INTRODUCTION

I'm Alex Hulteen, and I'm a computer science student major with a music minor. I came in undeclared, but after running out of general education requirements and taking CS101, I decided to declare for computer science.

In the fall of my freshman year I started making and releasing lofi beats, (a popular genre of background music). In the internet culture that surrounds this genre, there is an obsession with exceptionally marketed plugins that make wild and inaccurate claims, along with absurd sales as large as 95% off. After purchasing a questionable amount of these plugins and not observing the unique features they claimed to have, I started to wonder what was going on under the hood. How are developers, from big audio companies to independent developers, making these tools?

The array of audio effects that make up a Lofi plugin are diverse, useful for other kinds of processing, and variable in difficulty, making it an ideal candidate for my Senior Project. It has allowed me to satisfy my earlier questions regarding how audio plugins are made, along with giving me the tools to make my own software addressing issues in the current market.

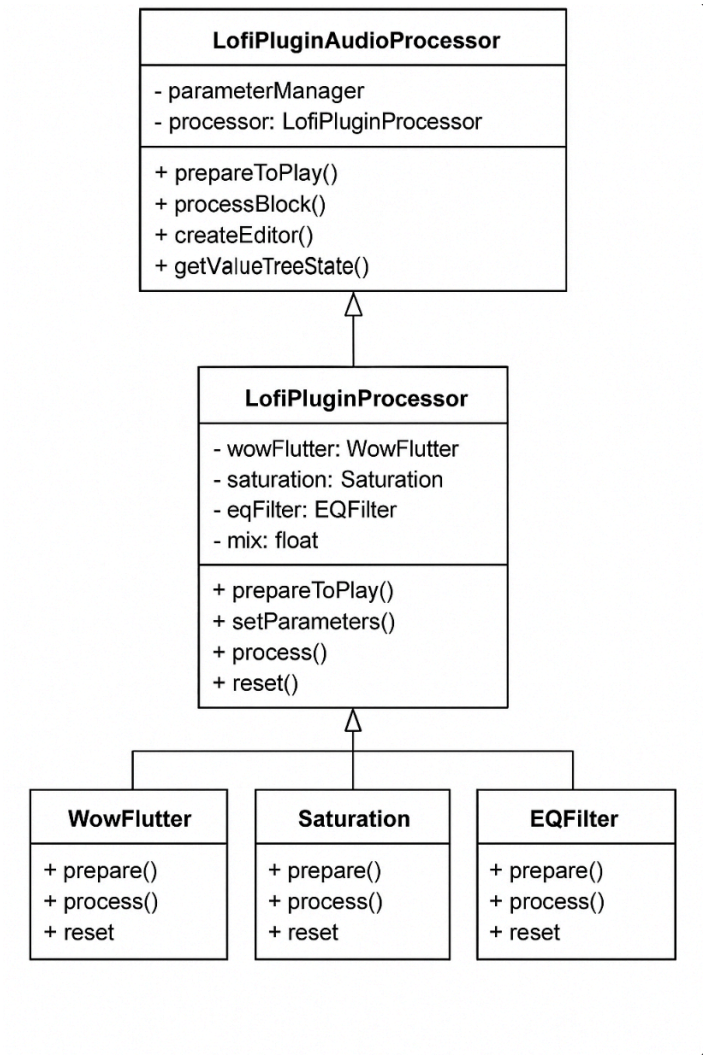
A Digital Audio Workstation (DAW) is a program that is used in every aspect of modern music, from recording, editing, producing, mixing, mastering, and everything in between. Programs like Ableton Live, Logic Pro, FLStudio, and ProTools are standards that have been in development for decades. Most plugins can only run inside a DAW to offload and standardize complications like audio routing and midi sequencing. For this project I have chosen Reaper, created by Justin Frankel (WinAMP, Gnutella) because of its fast load times, customizability, and affordable pricing.

For development, I used the JUCE (Jules' Utility Class Extensions) framework, made for C++ by Julien Storer. The JUCE framework is an open source C++ framework that allows software to be written such that identical source code will compile and run identically on all platforms and formats. This is especially useful for audio production, where different platforms have different formats (VST, AU, AAX). JUCE is a large tool that has many potential uses, but the reason it's great for audio development are the wrapper classes for building audio plugins. All platform and format specific code is contained in the wrapper, so only one version of the code is needed for development. It also includes the Projucer, an IDE tool for creating and managing JUCE projects. I used this tool to generate my Xcode project, along with all of the necessary files and settings inside.

The JUCE framework is built for C++, which allows for a high degree of control. This is important in audio programming where delay and real-time processing are crucial considerations. While C++ has a lot of features, many of the highest level engineers use it as C with classes to reduce the risk of undefined behavior and ambiguous feature combinations. There are a multitude of resources available online for learning JUCE including youtube tutorials, extensive documentation, and a very active forum. My program is built almost entirely from these resources.

In the JUCE framework there are two high-level classes that are automatically generated and required for the project to work: the AudioProcessor, and AudioProcessorEditor. I have made custom versions of these based on the needs of my project.

The LofiPluginAudioProcessorEditor is responsible for the GUI of the plugin. This is my AudioProcessorEditor custom class. It creates my GUI, and attaches my parameters to the GUI elements. It does not process audio, but it exposes controls that modify AudioProcessorValueTreeState parameters. (JUCE's standard parameter container, abbreviated APVTS). These parameters influence the DSP classes, and are managed by the ParameterManager class.



The ParameterManager class is the central state controller for all of the variables that can be changed by the user. My parameter manager creates and manages the APVTS. I created this because it isolates parameters from both the GUI code and the DSP code, making them easier to change and manage. Before I added this class there were issues with undefined behavior in my DSP code that disappeared once I centralized the parameters and removed them from the audio processing, as well as the GUI. Parameters are put into a State struct that give the DSP engine quick access to the parameters without needing to look at UI components or the APVTS. This ensures deterministic behavior and thread safety.

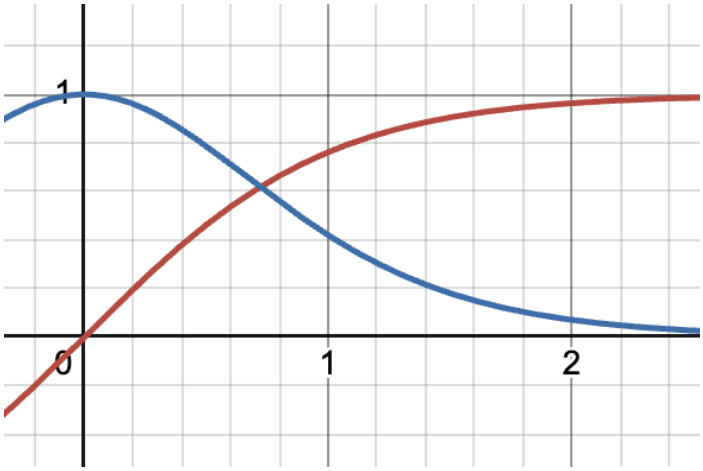
The LofiPluginAudioProcessor is my custom AudioProcessor class, inherited from JUCE. It is the execution entry point for all audio processing, and every instance loaded by Reaper is represented by one instance of this class. This is the central coordinator between the parameter system, the DSP subsystem, the GUI, and the DAW callback lifecycle. When LofiPlugin is loaded in Reaper, input and output channels are created, and it initializes the parameter system. This class calls the required prepareToPlay() method, which forwards information to the DSP layer. It also calls the method processBlock(), the execution loop, called by the host at audio rate. The live parameter values are fetched from the parameter manager, the parameters are applied to the DSP, and the DSP chain is run on the incoming audio buffer. This is also where createEditor() constructs the plugin's GUI.

The LofiPluginProcessor is the DSP engine of the plugin, initializing and managing the three separate effects. The processor takes audio blocks from the LofiPluginAudioProcessor and applies the DSP, making the mix functionality simple to implement. It also makes it easier to test. Being able to call my DSP classes separately was invaluable for debugging. Increasing modularity makes things easier to conceptualize, and classes are cheap.

DIGITAL SIGNAL PROCESSING ALGORITHMS

In analog hardware when signals get high, the components bend the waveform rather than cut it off above the limit. When creating a signal above digital 0, it causes linear clipping which is often glitchy and unpleasant. This saturation implementation

uses a nonlinear function to introduce a more pleasant, fuzzy distortion. Nonlinear processing generates new harmonics, so if the host is running at a sample rate of 44.1kHz, information created above 22.05kHz mirrors back into the audible spectrum of frequencies as aliasing artifacts. Oversampling combats this by increasing the sample rate, performing the DSP at the higher rate, then filtering and downsampling the signal cleanly. The Saturation class code oversamples the incoming audio buffer, applies a basic analog transfer curve (tanh(x)), then downsamples it back down. My oversampling runs the signal at double the host sample rate to reduce the aliasing effect. The drive controls how hard the signal hits the function. The following figure shows the graph of tanh(x) in red, with tanh'(x) in blue. The function mimics the non linear behavior of hardware saturation, with the effect changing in character as the drive parameter is increased.



A great feature to be added in the future is gain compensation. When harmonics are added to a signal, the total amplitude increases. To maintain the integrity of the original signal and increase control, the added gain should be compensated for by reducing the amplitude of the signal manually. The tone control gently filters the high or

low frequencies of the wet signal with a 6db per octave filter, using the same logic as the following EQFilter class.

The EQFilter class includes high and low pass filters, which cut low and high frequencies respectively at the specified frequency. The JUCE DSP library filters I'm using are second order filters with a 12db per octave slope. When the parameters change, it regenerates the IIR coefficients. A bug I ran into was that the UI wouldn't update the filters, which I solved by using an updateFilters() method in order to recalculate and use the coefficients that are used for the processing.

The WowFlutter class uses a circular delay line to store recent samples. Writing to the buffer increments the write index. Reading at an earlier index creates a fixed delay. In order to get pitch changes that can be perceived by humans, you need to implement delay control at a level smaller than individual samples. When making a delay(D) where $D = n + \alpha$ samples, and $\alpha \in (0, 1)$, samples must be interpolated between the samples n and n + 1. I used linear interpolation, expressed by the following formula:

$$y = x[n] + \alpha(x[n + 1] - x[n]).$$

Varying the amount of delay produces a change in phase, which is heard as a change in pitch. If the delay is increased the output is stretched, increasing the distance between samples which creates the pitched down effect. If the delay is decreased the waveform is squeezed, creating the pitched up effect. Wow is defined as slower modulation (approximately 4 kHz or higher), and flutter is modulation at any higher rate. They are the same effect in terms of implementation, which is why I combined them into one control. My LFO (low frequency oscillator) is implemented as sin(lfoPhase) with the

phase increment of the following: $\Delta\varphi = 2\pi * rate / fs$. (The phase difference is equal to two pi multiplied by the rate, divided by the sampling frequency, or sample rate) This creates smooth, band limited sin modulation with a customizable rate. Managing phase wrapping is important to prevent overflow, and keep the sine wave in place. A challenge I ran into was my delay skipping around in the buffer. Originally I used the product of depth * LFO to calculate my delaySamples, and clamped the negative values of the LFO to 1.0 floating point. The fix was to add a base delay offset with variation, using baseDelay + depth * LFO rather than the previous depth * LFO. Linear interpolation is easier to implement than other more complex options (sinc or cubic interpolation), and has a few tradeoffs. First, there can be some frequency dependent errors that act as a subtle low pass filter near the upper limit of the nyquist frequency (half the sample rate). It also does not preserve phase as well as other options, since it causes phase nonlinearity. This can create phase cancellation with very specific settings.

CONCLUSION

This project was more difficult than I was expecting. I started without the class structure, following along to a youtube tutorial series and filling in the gaps. My processBlock method quickly grew out of control, and parameters became ambiguous. I started getting errors revealing that the structure of my code was inherently flawed (such as necessary objects being referenced that had not been created yet), so I designed the system I use here. I didn't know what to expect in terms of implementing the DSP algorithms used to create the effects I wanted, and I found myself spending more time reading academic papers than I was expecting. Since I wanted a custom

implementation, translating these ideas from diagrams and theory to C++ code took a lot of trial, error, and study.

Working with buffers is something I saw in Networking, but using JUCE's circular audio buffers was a new challenge. The JUCE framework has many specific data structures and objects that are used, and it took time to learn the options available, as well as the best practices. The DSP algorithms were the most difficult, and took a lot of trial and error to get to a workable state. My WowFlutter class outputted loud static for weeks before I was able to diagnose the issue (the absence of interpolation or fractional samples). Once I got the initial class structure set up, I couldn't get it to build for a few days because of the limitations of Juce's standalone wrapper. To run as a standalone audio application, it would require separate audio and midi device configurations.

If I were to start this project again, I would first make a simple gain plugin in order to familiarize myself with the structure of the framework and audio programming as a whole, before diving into a complex chain of effects that are difficult to debug. I would fully outline my class architecture before starting to code, even if I wasn't completely sure that it would work. Coding before rigorous planning led to thousands of lines of deleted code, and a lot of junk that didn't need to exist.

In the future I would like to implement a more complex pitch modulation using cubic interpolation, as well as a reverb class. In my research I came across a lot of literature about digital reverberation, and as a followup project I would like to create a custom reverb and delay plugin that combines these ideas using a unique reverb/delay algorithm.

Alexander Hulteen

Afton, MN
(651)-436-3062
Personal email: ahhulteen@gmail.com
Artist email: alexhuelofi@gmail.com

EDUCATION

NORTHERN MICHIGAN UNIVERSITY **Marquette, MI**
Bachelor of Science in Computer Science ***Anticipated graduation: May 2026***
Minors: Music (Piano Performance), Mathematics

TECHNICAL SKILLS

Languages: C++, Java, Python, SQL, HTML/CSS

Frameworks & Tools: JUCE 8, Git, MySQL, Logic Pro X, Ableton Live

Focus Areas: Digital Signal Processing (DSP), real-time audio, plugin development

ACADEMIC PROJECTS / RELEVANT COURSEWORK

SENIOR PROJECT - Multi-Effect Audio Plugin (JUCE 8, C++)

- Designed and implemented a multi-effect VST plugin featuring custom DSP modules: wow/flutter, oversampled saturation, and high/low EQ filtering
- Developed realistic wow/flutter modulation using delay-line LFOs with tunable rate/depth; implemented oversampled tanh saturation for musical harmonic distortion
- Applied DSP optimization and artifact-reduction techniques to ensure stable real-time performance
- Designed a custom GUI with rotary encoders, labels, and a wet/dry mix system, maintaining accurate parameter to DSP synchronization

SOFTWARE ENGINEERING PROJECT

- Collaborated with 3 person team using Git to build software simulating medical biochemistry lab workflows
- Developed a Python interface backed by a SQL database for storing and retrieving medical data
- Implemented a secure account creation and login system using Python and SQL

RELEVANT COURSEWORK

- Software Engineering, Microcomputer Architecture, Operating Systems, Database Management Systems, Network Programming, Server-Side Web Development, Principles of Programming, Discrete Mathematics, Web-Site Development

PROFESSIONAL EXPERIENCE

INDEPENDENT MUSICIAN as ‘Alex Hue’

- Released 25 tracks across 14 independent record labels, totaling 6+ million streams on Spotify and Apple Music
- Experienced in music production, recording, mixing, and mastering of multitrack projects
- Collaborations with artists in three countries using remote production workflows
- Skilled with Logic Pro X, Ableton Live, and industry standard audio tools
- Private piano instructor with individualized curriculum development