Geomixer Team Project Report

Sean O'Connell, Parker Stogdill, Alec Greene, Marco Túlio Giachero Pajaro, Logan Kibler

EECS 452: DSP Design Lab

Table of Contents

I. Background and Motivation

For the EECS 452 Digital Signal Processing Design Lab final project, we were tasked with creating a system that used real-time digital signal processing that used at least one sensor whose data was processed on an embedded platform. We aimed to design a creative solution for sound mixing using geometry. We wanted the product to have a user experience similar to a game that required no musical background to interact with. Traditionally, effects mixing is done using an analog mixing console (Figure 1) or a digital audio workstation (DAW) (Figure 2).

Figure 1. Analog Mixing Console Example. Figure 2. Digital Audio Workstation Example.

We wanted to take a new interfacing approach that uses the same audio processing techniques as DAW effect plug-ins. The prototype we sketched looked as shown in Figure 3.

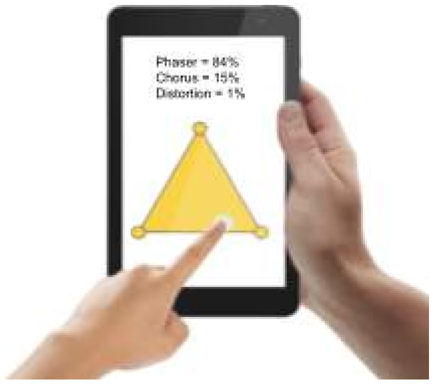

Figure 3. Geomixer Initial Drawing.

Currently, there are interfaces such as Izotope's Visual Mixer that enable the user to intuitively process pre-recorded audio tracks according to their respective position on the screen. However, that application differs from the Geomixer in two aspects: it does not perform real-time digital signal processing and the only adjustable parameters are volume levels and stereo balance. Despite the differences, the user-friendliness of Izotope's Visual Mixer was an inspiration for the design and implementation of our project.

II. Description of Project

The Geomixer is a creative approach on a traditional audio mixer. Geomixer implements a bank of digital audio effects in an intuitive user interface that will apply various levels of those audio effects to input audio from a microphone. The product serves as a user-friendly, tactile approach at audio mixing by using geometric shapes to represent combinations of effects. What we find interesting about this project is learning how to write signal processing software to objectively change audio in a creative manner.

The Geomixer uses a variety of audio digital signal processing techniques in order to achieve the audio effects we desire to create. The implemented effects are chorus, phase shifting, stereo spreader, and distortion. The processes required to achieve these effects include sampling, filtering, convolution, clipping, feedback delay networks, and more. In terms of hardware, the geomixer will require a dynamic microphone, headphones, teensy 4.1 development board, XLR to ¼ inch adapter cable, MacBook Pro computer, and iPad touch screen. The system design is shown in Figure 4.

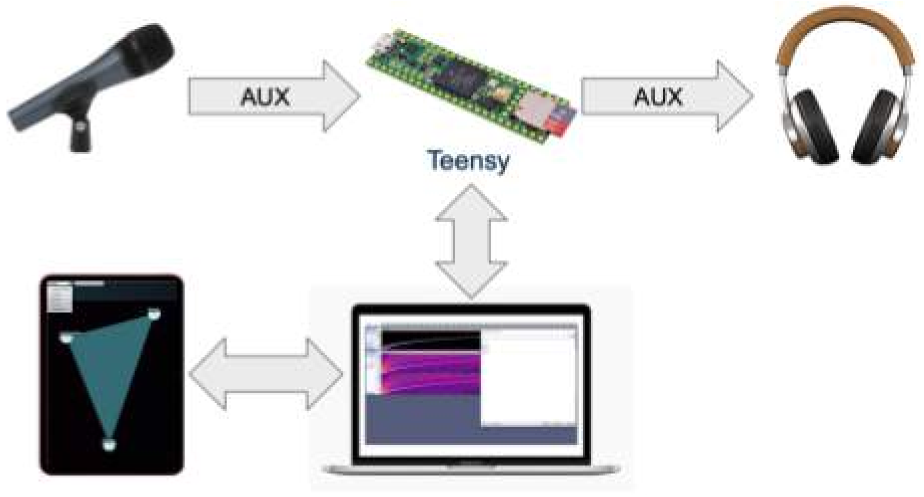

Figure 4. Overall System Design.

The computer uses Duet to run the webpage GUI on the iPad. The GUI allows the user to control the relationship between the effects by placing the effect nodes wherever they would like. The GUI then fills in the area between the nodes to represent valid locations within the shape the user can touch. A screenshot of an example usage of the finalized GUI is included in Figure 5 below.

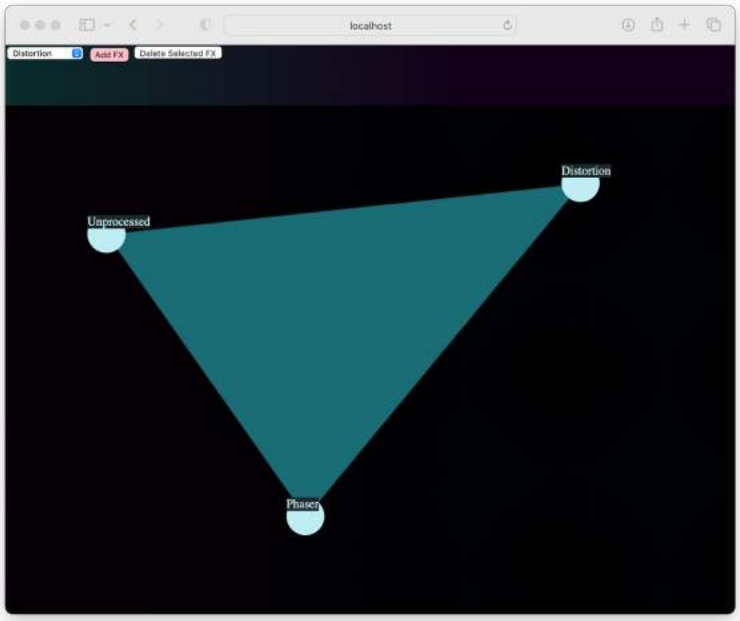

Figure 5. Finalized GUI Design.

The webpage GUI takes input from a person and computes the mix percentage and any adaptive parameter amounts using the location they touched on the screen as well as the location of each effect node. This information is sent over Serial to the Teensy. The microphone takes input from the microphone sensor source and feeds it directly into the Teensy where it is processed and the output is sent to both ears of the headphones. The Teensy Microcontroller processes are shown below in Figure 6.

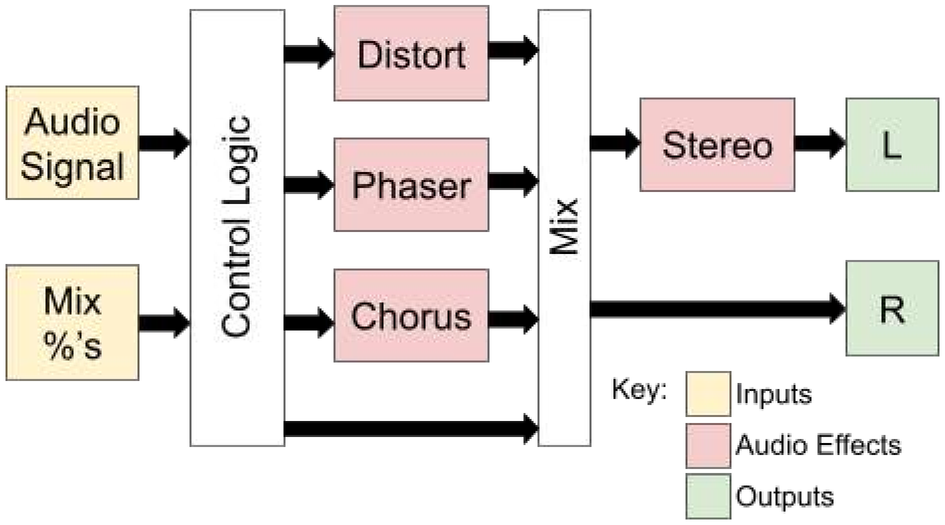

Figure 6. Detailed Block Diagram of Processes in Teensy Microcontroller.

The input to the teensy expects the data to be formatted in a specific way. From that, it parses the mix percentages and adaptive parameter values. The Teensy updates each of the effects with these parameters using the API designed by the Teensy Audio Library and the audio effects we implemented with its included framework.

The main signal processing in this project is the adjustment of audio effects in real time based on live user input data from the GUI. The effects that we implemented are phasor, chorus, stereo spreader, and distortion.

The phasor effect was implemented using an all-pass filter with modulated phase. The chorus effect was implemented using multiple all pass filters and gain adjustments to avoid overflow and fine tune the sound of the effect. Past samples were stored in a circular buffer and depending on the number of voices required, a number of samples would be averaged together in a weighted fashion using gains and outputted through the block transmit function. The distortion effect was implemented using a waveshaping algorithm to change the shape of the input signal by clipping it when the volume goes above a given threshold. The stereo spreader was implemented by delaying the signal in one ear by a value up to 40ms. This takes advantage of the Haas Effect which states that sounds more than 40 ms apart will be perceived as separate sounds. We implemented a circular buffer class to use instances of for the delay based effects.

For some of the effects, there are other adaptive parameters that will be mapped to the GUI interaction other than just mix percentage. For the phasor effect we adjusted the base delay as well as the oscillation speed of the sine wave that adds onto the delay based on the mix percentage. For example, a lower percentage of phaser corresponded to a longer delay with faster oscillations, so when the effect gets more mixed in it will sound closer to what the Chorus sounds like and should make for a more pleasant mix sound. For the chorus effect, we adjusted the number of additional “voices” that would be audible, ranging from 0 to 3, based on the mix percentage passed in by the GUI. For the distortion effect, the threshold to determine how harsh the clipping is an adaptive parameter. For the stereo spreader effect, the only changing parameter is the delay amount being used on one of the ears.

III. Technical Issues

When we set out to create the custom audio effects, we originally planned to implement reverb instead of the stereo effect. We first tried a simulation reverb based on the Shroder’s reverb algorithm but most implementations sounded too similar to the chorus effect with 3 voices. We then tried convolution reverb with an impulse response but it was too slow for real time at a 128 sample block size when the smallest recorded impulse response that we could create was around 9000 samples. We briefly thought about implementing an octave effect but after some research we concluded that it was more complicated to implement than we had time for at that stage of the project. In order to ensure we reached the goal of completing 4 effects, we settled on creating a stereo effect instead. It was important to the success of the project in the timeline we had.

We decided to limit the GUI geometric shapes to triangles because it simplified the parsing needed on the Arduino side. This also allowed for a clearer understanding for the user on how much of each effect is being applied. Also, we ran into an issue when testing with the touch screen instead of a mouse that when the user dragged their finger outside of the shape, it created another node. This was unwanted behavior so we ended up adding in a button to switch between play and create modes for a smoother user experience with the touch screen.

When we originally integrated all of the effects together with the GUI, we ran into timing issues. We were sending serial messages when the user input values were changing at a rate of 5 Hz, but this was too fast to run all of the update functions we needed to in the Teensyduino update function. We changed the communication rate to be 1 Hz instead to eliminate this issue, but had to sacrifice that time difference in user interface to updated audio output delay.

IV. System Testing Strategy & Results

Some of the different tests that we decided were important for validating the correctness of the system were spectrogram recordings that showed the characteristics of each effect on an audio file and comparing those with mixes between the effects. We also completed timing tests for each of the effects periodically in order to understand where the majority of the latency would be introduced and attempt to minimize this as much as possible in the C++ implementations.

For the timing tests, we measured the time in microseconds that it took to run each effect. The fastest effect was the distortion effect which was as expected since it did not require saving and accessing old samples. The slowest effect was chorus which was also as expected because it requires 128 memory accesses into old samples at different delays for each voice and allows for a maximum of 3 voices. The driver logic was very slow and required 1s to run and was the biggest factor in choosing a refresh rate. The measurement results are included in Table 1 below.

Effect	**Time (ms)**
Phasor	0.008-9
Distortion	0.002
Chorus	0.017
Stereo	0.004-5
Driver Logic	1000

Table 1. Timing Results for Audio Effect Implementations.

For the stereo effect, we recorded the output using a test audio sample in Audacity to measure the difference between prevalent peaks in the audio file. This test was used to ensure that the audio delay we expected between the ears was represented in the stereo audio output. The effect allows the delay in the left ear to be up to 40 ms when sent an input of 1 and therefore 20 ms when sent an input of 0.5. Figure 7 shows the results with Serial signals of 0.5 and 1 inputted.

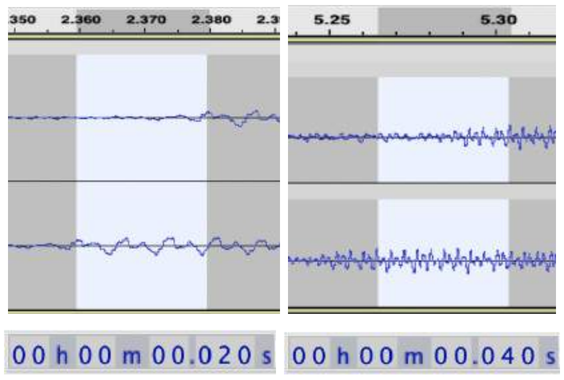

Figure 7. Stereo Spreader 20ms Delay and 40ms Delay Verification Tests.

For the phaser effect, we recorded the output on white noise to visualize the oscillating effect in the frequency domain at 40Hz. We expected to see oscillations representing emphasized frequencies moving up and down the frequency spectrum at a constant rate, which is shown in the results presented in Figure 8.

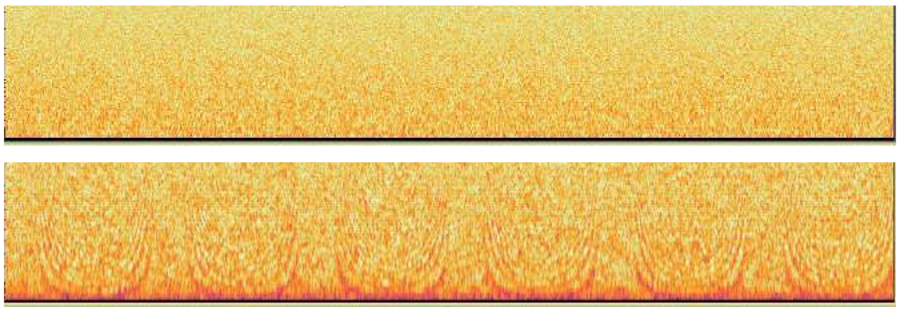

Figure 8. Phaser Input (top) vs. Output (bottom) Tests.

Additionally, we ran tests with Phaser on a pure sine wave. Phaser made for a good starting point for the rest of our effects because it's simpler than Chorus but still has to make use of a lot of the same features like all pass filters and a circular buffer. With the test results show in Figure 9, you can see what the phaser output looks like for a short moment in time on the sine wave, with the shortest delay the portion shown below is at a point of destructive interference so you can see the resulting output at the bottom has a much smaller amplitude, whereas the max delayed version is still at a moment of constructive interference and the amplitude is greater. If you were to look further forward in the time domain, both of the phaser outputs oscillate in amplitude periodically.

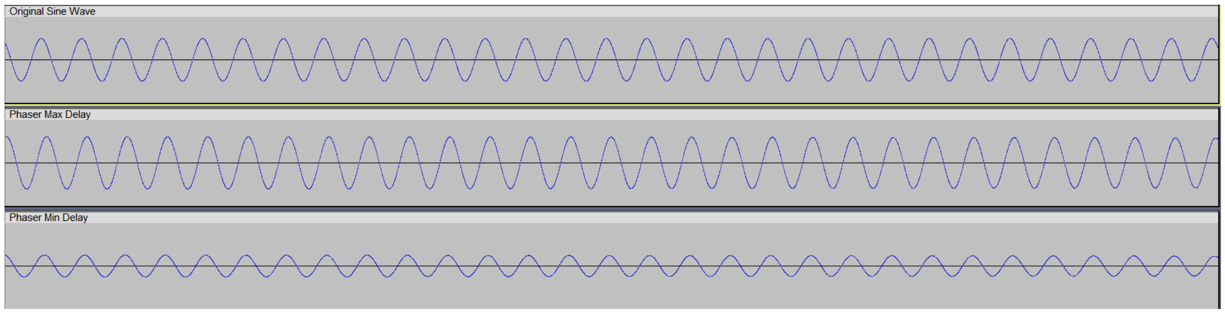

Figure 9. Phaser Output (Bottom Two) on a Sine Wave (top).

For the chorus effect, we recorded the output on a short violin sample. We wanted to be sure that we observed some frequency domain overlap between each note because the chorus effect is creating spectral bleed by adding delayed versions of the signal back into itself. As shown through more uniform coloring in the last tracks in Figure 10, there is more spectral bleed as the number of additional voices is increased.

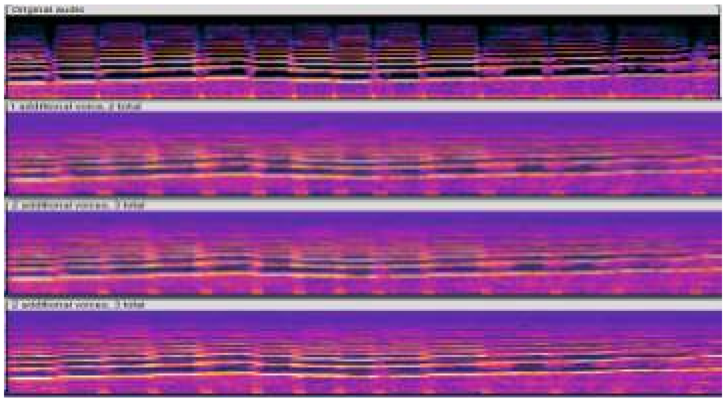

Figure 10. Violin Input vs. Different Chorus Outputs.

For the distortion effect, we recorded the output with maximum volume on a 500 Hz sine wave. We were looking to see that the peaks of the output were being cut off and the sine wave transformed to look closer to a square wave as shown in Figure 11.

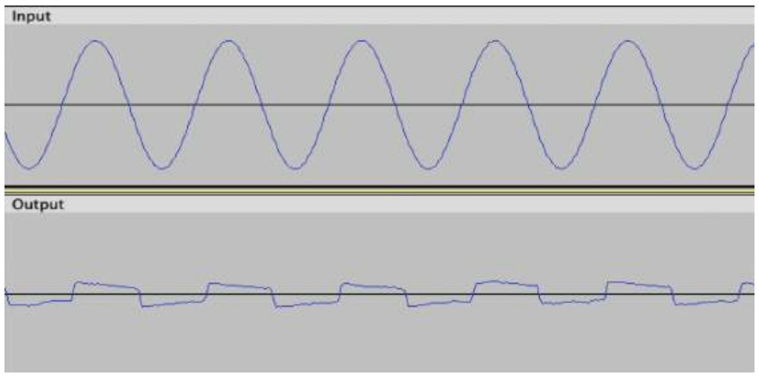

Figure 11. Distortion Input vs. Output on 500 Hz Sine Wave.

Using a spectrogram of white noise input, we could characterize the phasor and distortion effects separately, and then mix them to 50% of each and see that both of these effects are correctly being mixed together. The results of this test are shown below in Figure 12.

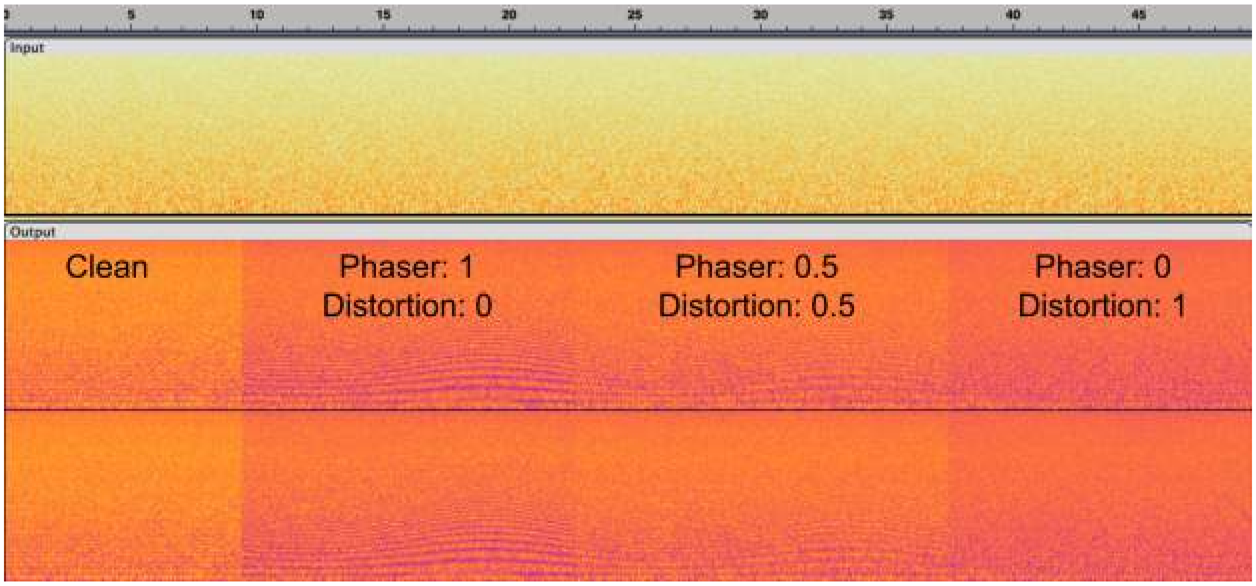

Figure 12. Mixing Distortion and Phasor on White Noise.

V. Future Work

In a future implementation of this project, we would like to change the way that the Teensyduino logic mixed between effects. We would like to have the wires themselves switch connections in order to add true bypasses for the effects. The way that the audio effects are written, we wrote in bypasses that just passed through the audio blocks in the interrupt driven update functions in C++. Instead, we could have the connections from the effects into the mixer change based on the effects being used so that the other effects are not running in the signal chain and would hopefully be ignored in runtime. This would speed up the output and create a more efficient implementation.

We would also like to add more functionality to the Geomixer. Since we have the framework written and working, we could fairly easily plug in many more audio effects such as reverb, bit crusher, and a frequency shifting effect. We would especially like to add more effects that deal with changing the audio in the frequency domain directly. We could also add an additional level of interaction to the GUI to allow us to change adaptive parameters independently from the mix percentage interaction, such as through selecting a node and changing those parameters on an additional slider or knob. A more complex GUI could also allow for creating multiple triangles of effects or a shape larger than a triangle as well with some edits also required on the Teensyduino logic side.

VI. References

"A Schroeder Reverberator," *A Schroeder reverberator called JCRev*. [Online]. Available: https://ccrma.stanford.edu/~jos/Reverb/A_Schroeder_Reverberator_called.html. [Accessed: 07-Dec-2022].

"Introducing a new way to mix: Visual mixer," *iZotope,* 01-Dec-2022. [Online]. Available: https://www.izotope.com/en/learn/introducing-a-new-way-to-mix-visual-mixer.html. [Accessed: 2022].

"Teensy audio library," *PJRC*. [Online]. Available: https://www.pjrc.com/teensy/td_libs_Audio. html. [Accessed: 2022].

U. Zölzer, *DAFX: Digital Audio effects*. Chichester (West Sussex): John Wiley & Sons, 2011.

Figures:

5088 16-channel Analog Mixing Console. https://www.sweetwater.com/store/detail/5088-16A–rupert-neve-designs-5088-shelford-analog-mixing-console-16-channel-with-automation.

Ableton Live 11. https://www.engadget.com/ableton-live-11-daw-hands-on-review-upgrades -explained-160047192.html.

Conrad ElectronicPJRC Microcontroller Teensy 4.1. https://www.conrad.com/p/pjrc-microc ontroller-teensy-41-2269230.

jhnri4, *Speaker*. https://www.1001freedownloads.com/free-clipart/speaker-5-2.

MacBook Pro. https://www.apple.com/mac/compare/.

Sennheiser E 835 Live Performance Microphone. https://en-us.sennheiser.com/live-performance-microphone-vocal-stage-e-835.

Wireless Headphones transparent PNG. http://www.stickpng.com/img/electronics/headphones/ wireless-headphones.