Using MobMuPlat to Develop a Mobile Audio Effects Unit

Charles N. Daigle, Benjamin J. Wakefield

December 2020

Abstract

This paper reports on the background, software/hardware design, and future plans of the Mobile Moldable (MoMo) system, where a mobile phone and portable audio interface are used to process real-time audio input, much like a physical "stompbox" style effects unit.

Keywords — Pure Data, MobMuPlat, mobile, audio effects, effects pedal, audio processing

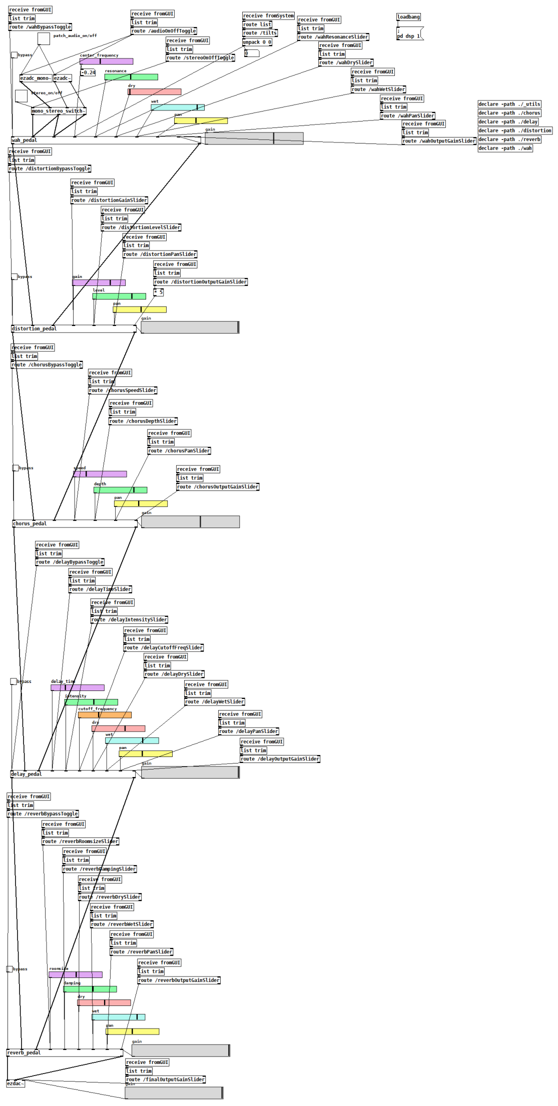


Figure 1: The Pure Data audio effects engine.

Contents

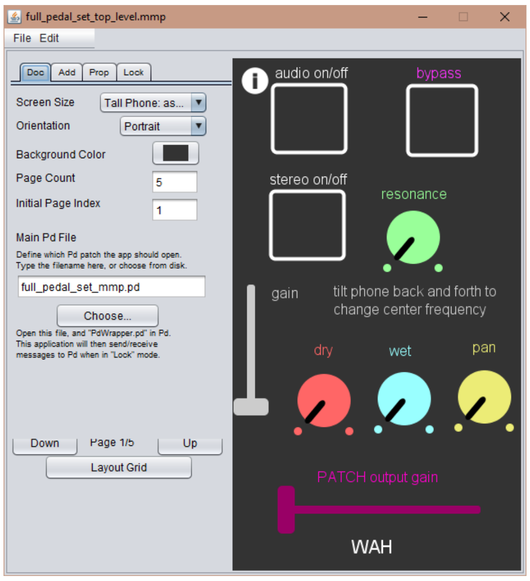
1 Introduction

Mobile devices are recent and powerful additions to the technological zeitgeist. They serve mainly as communications devices, entertainment centers, and portable workstations – they could even be used as a digital pedal to manipulate live audio. When using audio effect pedals, the cost of buying or making a physical stomp-box can be quite high, especially if multiple effects are desired. For example, the Line 6 Helix Multi-Effects Guitar Pedal, the current flagship of the multi-effects market, costs $1,699.99 [6]. Additionally, a physical pedal is difficult to modify. In this report, we present a proof-of-concept implementation of Daniel Iglesia's MobMuPlat (MMP) [13], which is described as "a software suite for running Pure Data [PD] on mobile devices." We create a pedalboard-like GUI that allows a user's mobile device to control a fully customizable Pure Data audio engine. By connecting an audio input via interface to the mobile device, the system can be used to control a guitar in a simple and intuitive way.

1.1 Goal

The goal of this project is to test the ease with which audio effect pedals can be created using Pure Data and MobMuPlat's system of development. By taking note of which features are useful, and which features are either lacking or hinder development, we take the first steps towards improving upon the ideas put forth by Iglesia's software.

1.2 Workflow and Program Structure


full_pedal_set_top_level.mmp

Screen Size: Tall Phone: as...
Orientation: Portrait
Background Color
Page Count: 5
Initial Page Index: 1
Main Pd File
Define which Pd patch the app should open. Type the filename here, or choose from disk.
full_pedal_set_mmp.pd
Open this file, and "PdWrapper.pd" in Pd. This application will then send/receive messages to Pd when in "Lock" mode.

Figure 2: The MobMuPlatEditor. GUI elements of a Pure Data patch must be recreated and linked via OSC-style messaging.

The first step in this process is to create an audio engine in Pure Data. This can be debugged using the GUI elements native to PD (sliders, toggles, radio buttons). Each effect is abstracted to a single block with a bypass inlet, two signal inlets (stereo), and a number of float inputs for sliders/knobs. Using the MMP development environment, MobMuPlatEditor, a corresponding GUI may be created and all GUI elements assigned unique OSC-style addresses (Figure 2). Multiple [receive]-[list trim]-[route] combinations are used to route data from each MMP GUI element to the Pure Data backend (Figure 3) These addresses are connected to the proper inlets of the pedal's abstraction.

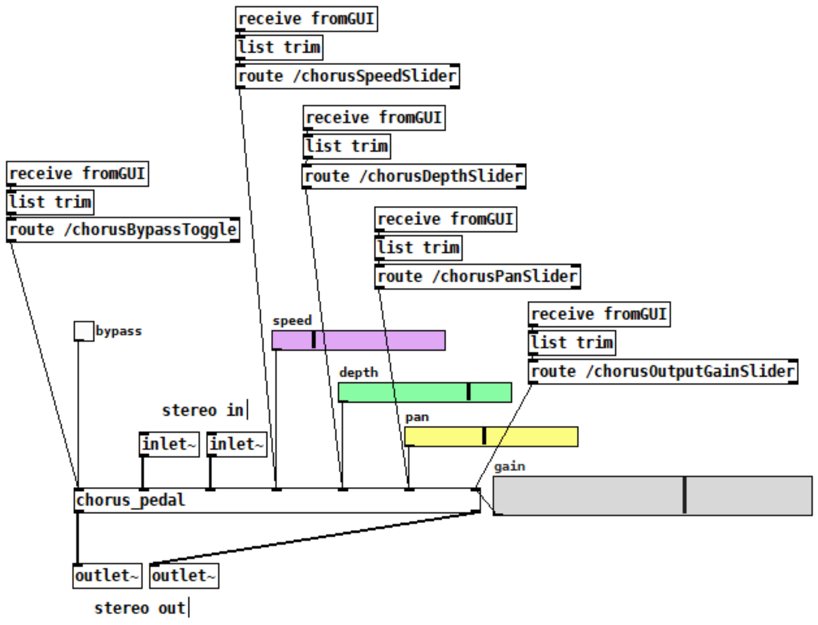


Figure 3: One of our pedals, which is ready to link with an MMP GUI.

MMP GUI elements send data each time they are updated. During development, OSC-addressed data sent from the MobMuPlatEditor GUI development environment is received by the open patch PdWrapper.pd, which then sends the data to the audio engine.

After transferring the .mmp GUI file and the .pd audio engine to the proper directory on a phone (we used a Pixel 3a XL), the GUI can be easily opened and used. Additionally, we used a Comica LinkFlex AD2 [7] to amplify a 1/4" input guitar signal to the phone's 3.5 mm audio jack, which must be configured to receive input through the phone's camera app.

2 Effect Overviews

2.1 Wah-Wah

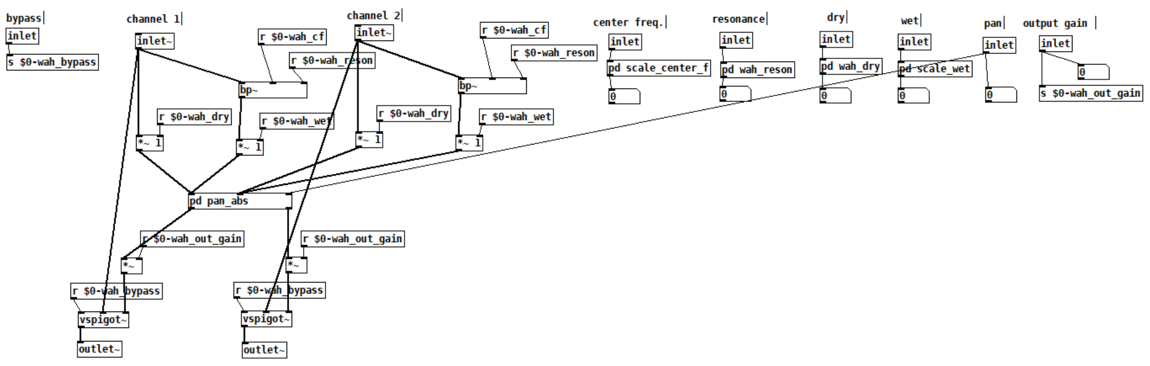


Figure 4: The wah-wah effect's internal structure.

The wah-wah effect (Figure 4) is an audio processing technique that sweeps a resonant low-pass filter envelope over the spectral domain. When this is applied to a sound, especially an electric guitar signal, a quick sweep of the central frequency from around ~300Hz to ~2000Hz shapes the signal in a way that makes it sound like the onomatopoeia "wah." This naming comes from the way that human vocal tracts shape sound in a similar fashion – by nearly closing the lips during the "w" (or rather, the "oo"), the high frequencies in the voiced sound are filtered out. As the lips open, and the "ah" vowel becomes clearer, the high frequencies are audible (Figure 5).

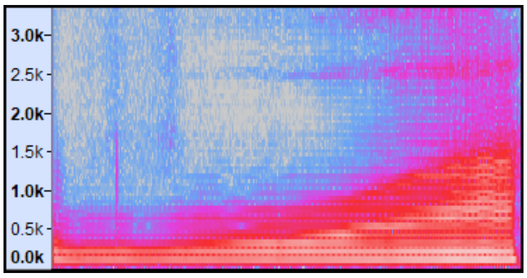


Figure 5: A spectrogram of the spoken word "wah." Note the emergence of upper harmonics as the vowel shifts from "oo" to "ah".

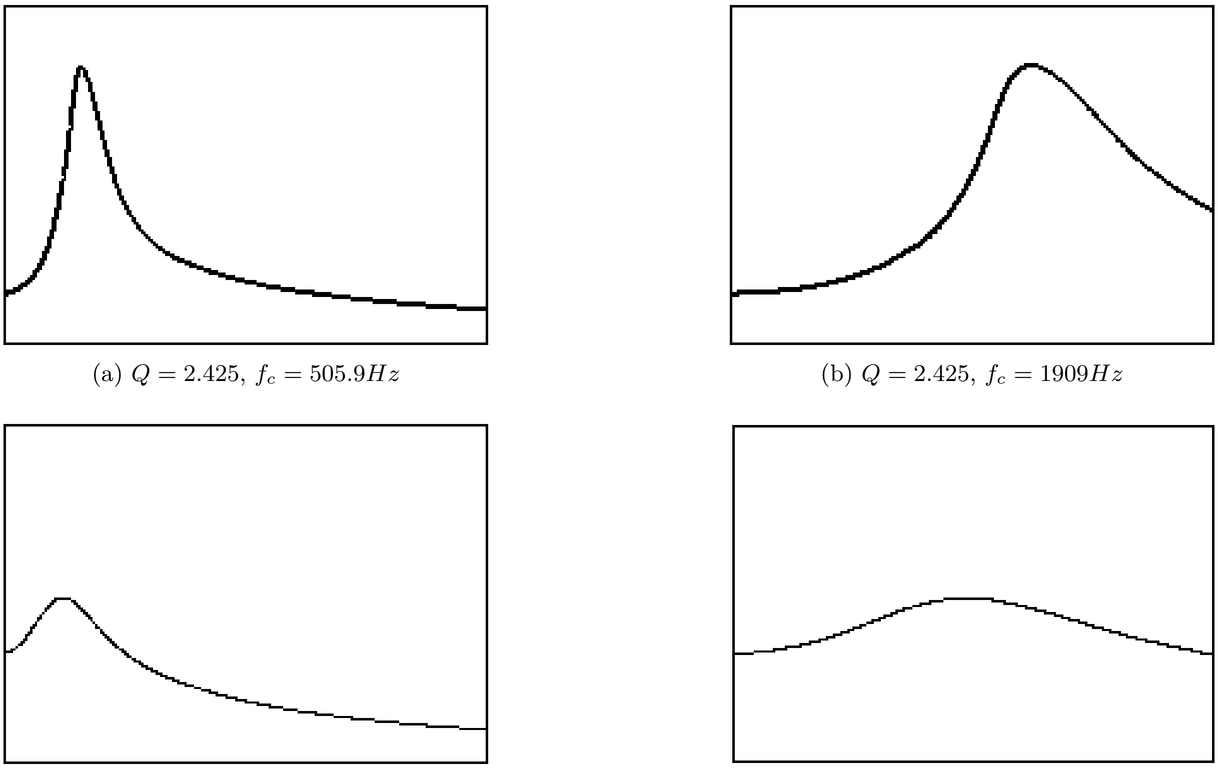


Figure 6: The frequency response of [bob~] for low and high resonances (Q) and center frequencies (f_c). Note the widening of the envelope as f_c changes.

In analog electronics, this effect can be achieved by constructing an RLC circuit using an inductor, a capacitor, and a variable resistor. The capacitor and resistor alone can be used to form a low-pass filter, but by using an inductor, a resonant peak in the spectral domain can be achieved, which is an improvement in clarity of sound over simply sweeping a non-resonant low-pass filter across a range of frequencies. By actuating the variable resistor with a foot pedal, the center frequency sweeps across a set range. One added benefit of analog electronics is that the varying resistance not only changes the center frequency, but changes the shape of the envelope as it sweeps. This adds a layer of complexity to the sound, as it has a narrow character that widens as the center frequency sweeps up (Figure 6). The first electronic wah pedal was made by accident in 1965, when Brad Plunkett decided to replace an expensive "mid-range boost" switch on a Vox AC-100 amp with a cheaper potentiometer. He then found that turning the knob while playing produced the curious "wah-wah" effect, and after placing the potentiometer into a foot-actuated enclosure, the wah pedal was born [8].

In the Pure Data implementation, the bandpass filter [bp~] was initially used as the spectral envelope whose center frequency was swept across the range. This created issues with clarity of the filtered sound – a bandpass filter simply does not allow many frequencies through, especially when it has high resonance. This led to a weak-sounding effect, where modulation of the center frequency was barely audible, and was too narrow-sounding to be understood as a modulation of the original sound. [bob~] is a filter created by Miller Puckette [19] and is included in the "extras" folder in the Pure Data files. It is described as a numerical approximation to an analog voltage-controlled Moog filter, and when used to replace [bp~] (Figure 7), the effect was much clearer, and because [bob] is an attempt to recreate an analog filter, the aforementioned widening/narrowing of the spectral envelope occurs. This creates a rich-sounding effect.

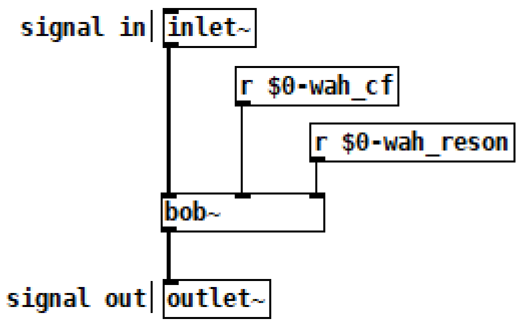


Figure 7: The simple use of [bob~] as a wah-wah effect.

The MobMuPlat implementation of the wah (Figure 8) is slightly more involved than the other effects in this demonstration. Whereas most variables are controlled via links to the GUI, the center frequency of the wah is controlled via the x-axis angular displacement of the mobile device. This allows a user to manipulate their device in the same fashion as a physical wah pedal. The device can be rotated by hand or can be placed in a springy enclosure that can be rotated by foot. This is one of the largest benefits of using a mobile device – they constantly monitor the phone's rotation in all three axes, along with other physically useful data. MobMuPlat sends tilt data in real-time to the backend, which simply goes into the center frequency inlet. The only other unique controllable parameter is the resonance of the filter.

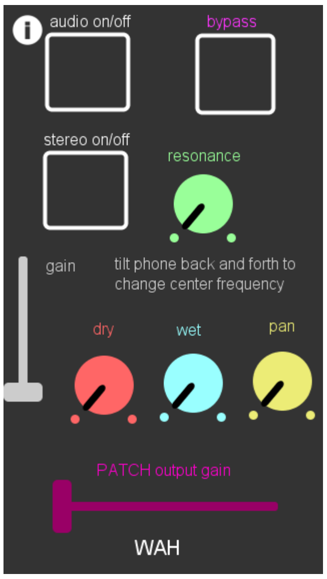


Figure 8: The mobile GUI for the wah effect.

2.2 Distortion

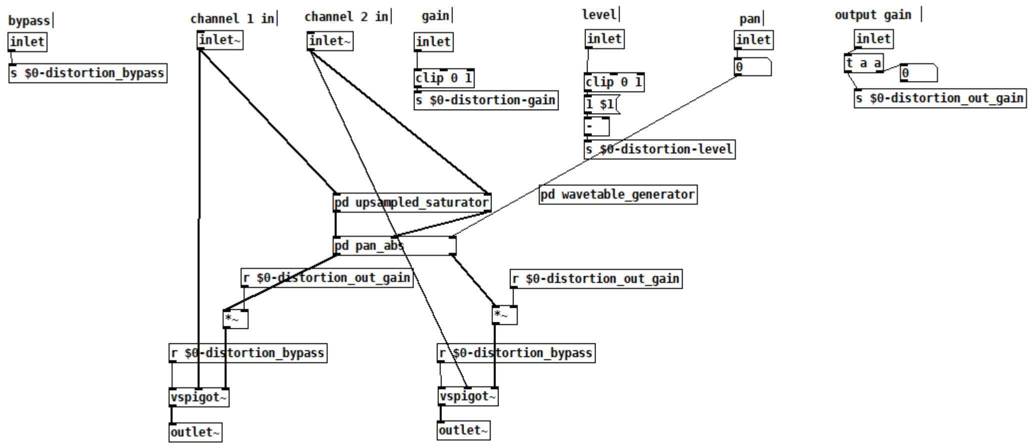


Figure 9: The top-level structure of the distortion effect.

Distortion is a form of audio signal processing used to alter the sound of amplified electric musical instruments, usually by increasing their gain, producing a "fuzzy", "barking", or "gritty" tone. The effect's origin is disputed, but in the early 1960s, Grady Martin discovered a fuzzy guitar tone accidentally caused by a faulty preamplifier while playing on the Marty Robbins song "Don't Worry". Later in the year he recorded "The Fuzz" using the same faulty preamp. Thus, Martin is generally credited as the discoverer of the "fuzz effect." The Maestro FZ-1 Fuzz-Tone was designed by recording engineer Glenn Snoddy and WSM-TV engineer Revis V. Hobbs, who based it on Grady Martin's tone. It was manufactured by Gibson and became the first commercial fuzz effect in 1962 [2]. Distortion is most commonly used in tandem with the electric guitar but may also be used with other electric instruments such as the bass guitar. The effect alters the instrument's sound by clipping the signal, pushing it past its maximum amplitude in order to shear off the peaks and troughs of the waveform. Clipping adds sustain and harmonic/inharmonic overtones, leading to a compressed sound that is often described as "warm" or "dirty", depending on the type and intensity of distortion used. Clipping is a non-linear process that produces frequencies not originally present in the audio signal. These frequencies can be harmonic overtones, meaning they are whole number multiples of one of the signal's original frequencies, or "inharmonic", resulting from general intermodulation distortion [5].

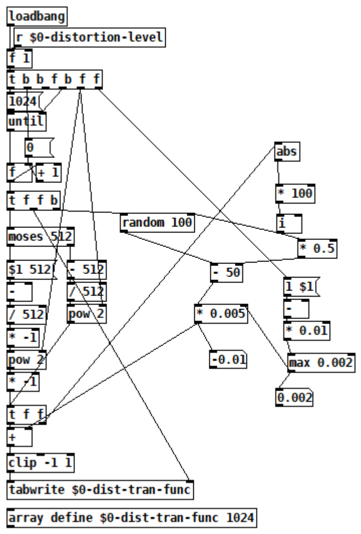
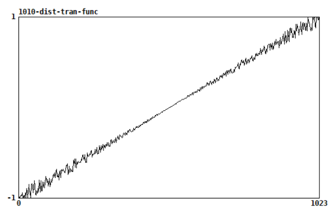
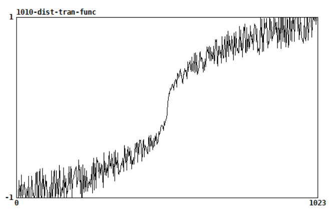


Figure 10: The internal structure of the wavetable generator.

The implementation of the distortion effect within the Pure Data multi-effects patch consists of a sub patch that accepts stereo audio input and parameters that control the bypass function, the pre gain, the level, the panning, and the post gain (Figure 9). The effect was designed to emulate heavy fuzz distortion using a wavetable generator to apply a transfer function to the input signal. This wavetable is generated by a sub patch called [wavetable_generator], which stores the S-shaped transfer function in an array called [dist-tran-func] (Figure 10). The distortion level parameter controls the shape of the transfer function, with lower values yielding a linear


(a) Low level distortion.


(b) Mid level distortion.

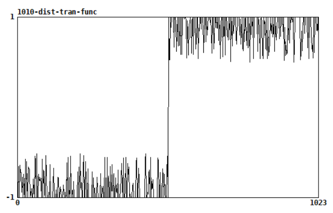

(c) High level distortion.

Figure 11: The stages of the transfer function waveshape.

function and higher values yielding a square function (Figure 11). A [random] object is used within the sub patch in order to add a noisy roll off to each end of the transfer function. This causes the input signal to clip at frequencies chosen at random, which drastically changes the character of the distortion as the level parameter is altered. Additionally, it yields a hard-clipping fuzz effect that boosts high frequency harmonics (Figure 12/Figure 13), which is reminiscent of an octaver/fuzz combination circuit.

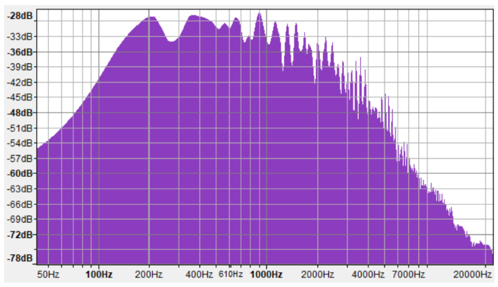


Figure 12: Spectrum of a guitar sample before distortion.

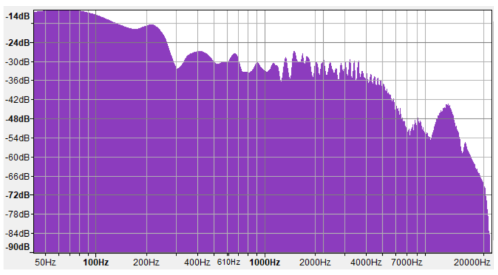
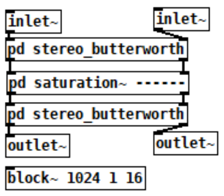
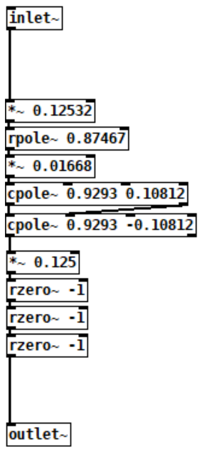


Figure 13: Spectrum of a guitar sample after distortion.

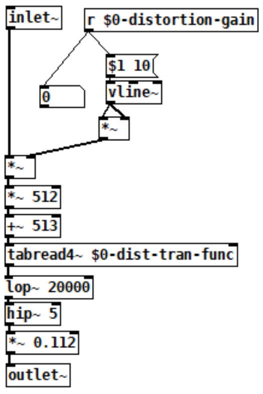
The transfer function is updated constantly within the [wavetable_generator] sub patch, and it is actually applied to the input signal within the [upsampled_saturator] sub patch. In this patch, dual [stereo_butterworth] filters are used to denoise the input and output signal of the [saturation] sub patch, removing harsh high frequencies. The [saturation] sub patch receives the stereo input signal as well as the distortion gain parameter, which controls the amount of gain applied to the stereo signal before it is distorted. The left and right signals are then passed separately through the generated transfer function using two [tabread4] objects. Finally, the distorted signals are then passed through low-pass filters and high-pass filters to eliminate any unnecessary frequencies present in the signals (Figure 14). The result is a practical high-gain distortion effect that can be used with both the electric guitar and the bass guitar quite effectively for both "crunchy" and "fuzzy" sounds.



(a) [upsampled_saturator]



(b) [stereo_butterworth]



(c) [saturation]

Figure 14: The components of the wavetable saturator.

The MobMuPlat GUI implementation of the distortion is trivial. The unique parameters are distortion gain (not like output gain) and level. The output gain is replaced by a volume slider, which acts the same way (Figure 15).

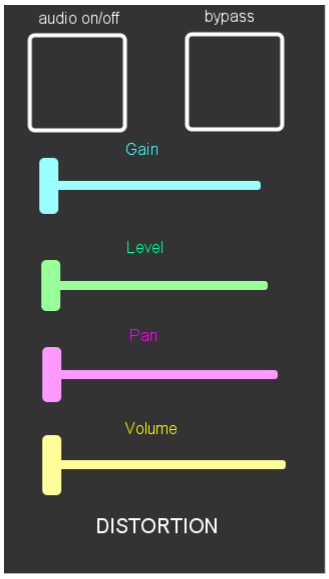


Figure 15: The mobile GUI for the distortion effect.

2.3 Chorus

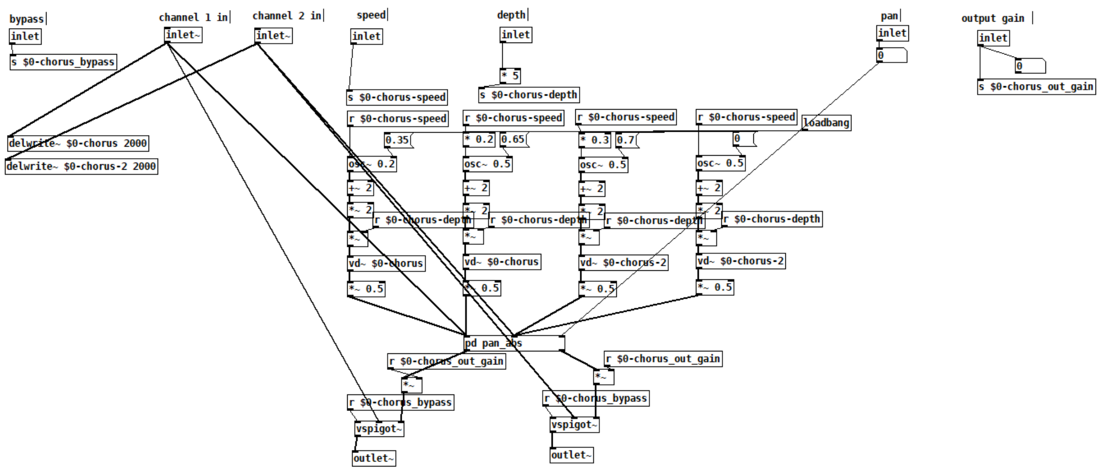


Figure 16: The top-level structure of the chorus effect.

The chorus effect occurs when different sounds with roughly the same arrival time and slightly varied pitches converge and are perceived as a single sound. When the effect is produced successfully, none of the component sounds are perceived as being out of tune. It is typical of sounds with a rich, sparkling quality that would be absent if the sound came from a single source. The sparkle occurs because of beating between the slightly detuned frequencies of the components. This effect can be produced naturally, but it can also be emulated by electronics. One of the first commercially available chorus effect pedals was the Boss CE-1, which was released in 1976. It was originally meant for keyboard and synthesizer players, but guitarists began using it as well [16]. In a digital chorus effect unit, the processor achieves the effect by taking an audio signal and mixing it with multiple delayed, pitch-modulated duplicates of itself. The pitch of the added voices is typically modulated by an LFO, which causes the overall effect to be similar to a flanger, except with longer delays and without feedback. Stereo chorus effect processors produce the same effect, but it is varied between the left and right channels by offsetting the delay or phase of the LFO. The effect is thus enhanced even further because sounds are heard from multiple positions in the stereo field [11].

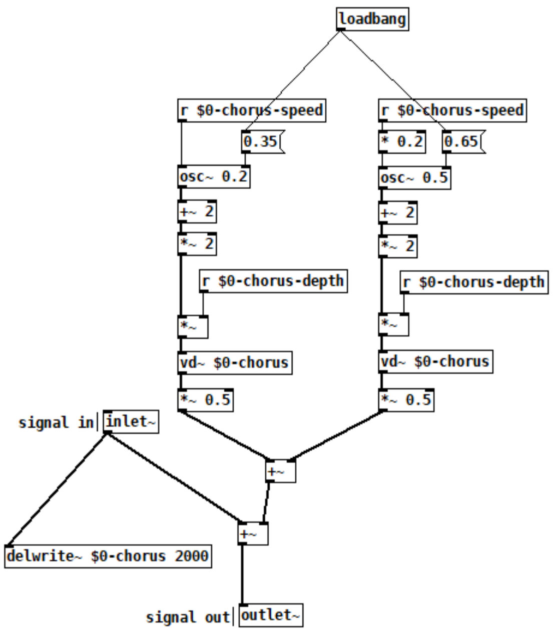


Figure 17: The internal structure of the chorus effect.

The implementation of the chorus effect within the Pure Data multi-effects patch involves a sub patch that receives stereo audio input and parameters that control the bypass function, the speed, the depth, the panning, and the post gain (Figure 16). The effect was designed to emulate a stereo chorus pedal for use with an electric guitar, and the range of its parameters also allow it to be used as a vibrato effect pedal. The effect incorporates four voices, each of which have different [osc] LFOs controlling the speed of the pitch variation. The chorus speed parameter can be used to alter the speed of each of these oscillators, as each oscillator has a different multiplier so each of the four voices remain unique. Additionally, the chorus depth parameter can be used to increase the intensity of the detuning, which is accomplished through a simple multiplier. If this parameter is increased to its maximum value, the effect becomes more like a vibrato if the speed parameter is relatively low. Two of the voices are routed to the left output channel, and two of the voices are routed to the right output channel in order to achieve true stereo chorus. Finally, the modulated signals are combined with the original signal before the panning stage, which completes the chorus effect (Figure 17). The result is a typical stereo chorus effect that is best used with a clean electric guitar or an acoustic guitar, and its wide range of parameters allow for subtle as well as extreme detuning effects.

The MobMuPlat GUI implementation of the chorus is trivial. The unique controllable parameters are the chorus's speed and depth (Figure 18).

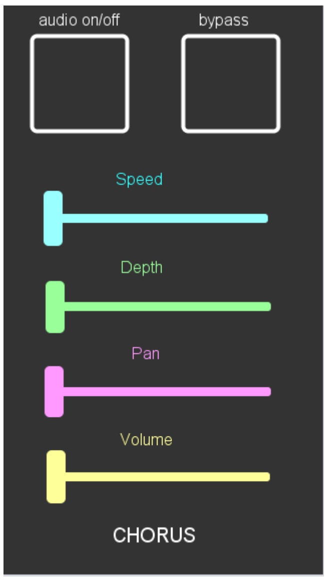


Figure 18: The mobile GUI for the chorus effect.

2.4 Delay

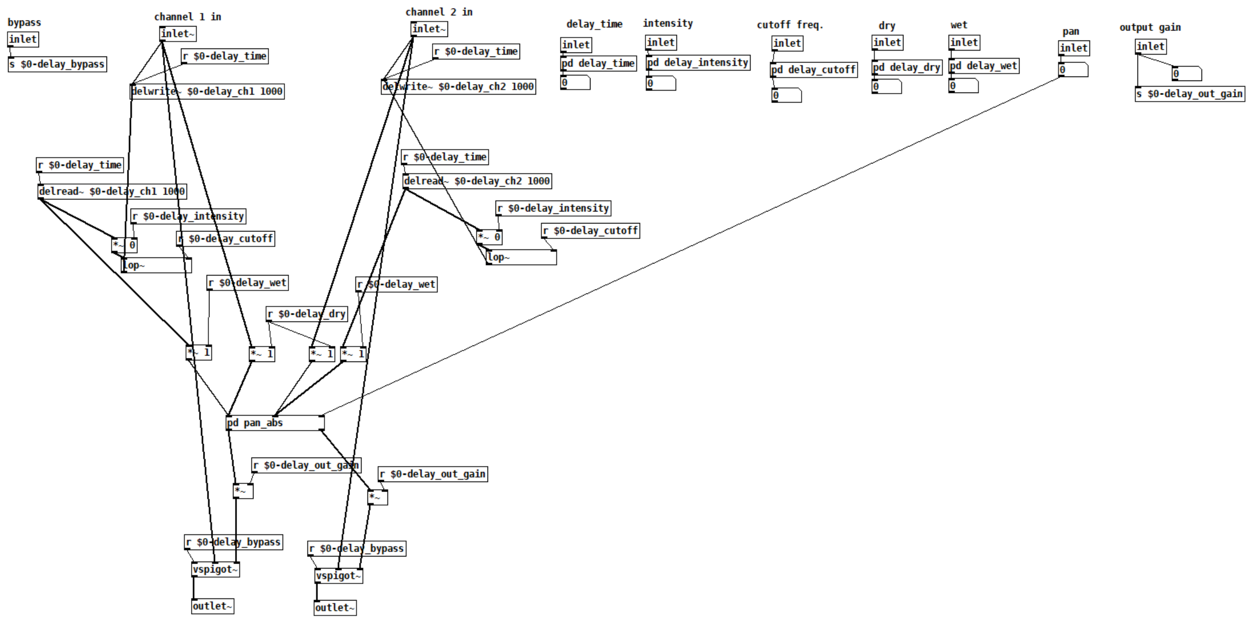


Figure 19: The top-level structure of the delay effect.

Delay is a simple audio processing technique (Figure 19) that reads an input signal into a buffer, then plays it again after a set time. Often, the signal is played back multiple times (rather than just once), and is diminished in volume each time, simulating an echo that decays over time. The first type of delay to store sound in a solid state was magnetic tape delay, where sound could be stored on tape (which acts like the buffer) and played back at any time. The same concept was then applied to solid-state delay, where a series of capacitors would slow down a signal, which could be mixed with the original. This gave way to digital delay, which simply uses an analog-to-digital converter, then stores and reads a signal at a set time interval, using a digital-to-analog converter for output [24].

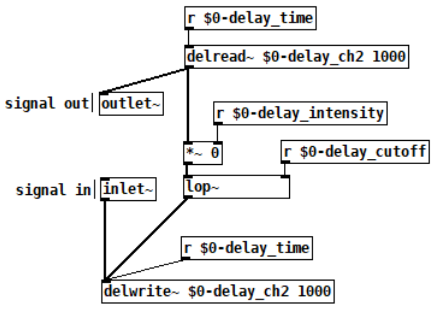


Figure 20: The delay effect's internal structure.

The implementation of this delay is simple (Figure 20) – a [delwrite~] block writes the input signal to a buffer, it is held for the set delay time, and then it is output from a [delread~] block. Additionally, the [delread~] goes through a gain stage and a lowpass filter (with cutoff frequency control) that feeds back into the [delwrite~]. Without the lowpass filter, the delay sounds very harsh and digital, but when each delay repeatedly feeds into the filter, it knocks off more high frequencies each time, until there is silence. This is a very rough way to evolve the delays over time – when each delay is the same, it stands out, especially when the delay time is very short. This also mimics real echo, as high frequencies are more easily scattered and absorbed by the environment than lower frequencies; by disallowing higher frequencies through, the echo is made more realistic.

The MobMuPlat implementation of the delay is trivial (Figure 21). The unique controllable parameters are the delay time, delay intensity, and the cutoff frequency (of the lowpass filter).

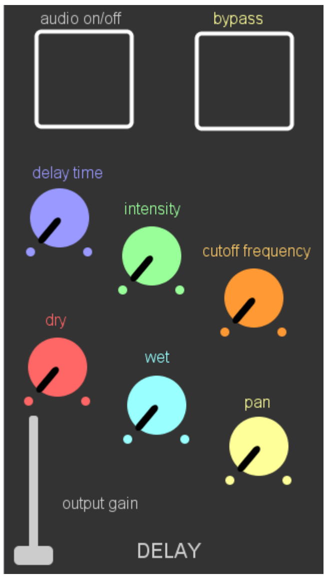


Figure 21: The mobile GUI for the delay effect.

2.5 Reverb

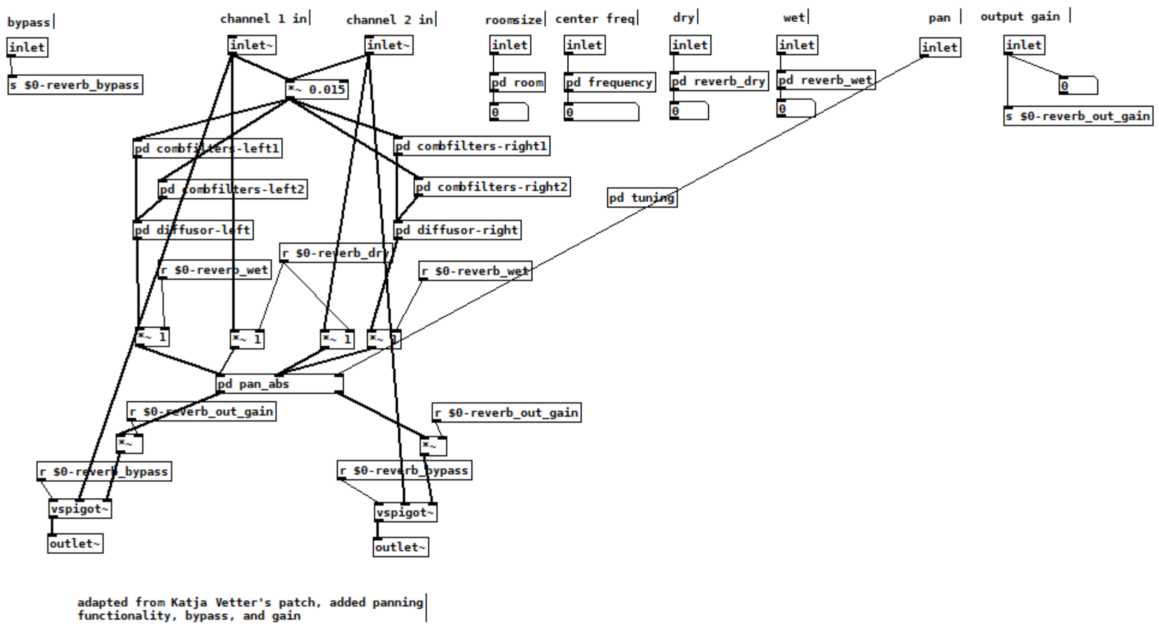
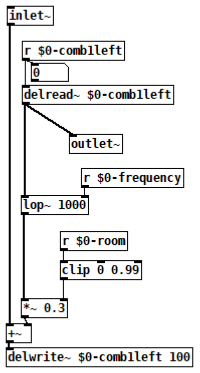


Figure 22: The top-level structure of the reverb effect.

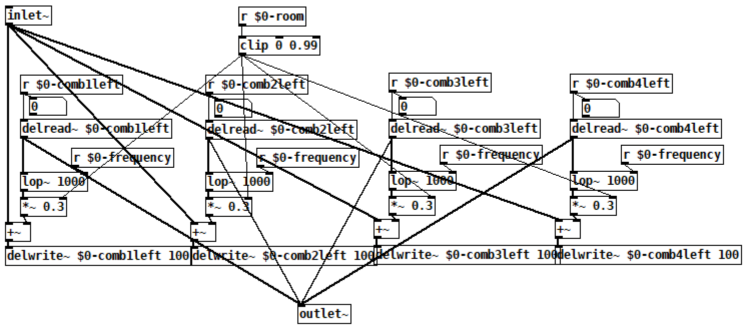
Reverb is an audio effect (Figure 22) that simulates repeated sound reflections off of surfaces. Schroeder reverb [23] is a popular type, where there is a number of parallel comb filters and a number of series allpass filters. A specific implementation of this is Freeverb [22], which uses eight parallel lowpass-feedback comb filters and four allpasses. Normally a public domain C++ program, a version by Katja Vetter can be found on the Pure Data forums, where it is made in vanilla [25]. Natural reverb is ancient – large caves and rooms all feature large spaces with hard surfaces to reflect sound. Mechanical reverb used either springs (1930s) or plates (1957) that create continous delays, and slowly dampen oscillations. Digital reverb was created by Dr. Barry Blesser, who used a series of small digital delays to simulate reflections of sound.

Vetter's patch is modified to match the design of the rest of the effect patches; namely, the features added are bypass, dry, wet, pan, and output gain controls. We use a premade reverb patch analogous to the use of [yafr_abs] in Assignment 6.

In the patch, each of the eight combs uses a delay line ([delwrite~] and [delread~]) that feeds through a lowpass filter [lop~] cutoff at an input frequency. These are all simulating repetitions caused by sound reflections off a wall (Figure 23).



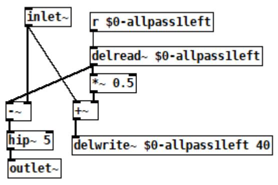
(a) One of the comb filters.



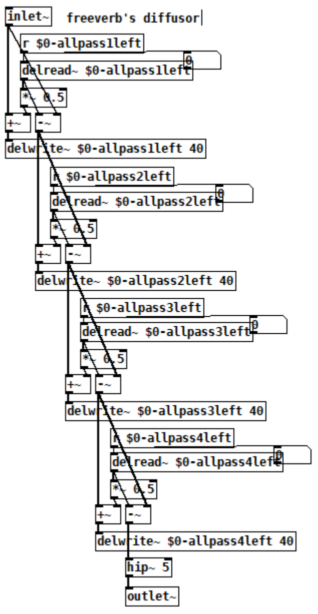
(b) Half the comb filters in parallel.

Figure 23: The comb filter alone and in its parallel stage.

This comb stage then feeds into the diffusor stage, which have similar delay line structure to the comb, but add the input signal to the delayed signal, and subtract the input signal from the delayed read signal. At the end, a highpass filters sub 5 Hz frequencies (Figure 24).



(a) One of the allpass filters.



(b) Allpass filters in series.

Figure 24: The allpass filter alone and in its series (diffusor) stage.

The MobMuPlat implementation of this reverb is somewhat trivial. The unique controllable parameters are the roomsize and the center frequency (of the lowpass filter inside each comb filter) (Figure 25).

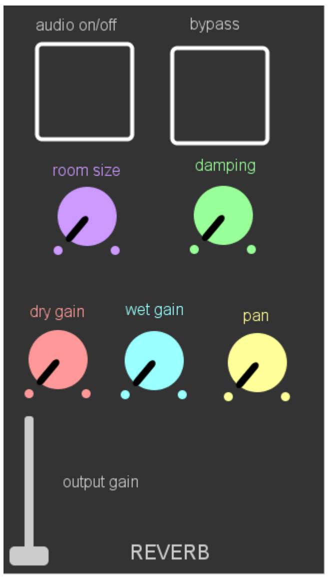


Figure 25: The mobile GUI for the reverb effect.

3 Recurring Design Elements

This section highlights some of the repeated features throughout the patches.

3.1 Scale and Send Parameters as Messages

In each effect's abstraction, parameter inlets go directly into a scaling subpatch, which properly scales them and sends them as a message. MobMuPlat only outputs floats between 0 and 1, so scaling allows for any continous input to be shaped accordingly. Sending the parameter as a message allows them to be used neatly in the patch, as Pure Data does not have a wire-routing feature. This leads to some unavoidably messy patches, which is mitigated by wirelessly transmitting data. This also allows for easier ordering of inlets, as they can exist in the desired order at the top of the patch without complicating it (Figure 26).

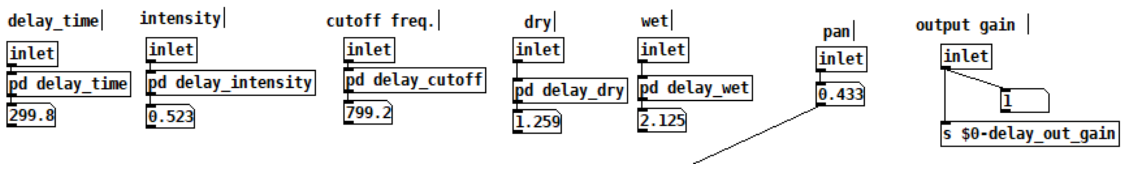


Figure 26: An example of parameter scaling and sending as messages.

3.2 OSC Message Naming Convention

The naming convention used for all OSC messages is like so: /<effect name><parameter name><control type>, for example, /delayDrySlider, or /chorusBypassToggle. This convention minimizes collisions, as it encompasses the smallest amount of information needed to distinguish each control.

3.3 Output Control (Bypass, Dry, Wet, Pan, Output Gain)

Some common controls between pedals are:

- bypass: toggles the effect on/off
- dry/wet gain: floats that control the mix of the unaffected (dry) and affected (wet) signals
- panning: float that controls the stereo balance
- output gain: float that controls the level of the output signal

These are all represented by consistent colors and placement in the user interface.

4 PureData vs. Max

Pure Data is a visual programming language developed by Miller Puckette in the 1990s for creating interactive computer music and multimedia works. While Puckette is the chief developer of the program, Pd is an open-source project with a large developer base working on new features. Pd is very similar in scope and design to Puckette's original Max program and is often interchangeable with Max/MSP, the commercial precursor to the Max language. Pd is natively designed to enable live collaboration across networks or the Internet, allowing musicians connected locally or even globally to create music together in real time with minimal latency [20].

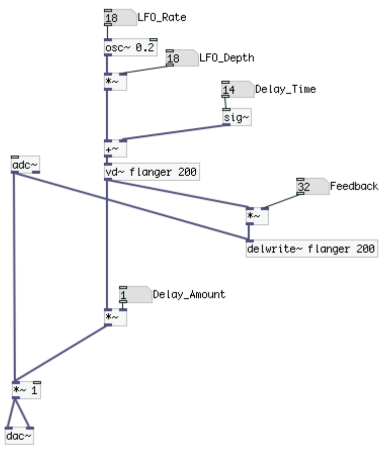


Figure 27: A flanger patch in Pure Data.

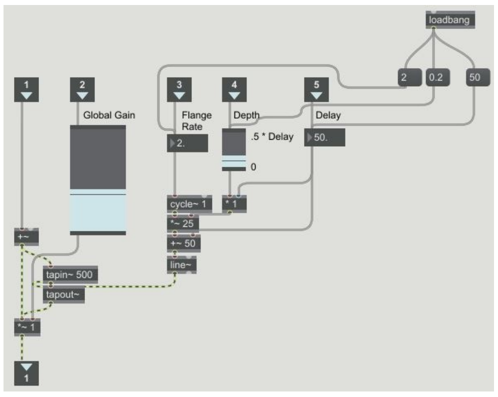


Figure 28: A flanger patch in Max.

4.1 Parallels

Pure Data and Max are both examples of dataflow programming languages. In these languages, function objects are patched together in a graphical environment which models the flow of the control and audio (Figure 27/Figure 28). Unlike the original version of Max, however, Pd was always designed to perform control-rate and audio processing on the host central processing unit rather than offloading the sound synthesis and signal processing to a digital signal processor. Similarly to Max, Pd has a modular code base of objects which are used as building blocks for programs written in the software. This makes the program arbitrarily extensible through a public API, and encourages developers to add their own control and audio routines in the C programming language. However, Pd is also a programming language, and modular units of code written natively in Pd are used as standalone programs and freely shared among the Pd community [14].

4.2 Advantages of Pure Data

Pure Data has many advantages over Max due to it being an open-source project. For example, Pd is free for anyone to use while Max is commercial software that requires a license. Furthermore, Pd is available for Windows, MacOS, and Linux, which makes it able to be used with portable devices such as smartphones and the Raspberry Pi. Max is only available on Windows and MacOS, severely reducing its portability and practicality for DIY projects [21]. Specifically for this project, this factor was a dealbreaker, as the goal was to create a multi-effects unit that runs entirely on a portable device. Although an instance of Max is controllable via a portable device using the Mira app (Figure 29), said instance must already be running on another host device, defeating the purpose for this project [9].

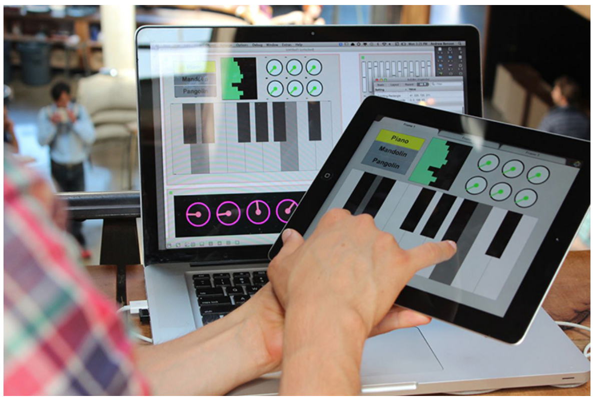


Figure 29: Using MIRA to control an instance of Max on a PC.

4.3 Limitations of Pure Data

Though a powerful language, Pd has certain limitations in its implementation of object-oriented concepts. For example, it is problematic to create massively parallel processes because instantiating and manipulating large lists of objects is impossible due to the lack of a constructor function. Additionally, Pd arrays and other entities are susceptible to namespace collisions because passing the patch instance ID is an extra step and is sometimes difficult to accomplish [12]. Furthermore, Pd is not enhanced to the same extent as Max, leaving such features as anti-aliased oscillators and optimized objects absent, which results in Max having higher audio fidelity than Pd [15]. Along the same note, Pd is not updated regularly, which can lead to bugs and roadblocks in certain implementations. This issue is resolved in Max, which is constantly updated in tandem with Ableton Live [1].

The most important feature of Max that is absent in Pure Data is the ability to load patches in “presentation mode”. Being able to take advantage of this feature for this project would have eliminated the extra step of developing the front-end interface of the multi-effects unit in MobMuPlat. Additionally, certain functions that are native to Max are only accessible in Pd through external libraries, such as Cyclone, which was built specifically to clone objects from Max [18]. However, this can lead to compatibility issues and bugs, as the external libraries are all maintained, or not maintained, by community developers.

5 Future Ideas

5.1 Custom Mobile Application

Although a working prototype that satisfies the main goal of the project was built and tested, there are always additional improvements that can be implemented during continued development. One source of inefficiency in the current design is having to build a patch in Pd, wrap the patch to MobMuPlat, transfer the files to a phone, and finally use the effect. This process is inconvenient for the user and is not practical for everyday usage. One solution to this problem would be to design a custom application for both Android and iOS that supports native Pd patch building. An external library would need to be developed for Pd that mimics the "presentation mode" feature of Max, which would allow users to design their effects entirely on a mobile device, eliminating the need for transferring between a computer and a mobile device. This application could be developed for Android using Android Studio [3] and for iOS using Swift [4].

5.2 Drag and Drop Signal Chain

Another important feature of multi-effects units that could be implemented in the design is the ability to alter the signal chain of the effects. Musicians often experiment with the succession of their pedals in order to obtain interesting sounds, and the current implementation only supports the most commonly used ordering. Giving musicians the ability to drag and drop their pedals in a signal chain is a familiar feature seen in similar applications, such as digital audio workstations, and is especially ergonomic with a touchscreen interface. This feature is possible to implement in Pd using matrices [17], but is not able to be interacted with through the MobMuPlat GUI. Developing a custom mobile application would address this limitation as well.

5.3 Patch Browser Forum

Finally, including a built-in feature that allows users to upload and download other users patches would greatly increase the appeal of using the application. This concept could be implemented by creating a patch browser forum in which users can register, upload their own patches to their profile, rate and comment on other users patch submissions, and even re-purpose them in their own designs. Ideally, this feature would be seamless in the custom application, and users would simply switch to a "browse" tab to download patches, only to have them appear ready to use in an "effect" tab. This concept has already been implemented in the Fender Tone application (Figure 30), but requires linking a mobile device to a Fender amplifier that supports the application [10]. This project would improve upon Fender Tone by allowing users to customise each effect pedal in Pd as opposed to using pre-built effect pedals, as well as removing the need for a specific amplifier to enable functionality, keeping all features contained within the mobile device itself.

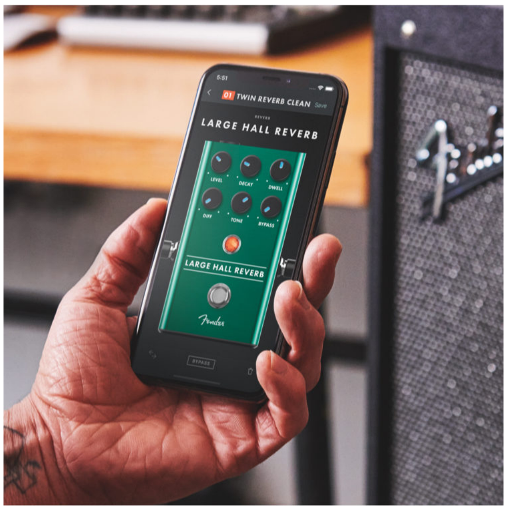


Figure 30: Using Fender Tone to adjust effects on a Fender amplifier.

6 Conclusion

The goal of this project was to implement a guitar multi-effects unit on a mobile device using Pure Data as a back-end platform and MobMuPlat as a front-end interface, addressing the problem of the high cost and limited mobility of physical effects units. Additionally, physical effect pedals cannot be easily customized internally, which prevents additional modification to suit one's needs. In this project, we were able to successfully implement a free, fully customizable multi-effects unit that runs natively on most mobile devices. Although the prototype currently only has basic functionality, the concept has high potential to be expanded and optimized in the future, which would make it a viable open-source alternative to commercial products already being used. In essence, this proof-of-concept was a success, and will serve as a foundation for future projects.

References

[1] Ableton. Max for live. URL `https://www.ableton.com/en/live/max-for-live/`.

[2] Koosha Ahmadi. Initial technology review: Distortion. 2012. URL `https://ses.library.usyd.edu.au/bitstream/handle/2123/8299/tech_Review_distortion_Koosha_Ahmadi.pdf?sequence=2`.

[3] Android. Android studio. URL `https://developer.android.com/studio`.

[4] Apple. Swift - apple developer. URL `https://developer.apple.com/swift/`.

[5] Jon Blackstone. Distortion 101. URL `https://blackstoneappliances.com/dist101.html`.

[6] Guitar Center. Line 6 helix multi-effects guitar pedal. URL `https://www.guitarcenter.com/Line-6/Helix-Multi-Effects-Guitar-Pedal-1430146856949.gc?source=4xvbn01`.

[7] Comica. Linkflex ad2. URL `https://www.amazon.com/LINKFLEX-AD2-Microphone-Phantom-Monitor-Interface/dp/B07RXNC9SR`.

[8] Zachary Crockett. The invention of the wah-wah pedal. URL `https://priceonomics.com/the-invention-of-the-wah-wah-pedal/`.

[9] Cycling'74. Mira: An ipad app that automatically connects to max and mirrors your interface. URL `https://cycling74.com/products/mira`.

[10] Fender Digital. Fender tone companion app for fender amplifiers. URL `https://www.fender.com/tone`.

[11] Christopher Dobrian. A basic chorus effect in max. URL `https://music.arts.uci.edu/dobrian/maxcookbook/basic-chorus-effect`.

[12] Hans. Namespaces implementation in pure data. URL `https://puredata.info/dev/PdNamespaces`.

[13] Daniel Iglesia. The mobility is the message: the development and uses of mobmuplat. URL `http://www.danieliglesia.com/mobmuplat/IglesiaMobMuPlatPaper.pdf`.

[14] Johannes Kreidler. *Loadbang: Programming Electronic Music in Pure Data.* Wolke Verlag, 2009. URL `https://www.wolke-verlag.de/wp-content/uploads/2018/06/Johannes-Kreidler-Loadbang.pdf`.

[15] FLOSS Manuals. Antialiasing in pure data. URL `https://write.flossmanuals.net/pure-data/antialiasing/`.

[16] Florence Munoz. Boss ce-1 chorus ensemble. URL `https://bossarea.com/boss-ce-1-chorus-ensemble/`.

[17] Pierre. Guitar effects chain with pure data. URL `https://guitarextended.wordpress.com/2012/03/05/guitar-effects-chain-with-pure-data/`.

[18] Porres. Cyclone: A set of pure data objects cloned from max/msp. URL `https://github.com/porres/pd-cyclone`.

[19] Miller Puckette. bob . URL `https://github.com/pure-data/pure-data/tree/master/extra/bob~`.

[20] Miller Puckette. Pure data documentation. URL `https://puredata.info/docs/`.

[21] Miller Puckette. What is the difference between pd, max/msp, and jmax? URL `https://puredata.info/docs/faq/pdmaxjmax`.

[22] Julius O. Smith. *Physical Audio Signal Processing.* online book, 2010 edition URL `https://ccrma.stanford.edu/~jos/pasp/`.

[23] Julius O. Smith III. Schroeder reverberators. URL `https://ccrma.stanford.edu/~jos/pasp/Schroeder_Reverberators.html`.

[24] Phil Taylor. History of delay. URL `https://www.effectrode.com/knowledge-base/history-of-delay/`.

[25] Katja Vetter. Freeverb in vanilla pd. URL `https://forum.pdpatchrepo.info/topic/6247/freeverb-in-vanilla-pd`.