the nearest integer $Y_i[n]$ (where B is the target bit depth set by the user). We can then determine the value of the output sample as $y[n] = \frac{2Y_i[n]}{2^B} - 1$ (Brown).

The last distortion algorithm implemented in BitMangler is a process which emulates a faulty analog-to-digital converter stage by creating sample-and-hold errors at a user-defined percentage rate. This is accomplished by keeping track of the last sample to pass through the function. For every sample, we randomly determine whether or not to apply error based on the desired percentage of error set by the user. If we are applying error to an input sample, we output the value of the previous sample instead of that of the input sample. The logical flow of this process can be more easily understood by viewing the source code.

```
//An original algorithm design to simulate sample-and-hold errors at the ADC
//stage.
void BitMangler::applyError(float *leftSample, float *rightSample) {
	float randL = (rand() % 101) / 100.0f;
	float randR = (rand() % 101) / 100.0f;
```

```
        //If we have randomly chosen this sample, apply error to it via sample-
        //and-hold.
        if(randL < getError()) {
               *leftSample = getLastLeftSample();
        } else {
               setLastLeftSample(*leftSample);
        }

        if(randR < getError()) {
               *rightSample = getLastRightSample();
        } else {
               setLastRightSample(*rightSample);
        }
}
```

It is important to note that for samples which are selected for error we are also not setting the last sample. This causes an interesting behavior when error is applied to consecutive samples. Because the last sample is not being reset, all samples to which error is consecutively applied will be output at the same amplitude value. This results in a distortion character that is similar to white noise, and perhaps a bit more garbled-sounding. The sound of the distortion becomes more intense in an intuitive manner as the percentage of error is increased, which helps with ease of use. Because the two channels of stereo audio are both processed independently and randomly, the distortion applied to one audio channel is completely unrelated to the distortion applied to the other channel. This results in a sound that is very "wide," and appears to fill much of the stereo spectrum with noise. It is not appropriate for mono sound sources, generally, because of this. The stereo field location of a mono sound source can easily become obscured or blurred. Stereo audio may sound even wider with a small amount of error applied.

Creating audio filters in the frequency domain is a particularly daunting task in the digital signal processing realm because all of the input information is in the amplitude-time domain. Filtering is a critical part of creating a low-fidelity sound, though, because early audio playback equipment had a very limited frequency response, and this is often the sound that a low-fidelity effect is attempting to emulate. Luckily, filters can be defined simply in the digital domain as a unique set of coefficients. We can then multiply these coefficients with the input signal to produce the desired frequency response in the output signal. Every different filter has a unique set of coefficients $a_1 - a_N,\ b_0 - b_N$ that define its behavior, where N is the number of poles in the filter. Then we create a delay line, which is a FIFO buffer

implemented as an array of samples, with a length of N samples and zero the contents. Input advances one sample at a time through the delay line, and output is defined according to the general difference equation $y[n] = a_1 y[n-1] + a_2 y[n-2] + \cdots + a_N y[n-N] + b_0 x[n] + b_1 x[n-1] + \cdots + b_N x[n-N]$ (Yeh DSP 126).

It is important to note that this equation contains a feedback component, which characterizes this system as an infinite impulse response filter. The feedback component causes the output to remain non-zero indefinitely when an impulse (a single, maximum-amplitude sample) is input into the system, although the output level quickly becomes so low that it can be considered zero for most practical purposes. A finite impulse response filter contains no feedback components and therefore does converge to zero output. In general, infinite impulse response filters can accomplish the same or better frequency-domain performance with less computational expenditure than finite impulse response filters, making them more useful for implementing the filters in BitMangler. The advantages to finite impulse response filters include their inherent stability (they will always converge to zero output and are never subject to runaway feedback). Additionally, they can be designed with a linear phase response; however, this was not a design criterion of the filters for BitMangler (Yeh 135-136).

To aid understanding, the filtering operation can be thought of as a similar process to convolution. A desired frequency response defined as a set of coefficients can be thought of as conceptually similar to an impulse response, which is then multiplied with an input signal to impose the desired frequency response upon the output signal. Unfortunately, we cannot simply define our infinite impulse response filters as a convolution operation because convolution does not typically utilize a feedback component (Yeh 139).

BitMangler's own filters are implemented as simple two pole (12dB/octave) high-pass and low-pass filters, respectively, with a fixed Q of .707 for maximum flat response in the pass band. A 12dB/octave slope was selected to best mimic the sound of low-bandwidth speaker systems, which are often ported and therefore roll off at 12dB/octave below the speaker's tuning frequency. The coefficients are generated in real time as necessary using the equations found in the "Audio EQ Cookbook," a well-known programmer's resource for designing simple audio filters (Bristow-Johnson). Re-defining the filter coefficients for BitMangler in real time does not create clicks or pops, so no additional crossfading or interpolation is required.

BitMangler also includes separate input and output gain controls to use in conjunction with the other distortion and filtering effects. These are useful utilities; distortion inherently wreaks havoc on the gain staging of a signal, and too few plugins provide ample flexibility to compensate for its effects. For

example, the input gain can be increased to drive the distortion harder and the output gain can be decreased to reign in the high signal level. Likewise, the waveshaping distortion algorithms directly affect the dynamic range of the signal, which can cause discrepancies in the perceived volume of the output signal. The output gain can be boosted to accommodate for this.

BitMangler is more of a special effect than a utility plugin, but it can prove useful for creating some truly lo-fi sounds. Many rock and metal artists throughout history have put their guitar, vocals, or something else entirely through ridiculous distortion effects to create a unique, raw sound. The overall sound of BitMangler can range from choked telephone and radio tones to blasting full-range distortion, making it perfect for experimenting with.

Replicator

The Replicator plugin features multiple delay lines in a mixer interface. The primary design objective was to create a delay plugin with as much flexibility as possible while remaining simple to understand and operate. In particular, the GUI being reminiscent of an analog mixing console provides a new user with immediate familiarity, since mixers are close to ubiquitous in the audio engineering world. In fact, most users opening Replicator will already be working inside a DAW interface that functions in a very similar manner.

Replicator incorporates elements of all three of the other plugins discussed thus far, such as controls for gain, pan, and filtering for each delay line, so although the GUI looks quite dense, there are only a few controls whose operations have not yet been discussed. The fundamental digital signal processing operation at work behind Replicator is, of course, delay. Simple delay lines have been discussed as a part of frequency-domain filtering, but their impact on the time domain of the output signal is of even greater consequence and can be used for many different purposes.

For the Replicator plugin, we will be implementing delay lines, like in the BitMangler plugin, as arrays of float values where each value represents one sample. The length of each delay line determines the maximum possible delay time of that line based on the expression $t_S * N$, where N is the length of the delay line in samples. Unlike in BitMangler, however, these delay lines are treated as circular buffers rather than rudimentary FIFO buffers. What this means is that instead of cycling samples through the buffer every time a new sample is introduced, once samples are written into the buffer they do not move. Instead, two pointers are iterated along the length of the buffer, one reading data and the other writing into the buffer. Whenever a pointer reaches the end of the buffer, it is replaced back at the beginning and resumes moving forward. In this sense, the buffer is “circular” because the pointers travel

continuously around it without ever reaching a stopping point. Information in the buffer gets overwritten every time the write pointer passes through. This data structure maximizes memory efficiency by only reassigning two pointers for every output sample calculation, rather than shifting thousands of samples in an array (Reiss 25-26).

In this buffer configuration, the amount of delay time introduced is determined by the distance in samples between the write pointer and the read pointer in the buffer, which is expressed as $t = p_w - p_r$. We can also use this relationship in reverse by solving for p_r; we get $p_r = p_w - t$. This simple relationship can be used to set the location of p_r using a time value input by the user. For ease of use, the Replicator GUI displays the time value in milliseconds. The conversion is accomplished using the relationship $t = \frac{t_{ms} * f_S}{1000}$.

An important component of musically effective delays that has been overlooked thus far is feedback. Many delay effects give the user the option to increase the gain of feedback from the output to the input, where is then delayed again and output later at a lower volume. This manifests as multiple repeats, or "echoes," that fade over time. The more feedback, the more repeats that will be audible, and, accordingly, the repeats take longer to decay (Reiss 22). This is implemented when writing input samples into the buffer: the current output sample (which can be read directly from p_r) is multiplied by a user-defined feedback gain g_{fb} . The result is then added to the input sample and the sum is written in the location of p_w (Reiss 25). It is noteworthy that a feedback setting at or above unity will cause uncontrolled and unbounded positive feedback which could be damaging to equipment. For stability, the relationship $g_{fb} < 1$ should be held (Reiss 23). In Replicator, it is possible to take the feedback up to exactly unity gain, but the consequences for doing so are loud!

The Replicator GUI, like the previously discussed plugins, displays its gain values in decibels full-scale (dBFS) using the same conversion equations. All of the pan controls use an equal-power panning curve.

One interesting consequence of exploring the time-domain consequences of delay lines is that we can still apply this behavior to filters, which use the delay lines for an entirely different purpose. In other words, signals passing through a filter are delayed by the length of the delay line, which is N samples (Yeh). This is a primary reason why many digital effects introduce latency in addition to their intended effects. For the vast majority of filters, this is not an audible delay (2 samples in the case of the filters in this project), but some complex finite impulse response filters can approach thousands of samples of delay, which becomes audible. Even small amounts of difference in delay between two stereo channels can quickly become evident as comb-filtering in the frequency domain, however.

Luckily, most DAWs contain built-in latency compensation which automatically applies a precisely calculated amount of delay to every channel. Channels which are more latent than others due to effects are delayed less in order to re-align these channels with the rest of the audio. The resulting small overall system latency is not noticeable on most digital systems.

Replicator has a number of uses limited only by the imagination. It can do everything from a basic, 100% wet, no feedback delay used to offset an audio stream to a double slapback vocal echo to a reverb-like high-feedback ambience wash. The pan functionality, in particular, creates a new world of possibilities, because the dry signal and all of the delays can be panned in any direction without having to use complex routing and signal summing to achieve the same results with a DAW's pan pots. Bizarre vintage effects like Ozzy Osbourne's characteristic vocal delay on some of Black Sabbath's early songs (notably "Iron Man"), where Ozzy's voice is panned hard one direction and a short slapback delay is panned directly opposite, can be emulated with a single instance of Replicator, making it a perfect experimental tool for trying to find "that sound" (Black Sabbath).

Results:

The complete source code for all four plugins can be found in open-source format under the MIT license at https://www.github.com/BurningCircus. Pre-complied binary releases for Windows 7 64-bit operating systems can also be found in the project repositories. I will be actively maintaining and updating these four plugins at their current GitHub repositories.

The first objective of this project was to increase my understanding of the DSP algorithms that underpin the world of audio software. Through my research, I gained a comprehensive understanding of the processes of gain, panning, delay, distortion, waveshaping, and filtering in the digital domain. These techniques in combination form the basis for a huge portion of audio software currently in use by audio engineers, and have given me an excellent grounding in the fundamentals of digital signal processing.

The second objective of this project was to create a small library of VST plugins that implement these algorithms in useful ways. While it is debatable whether four plugins constitute a "library," each one provides useful capabilities for an audio engineer working in a DAW. Gain provides a simple, CPU-light gain staging control which can be inserted anywhere in a plugin chain, as well as a polarity inversion switch to check the phase relationship between two audio sources. Panorama provides more robust panning control than a typical DAW with three panning algorithms and an adjustable pan law to help shape the overall stereo image of a mix. It can also be placed anywhere in a plugin chain, allowing

signals to be panned before they hit the input of a stereo effects unit without having to use a separate effects bus. BitMangler can be used in a huge variety of situations to shape sound sources, simulate old speakers, add some distortion and fuzz, or just create weird noises. Replicator can be used in almost any situation where an audio engineer might employ a delay effect. It also includes some abilities that are unusual in a delay plugin, such as the ability to pan separate delay lines and the dry signal. I have used all of these plugins in mixing applications soon after developing them, and all of them fulfilled their design goals excellently.

Bibliography

"3rd Party Developer." *Steinberg.net.* Steinberg Media Technologies, GmbH, n.d. Web. 23 June 2015.

"AAX Connectivity Toolkit." *Avid.com.* Avid Technologies, Inc. n.d. Web. 25 June 2015.

Black Sabbath. *Paranoid*. Warner Bros. Records, Inc. 1970. CD.

Brown, Pete. "A simple bitcrusher and sample rate reducer in C++ for a Windows Store App." *10rem.net*. n.p. 13 January 2013. Web. 6 December 2015.

Burk, Phil, Larry Polansky, Douglas Repetto, Mary Roberts, and Dan Rockmore. "Section 4.6: Waveshaping." *Music.columbia.edu*. n.p. n.d. Web. 16 November 2015

"Decibels." *Hyperphysics.phy-astr.gsu.edu*. Georgia State University. n.d. Web. 30 August 2015.

"Features." *Juce.com*. ROLI Ltd. 2016. Web. 10 July 2015.

Heiduska, Cyrus J. "What does 'line level' mean?" *ovnilab.com*. n.p. n.d. Web. 5 March, 2016.

"Hyperbolic Tangent." *Wolfram MathWorld.* Wolfram Research, n.d. Web. 20 November 2015.

"Linear Pulse Code Modulation." *Techopedia.com*. Techopedia, Inc. n.d. Web. 10 March 2016

Lyons, Richard G. "1.2 Signal Amplitude, Magnitude, Power." *Informit.com*. Pearson Education 2010. Web. 10 March 2016.

Mellor, David "What is the difference between 0 dB and 0 dBFS?" *audiomasterclass.com*. Audio Masterclass 2006. Web. 14 March 2016.

"MSP Panning Tutorial 2: Stereo Panning." *Docs.cycling74.com*. Cycling '74 n.d. Web. 24 September 2015.

Park, Tae Hong. *Introduction to Digital Signal Processing: Computer Musically Speaking*. World Scientific Publishing Co. Pte. Ltd. 2010. Print.

Reiss, Joshua D. and Andrew P. McPherson. *Audio Effects: Theory, Implementation, and Application.* Taylor & Francis Group, LLC. 2015. Print.

Shambro, Joe. "What Are VST Plugins?" *homerecording.about.com*. About, Inc. n.d. Web. 30 February 2016.

Smith, Steven W. “Digital Signal Processing Chapter 21: Filter Comparison.” *Dspguide.com.* California Technical Publishing, 2011. Web. 2 March, 2016.

“Tutorial 22: MIDI Panning.” *Docs.cycling74.com*. Cycling ’74 n.d. Web. 24 September 2015.

Yeh, David Te-Mao. *Digital Implementation of Musical Distortion Circuits by Analysis and Simulation*. Diss. Stanford University, 2009. N.p. Print.