CIS 595 Independent Study
Spring 2026

Audio Software Engineering
Audio Plug-in development with DAW integration
Utilizing C++ and JUCE Framework

Michael Evan 02081213
Graduate Computer Science Department
UMassDartmouth

Abstract

This project presents the design, implementation, and local integration of a VST3, Apple AU (audio unit), and stand-alone audio plug-in using the JUCE C++ framework, producing a host-compatible real-time effect module for desktop DAWs. The work applies modern C++ practices for DSP (digital signal processing) and reproducible build tooling to implement a low-latency signal processing algorithm with parameter automation, preset management, and a responsive GUI (graphical user interface). Key engineering challenges addressed include efficient audio-callback handling, latency and CPU minimization via algorithmic optimizations and SIMD-friendly (single instruction multiple data) code paths, and reliable host interaction for sample-accurate parameter changes. Validation is performed by integrating the plug-in into multiple DAWs on a local development machine, measuring processing time per block, end-to-end latency, and memory footprint across sample rates and buffer sizes. A brief usability assessment gathers feedback on GUI ergonomics and workflow. Deliverables include a portable VST3/AU binary, stand-alone application, documented source code, and a reproducible build pipeline, offering practical guidance for future JUCE-based plug-in development.

Introduction

The goal of the project was to design and implement a compact, production-quality digital delay effect that works reliably inside digital audio workstations (DAWs). The codebase demonstrates common patterns in audio plugin development: AudioProcessor lifecycle management, parameter handling via JUCE's AudioProcessorValueTreeState, circular delay buffers with fractional interpolation, and a message-thread UI that responds to audio-thread state via safe synchronization primitives. The UI (user interface) and DSP (digital signal processing) are separated to keep the audio path fast and predictable while still offering a polished, user-friendly experience.

- **Plug-in**

In the realm of Digital Audio Workstations (DAWs), a plug-in refers to a software component that adds specific functionalities to the core "host" application. They

are commonly referred to as a VST, referring to the original designer, Steinberg's Virtual Studio Technology. The functionalities can range from virtual instruments that synthesize sounds to effects processors that manipulate audio signals, such as reverbs, delays, and compressors, to name a few. Plug-ins integrate seamlessly with the DAW via a specific plug-in API, allowing professional recording engineers and musicians with home recording studios to expand their creative toolkit and help refine their audio productions with specialized tools that might not be built and integrated into a specific DAW's architecture. The unique design aspect of the portable integration of plug-ins enables a modular approach to modern audio production. The programming abilities of third-party developers can be utilized which contributes to the users preferential expansion of a DAWs capabilities.

- **Common Plugin Specifications**

VST2 (Steinberg's virtual studio technology) ... deprecated.
VST3 (Steinberg's current industry standard version)
AAX (Avid Audio eXtension) Avid "Pro Tools" DAW proprietary format.
AU (audio unit) Apple "Logic Pro" DAW proprietary format.

- **OS Specific Integration**

Windows:	VST3, AAX, stand-alone
Mac:	VST3, AAX, AU, stand-alone
Linux:	LV2 (linux standard) ... not utilized for this project.
iOS:	AUv3 (Apple mobile version) ... not for this project.

Note: another number of formats are currently in the development stage, but for the project the most common VST3 format, Apple AU, and stand-alone will be the only specifications designed and tested due to academic limitations.

- **Digital Audio**

Digital audio is represented as a sequence of floating-point numbers, with each number encoding the audio sample’s amplitude at a specific moment in time. Digital signal processing (DSP) programming is used to analyze, filter, modify, and synthesize this numerical audio data for effects, noise reduction, and other transformations during the digital audio signal path.

Main Technologies

JUCE audio framework & C++ programming language

- **JUCE**

JUCE originally stood for "Jules' Utility Class Extensions," named after its creator Julian "Jules" Storer. JUCE is a C++ programming framework providing pre-made source code for use within individual projects. The code provided by JUCE enables the cross-platform capabilities for creating the user interface (UI) so the developer need only concentrate on the DSP coding and design of the plug-in user interface a single time. This accelerates the development time for cross-platform operating system integration. JUCE is considered the industry standard by commercial software engineers for audio plug-in development. Writing the plug-in against the JUCE framework API (application programming interface) allows JUCE to handle VST3, AU, AAX, LV2, stand-alone, or any future plug-in format specification that becomes available.

The JUCE framework is not only for plug-in development. It is a software platform that can be utilized to develop fully functional cross-platform desktop audio applications. An example is the Tracktion DAW engine, which was entirely developed utilizing JUCE. Other alternatives to JUCE exist, but only the JUCE framework and C++ will be utilized for plug-in development in this project.

- **C++ programming language**

C++ is the go-to language for audio plug-ins because it gives the low-level speed and predictable memory control real-time audio needs, enabling low audio latency and efficient CPU usage. It maps directly to native plugin SDKs (VST3/AU/AAX...) and has mature frameworks like JUCE to handle cross-platform builds and UI. You can write SIMD optimized code and avoid garbage-collection pauses that would break audio threads. Plus the tooling and developer ecosystem make it easier to build and ship reliable, high-performance commercial grade plug-ins.

- **Projucer**

Projucer is a tool within JUCE utilized for managing JUCE projects for cross-platform distribution. It creates **.jucer** files describing everything about the specific plug-in project.

- C++, Objective-C, Swift, or JavaScript files
- Images
- fonts
- sound files
- other plug-in assets
- compiler settings for how everything is compiled

The **jucer** file enables multiple IDE & cross-platform use of the project just by copy/paste of the jucer file to another OS containing JUCE & your preferred IDE. This enables cross-platform compiling of the plug-in for use on multiple operating systems. Initial development of a project from start to completion only needs to be accomplished on one system, Mac, Windows, or Linux, then re-compiling on a new target OS. This negates the need for initially maintaining the project on multiple local machines & operating systems.

Note: **Important to remember** -- Addition of any new source files during development of the project must be done in Projucer, **NOT in the IDE**. Export is done from Projucer to the respective IDE (Xcode, VSCode ... etc.) to update the project files. This also pertains to changing any compiler, configuration settings, or file locations. Any changes to project files in the IDE other than code development and refactoring will be overwritten by Projucer. It is important to note, you cannot compile code from within Projucer. To test and build the application, all developer source code along with the JUCE framework project files must be compiled within the IDE. Compiling the project files will render all selected versions of the working plug-in in its current state.

Rule-of-thumb: Projucer is in charge of the project, **not the IDE**, which is unique to most software development. Projucer creates and manages the hierarchy of the entire project file evolution for application development from start to finish.

Relevance and Importance

From the perspective of a graduate computer science student who is also an active recording and touring musician, this project has both academic relevance and job related practicality. Academically, it consolidates real-time programming concepts, concurrency control, and efficient memory management within an applied software engineering context. Practically, a tempo-synced delay is a staple audio effect in music production and live performance. The plugin is designed to be musically useful while demonstrating software engineering best practices that are valuable for academic research and professional DSP development.

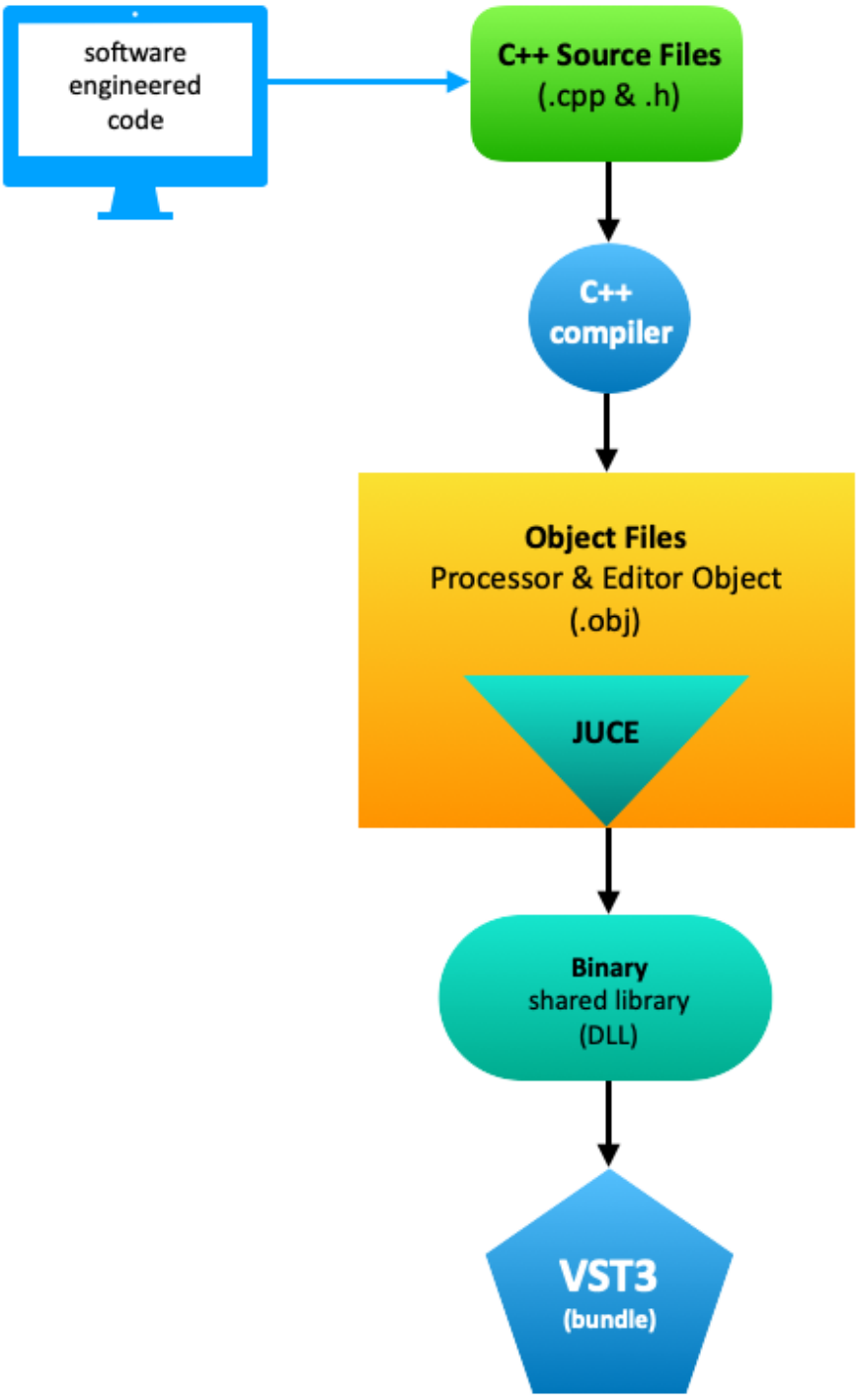
Problem Definition

The project implements a stereo delay with two operational modes: free milli-second (ms) and tempo-synced (musical notes). It needed smooth parameter transitions to avoid zipper noise, an audio phenomenon that occurs when manipulating the delay time, per-channel feedback with stereo control, basic tone shaping via filters in the feedback path, and visual feedback with inclusion of an output gain level meter. The plugin is designed to be host-friendly reading BPM (beats per minute) from the DAW when available (120 BPM default), allow safe UI-to-audio binding, and avoid expensive or unsafe operations on the audio thread.

Major Contributions

The code provides several useful components. ***DelayLine*** implements a circular buffer supporting fractional delays with a four-sample interpolation routine for better audio quality during parameter modulation. Parameters centralizes APVTS setup and performs conversions and smoothed updates for UI and audio use. ***Tempo*** reads host transport info and converts musical note selections into milliseconds using precomputed multipliers. The UI stack includes a reusable ***RotaryKnob*** component to render all rotary controls, a custom ***LookAndFeel*** with an embedded dedicated font and visual styling, and a ***LevelMeter*** that reads atomic peak values from the audio thread to display L/R levels on the message thread. A small ***protectYourEars*** utility runs in debug builds to detect and silence invalid audio outputs during development that may induce dangerous audio levels to the developer or studio speakers. Final build would remove this feature prior to deployment for increased CPU performance.

How Plug-in Is Engineered



DSP and Plug-in Engineering

DSP programming deals with lists of floating point numbers. Their range are values between -1.0 and +1.0 that are processed by a computers DAC (digital-to-audio) converter. The DAC converts these numbers into electrical signals which rapidly move an electromagnetic coil back and forth in a load speaker to produce sound.

For the purpose of DSP development, initially, when dealing with analog audio it is converted to digital format via the computers ADC (analog-to-digital) converter. This is done every few microseconds but the numbers are initially stored as 16-bit or 24-bit integer. This is called the ***bit depth*** of the sample. A bit depth of 16 bits can record sound pressure levels as 65536 (2^{16}) possible values. 24-bit or (2^{24}) becomes 16.8 million possible values. So the host application maps the integers into floating-point numbers between -1 and +1 for ease of use.

A theorem in DSP says the "***sampling rate***", which is the speed at which the ADC takes voltage readings must be at least 2 times the highest frequency in the signal that is being sampled to faithfully represent the original audio signal. Human hearing range tops out at approximately 20,000 Hz or 20 kHz (kilo hertz). This number unfortunately decreases as we age. Doubling this frequency gives us a desired sampling rate of *40000 samples per second* or 40 kHz. During the development of video/audio and the Compact Disc (CD), and the math involved with DSP, researchers settled on a rate of 44100 samples per second or ***44.1 kHz*** *as the initial industry standard*. 48 kHz and 96 kHz are also used in modern DSP but those choices don't necessarily mean better audio which also complicates development. Thus, the project will utilize the 44.1 kHz industry standard sampling rate.

Audio processing code measures time by the duration of each sample or the *sampling interval T.* This is equal to 1/sampleRate. So 1/44.1 = ~ 0.02267 so the *sampling interval* ***T = 23 microseconds*****.**

When the host application (DAW) needs audio from the plug-in it tells the plug-in to execute ***processBlock*** which is called audio call-back. This has the potential to happen up to 1000's of times/second. To reduce the amount of call-backs to be less than the sampling rate the audio stream is chopped up into ***blocks***. A typical block contains 128 samples but can range from 16 to 4096. If the block size is 128

samples with a sample rate of 44.1 kHz the plug-ins's processBlock performs 44100/128 = 344.53 (345) times/second on average. The smaller the block size the more often the host needs to send the processBlock command which can lead to audio stutter and increasing the block size to a certain point can increase the length of time between the request and result which can introduce delay or what's known as ***latency***. Since latency is unavoidable with DSP, ideally we hope that it is so small that it is insignificant with respect to timing issues during recording and we don't perceive the latency effect.

Most DAW's let the user select the block size, which is normally labeled the Buffer size in the user settings. Block and buffer mean the same thing. So the larger the Buffer the more latency. A sample rate of 44.1 kHz & buffer of 256 samples = 5.8 millisecond latency. It must be noted, the block size is not engraved in stone and is constantly being manipulated by the DAW depending on the audio stream.

Anything happening in the processBlock runs in its own thread called the ***audio thread***. Under the hood the CPU prioritizes this thread over all others. It takes precedence ideally keeping audio free of any glitches during recording and play-back.

In the code all initial objects provided by JUCE start with juce:: this is called the names-pace. JUCE also uses British English to spell its names. Anything coded std:: refers to the C++ standard library.

The Source directory for the project contains all the code files necessary for creating the plug-in. Below is a brief summary of the functionality of all the source code files for the project with a link to my GitHub repository for the project.

Project Code Link

https://github.com/MichaelJEvan/UMass-Dartmouth-CIS595_Audio-Engineering

DelayLine.h / DelayLine.cpp

- Implements the circular delay buffer.
- setMaximumDelayInSamples / reset allocate and prepare the buffer.
- write() stores the latest sample; read(delayInSamples) does fractional (4-point cubic-style) interpolation to return a smooth delayed sample.

DSP.h

- Small header of DSP utility functions (e.g., panningEqualPower) used by other modules for common math/helpers.

LevelMeter.h / LevelMeter.cpp

- GUI component that draws the left/right level meters.
- Uses a timer to poll Measurement atomics, converts linear level -> dB -> on-screen position, and draws tick marks and colored level bars.

LookAndFeel.h / LookAndFeel.cpp

- Custom LookAndFeel implementations and colour/font definitions.
- Draws the rotary knob appearance, button styles, custom label/textbox behavior, and provides Fonts via embedded BinaryData.

Measurement.h

- A tiny thread-safe wrapper around std::atomic to track peak/level values between audio thread and GUI safely (compare_exchange, exchange).

Parameters.h / Parameters.cpp

- Defines and creates all plugin parameters via AudioProcessorValueTreeState (APVTS).
- Handles parameter smoothing (LinearSmoothedValue), conversions (ms, dB, Hz strings), and exposes runtime values (gain, delayTime, mix, feedback, tempoSync, delayNote).

PluginEditor.h / PluginEditor.cpp

- The GUI editor: places groups, knobs, tempo sync button, level meter, and wires attachments.
- Listens for parameter changes and toggles visibility (e.g., show delay time vs. choice when tempo sync is active). Uses BinaryData images for background/ logo.

PluginProcessor.h / PluginProcessor.cpp

- The audio core (AudioProcessor). Central coordinator: creates APVTS, instantiates Parameters, DelayLine(s), Tempo, filters, and Measurement objects.
- prepareToPlay sets up buffers and filters; processBlock runs the per-sample/ per-block audio loop: reads inputs, writes to delay lines, reads fractional delays, applies cross-feedback, filtering, mixing, smoothing, and updates level measurements. Also handles state save/load.

ProtectYourEars.h (and where used)

- Debug helper that scans buffers for NaN/Inf/oversized samples and silences them during development (used under JUCE_DEBUG).

RotaryKnob.h / RotaryKnob.cpp

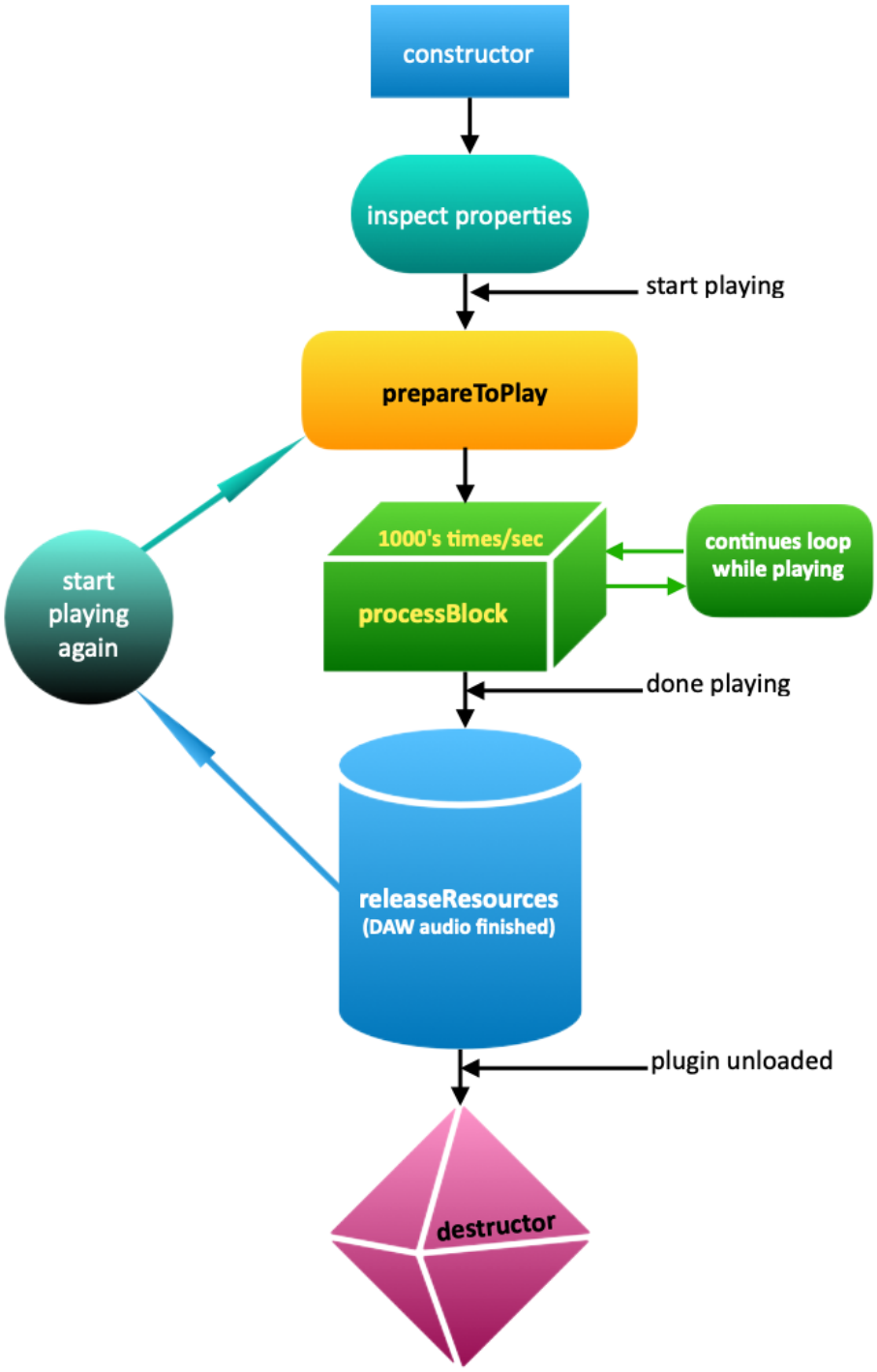
- Small component wrapper around a juce::Slider and Label hooked to an APVTS attachment.
- Sets rotary parameters, text box style, look-and-feel, and an optional “drawFromMiddle” property used by the custom knob drawing code.

Tempo.h / Tempo.cpp

- Reads host tempo via juce::AudioPlayHead (AudioPlayHead::getPosition()) and stores bpm. Defaults to 120 BPM when no host tempo provided.
- Provides getMillisecondsForNoteLength(index) that maps a note-division choice to delay milliseconds using the noteLengthMultipliers table (used to implement tempo-sync).

Plug-in Lifecycle

Diagram below shows the basic coding logic behind the design and implementation of the plug-in. The plug-in's job is to process audio. When the plug-in is active, the host (DAW) sends the ***processBlock*** commands every few milliseconds telling it to perform its designed function.



Methodology

Architecturally, the project separates concerns: the audio processor maintains only the real-time-critical code path; parameters and smoothing live in a helper class; delay buffering and interpolation are isolated in DelayLine; host BPM (beats per minute) and musical conversions live in Tempo; and the editor handles UI layout and attachments. Memory allocations happen in prepareToPlay and resource initialization, ensuring the processBlock path performs only bounded work. Parameters use LinearSmoothedValue and a one-pole smoothing coefficient to prevent abrupt changes in audio. The level meter uses atomic exchange operations to safely share peak data between threads without locks. For tempo synchronization, the plugin queries the host AudioPlayHead and falls back to 120 BPM when no information is available.

Development & Implementation

- **Blank plugin template and DAW/host testing**

I began with a minimal “blank” JUCE plugin created with Projucer and exported as an Xcode project to verify the basic build and host-loading pipeline before adding DSP or UI features. That blank plugin was built and tested in the JUCE AudioPluginHost and in several DAWs (Studio One Pro, Reaper, and Logic Pro) to confirm plugin discovery and proper loading, parameter automation and state recall. To exercise audio I/O (input/output) and GUI/parameter links early, I routed a Steinway piano plugin through the blank build inside the host so I could listen for correct audio passthrough with no signal degradation via the plugins audio I/O, and parameter behavior before integrating the actual delay DSP and UI (see Appendix Figure 1).

- **Project structure, build targets and asset handling (Projucer + Xcode)**

The project was organized into clear modules: PluginProcessor (audio core), Parameters (APVTS), Tempo and DelayLine (timing and buffering), LookAndFeel and Editor components (UI), and small utilities (Measurement, ProtectYourEars). All assets (images, fonts) were embedded using Projucer’s BinaryData mechanism so the same bundle worked for AU, VST3 and the Standalone target versions.

Development, iterative builds and debugging were done in Xcode (the exported Projucer project), using Xcode's run, breakpoint and logging features to diagnose integration issues on macOS. There was no use of CMake in this workflow which is sometimes utilized during DSP development so everything used the Projucer-generated Xcode project and standard JUCE exporting features to keep the project manageable within the given academic time frame.

- **Audio processing and integration (test synth)**

The audio core implements a stereo delay with fractional delay reads, cross-feedback, state-variable filtering and tempo sync. To validate real audio behavior and GUI control paths I used the ***JUCE Plug-in Host,*** DAWs, and the stand-alone plug-in in conjunction with a MiniMoog plug-in and external KAWAI midi controller keyboard (as a musical source) through the delay. Initial DSP engineering included the Output Gain, Delay Time, Mix control, and an On/Off Bypass switch while obtaining immediate feedback (see Appendix Figure 2 & Figure 3). Subsequent engineering included tempo-sync, feedback, filters and parameter smoothing. The ***DelayLine::read*** interpolation, per-sample smoothing strategy, and use of ***juce::dsp*** filters were exercised under different buffer sizes and sample rates to ensure stable, low-latency behavior in real host scenarios.

- **Parameter management, GUI testing and final UI polish**

Parameters are exposed through ***AudioProcessorValueTreeState*** so they support host automation, preset recall and editor attachments. The custom ***LookAndFeel*** and ***RotaryKnob*** components were iterated in Xcode while driving the plugin with the MiniMoog source in both the JUCE host and real DAWs, verifying that tempo-sync followed host BPM and that attachments updated audio parameters correctly. ***LevelMeter*** and Measurement atomics were validated visually (60 Hz GUI timer) and for thread safety. Final assets and fonts were embedded via BinaryData to simplify distribution and make builds reproducible across VST3, AU and Standalone targets.

- **Real-time safety, QA testing and packaging for AU/VST3/Standalone**

Real-time-friendly choices were validated throughout. Allocations happen at ***prepareToPlay*** or editor setup, audio-thread code avoids locks and uses atomics for cross-thread reporting, and debug checks (protectYourEars) were used during QA. The QA cycle included DAW compatibility tests (buffer-size sweeps, sample-rate sweeps, automation recall), stress tests with rapid GUI changes, and listening tests using both the MiniMoog and real audio tracks. Packaging used the Projucer/Xcode build outputs to produce final AU (see Appendix Figure 4) and VST3 bundles and a macOS Standalone app; macOS builds were not code-signed or prepared for notarization per Apple requirements due to the academic only nature of the project.

Discussion

- **Evaluation**

Functionally the plugin meets the specified requirements. Tempo synchronization works when the host DAW supplies BPM, and the tempo-to-ms conversion produces sensible musical subdivisions. Fractional interpolation produces smooth delay reads when the delay time changes or is modulated. The filters in the feedback loop control tone and prevent runaway resonances. The UI behaves intuitively: the tempo-sync toggle switches between numeric delay time and note-choice controls, and the level meter reflects true dB levels & audio peaking. Performance-wise, the design avoids heap allocations in processBlock and keeps per-sample work to simple arithmetic and filter calls, which is appropriate for typical CPU budgets in DAWs.

- **Successes**

The primary successes are the reliable tempo synchronization when available, robust parameter smoothing that prevents audio artifacts, a reusable UI component model, and safe cross-thread metering. The DelayLine interpolation reduces audible artifacts when delay time is changed, and the project demonstrates sensible default fallbacks, making the plugin usable across different hosts supporting VST3 and the Apple AU format along with a stand-alone use feature.

- **Challenges**

Key challenges included ensuring no dynamic memory or expensive operations occur on the audio thread, implementing correct buffer wrapping and index handling for interpolation, and handling host variability in providing transport information. Thread-safety for GUI metering required atomic operations and a design that avoids blocking the audio thread. Debugging numeric edge cases (NaN/Inf or extreme values) prompted the addition of defensive code that is active in debug builds.

- **Limitations and Future Work**

There are several logical next steps and limitations to note. Tempo sync relies on a host-supplied BPM; some hosts do not provide this consistently, so a MIDI (musical instrument digital interface) clock or internal tappable tempo button could be added as additional sources. In its current state the GUI is at a pre-selected non scalable size. The interpolation routine is a good compromise for quality and cost, but offering selectable interpolation algorithms could let users trade CPU for fidelity. Finally, adding a pre-set management system, improved testing (unit tests for DSP routines), redesigning the GUI for a more professional look with possible GUI scaling, and addressing platform-specific code signing for macOS would make the project more production-ready.

Conclusion

The project achieved its goals of delivering a tempo-aware, stereo delay with a clean UI and attention to real-time academic constraints while completing the academic object. In its current development, three available plug-in options include the industry standard VST3, Apple AU, and a stand-alone version use case feature which allows for use in live performance and demonstration modes. This allows for a wide range of compatibility with most commercially available DAW's. The resulting plugin is practical for both studio and live contexts, with an architecture that is maintainable and extensible.

References

Introduction to Signal Processing: by Sophocles J. Oranidis
link: introduction to signal processing

The Scientist & Engineer's Guide to Digital Signal Processing: by Steven W. Smith
link: https://users.dimi.uniud.it/~antonio.dangelo/MMS/materials/Guide_to_Digital_Signal_Process.pdf

Understanding Digital Signal Processing: by Richard G. Lyons

Think DSP: by Allen B. Downey
link: https://greenteapress.com/thinkdsp/thinkdsp.pdf

Designing Audio Effect Plugins in C++: by Will C. Pickle

The Complete Beginner's Guide to Audio Plug-in Development: by Matthijs Hollemans

Audio Effects: Theory, Implementation and Application: by Joshua D. Reiss & Andrew McPherson

The C++ Programming Language 4th Edition**:** by Bjarne Stroustrup

The C Programming Language 2nd Edition**:** by Brian W. Kernighan & Dennis M. Ritchie

cppreference.com
link: https://www.cppreference.com

JUCE
link: https://juce.com/tutorials/tutorial_rectangle_advanced/

Musicdsp.org
link: https://www.musicdsp.org/en/latest/

Evan, M. J. (2026). plug-in phases [Photographs]: Personal collection

Appendix

Figure 1):

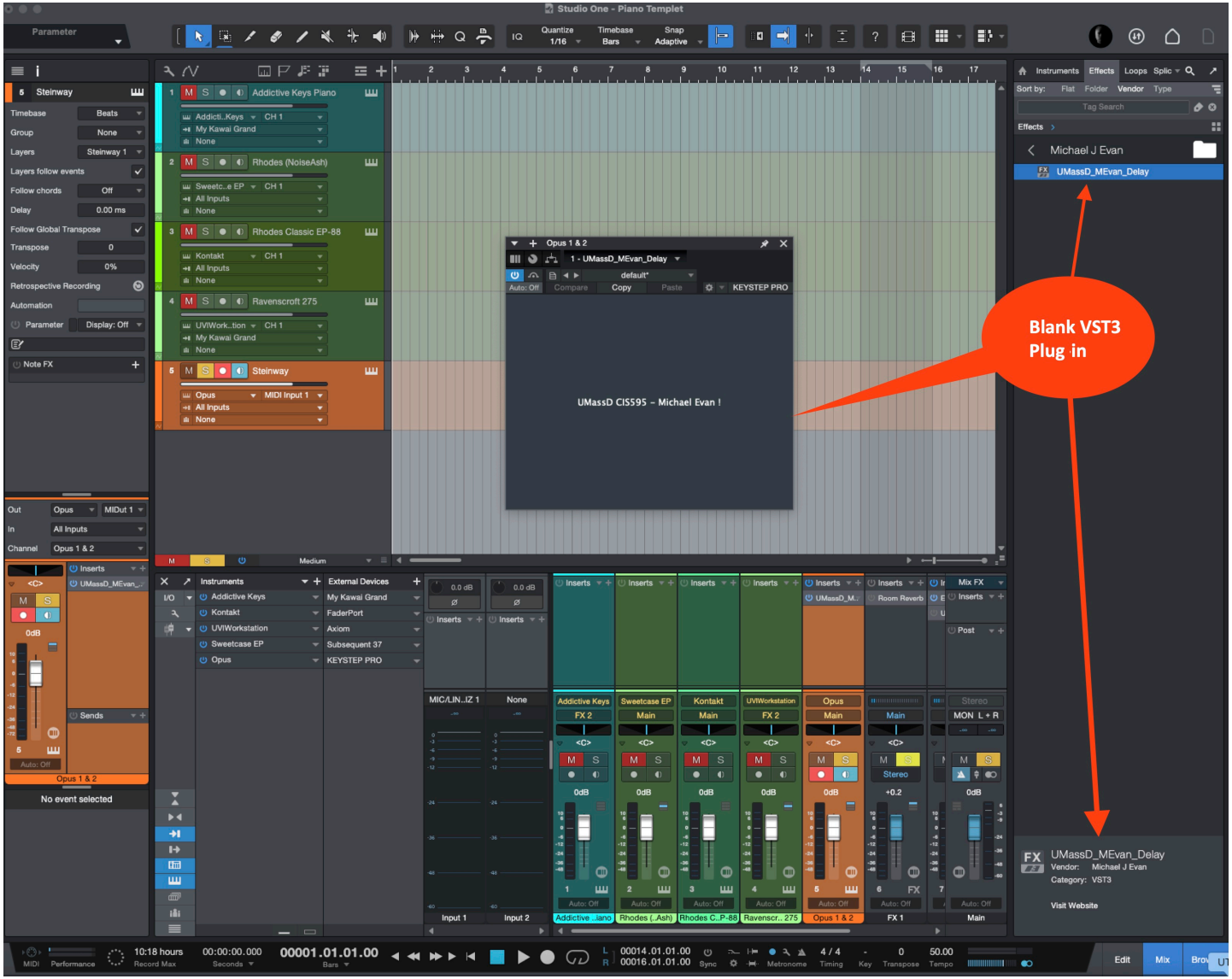


Figure 2):

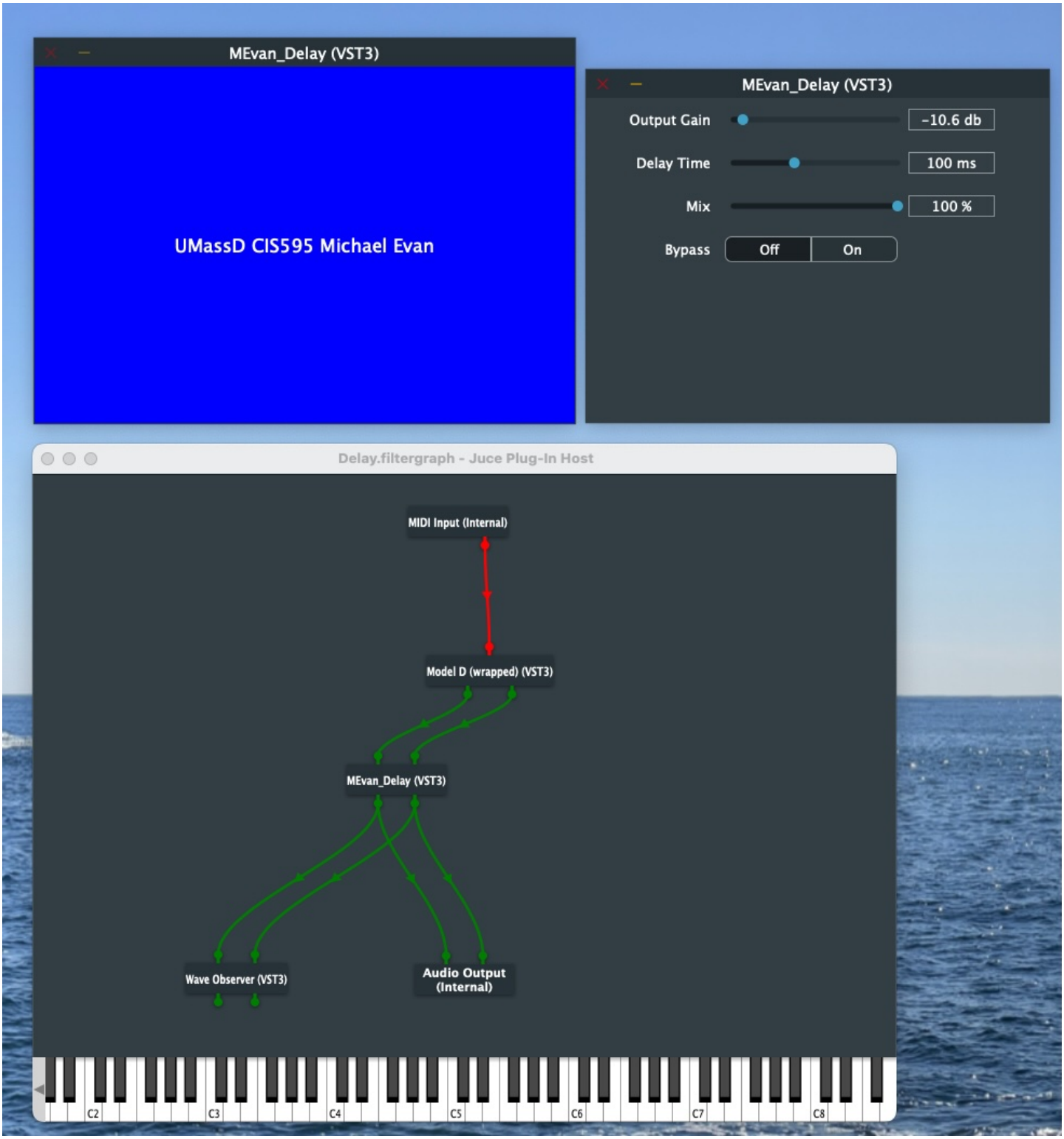

MEvan_Delay (VST3)
UMassD CIS595 Michael Evan
MEvan_Delay (VST3)
Output Gain
-10.6 db
Delay Time
100 ms
Mix
100 %
Bypass
Off
On
Delay.filtergraph - Juce Plug-In Host
MIDI Input (Internal)
Model D (wrapped) (VST3)
MEvan_Delay (VST3)
Wave Observer (VST3)
Audio Output (Internal)
C2
C3
C4
C5
C6
C7
C8

Figure 3):

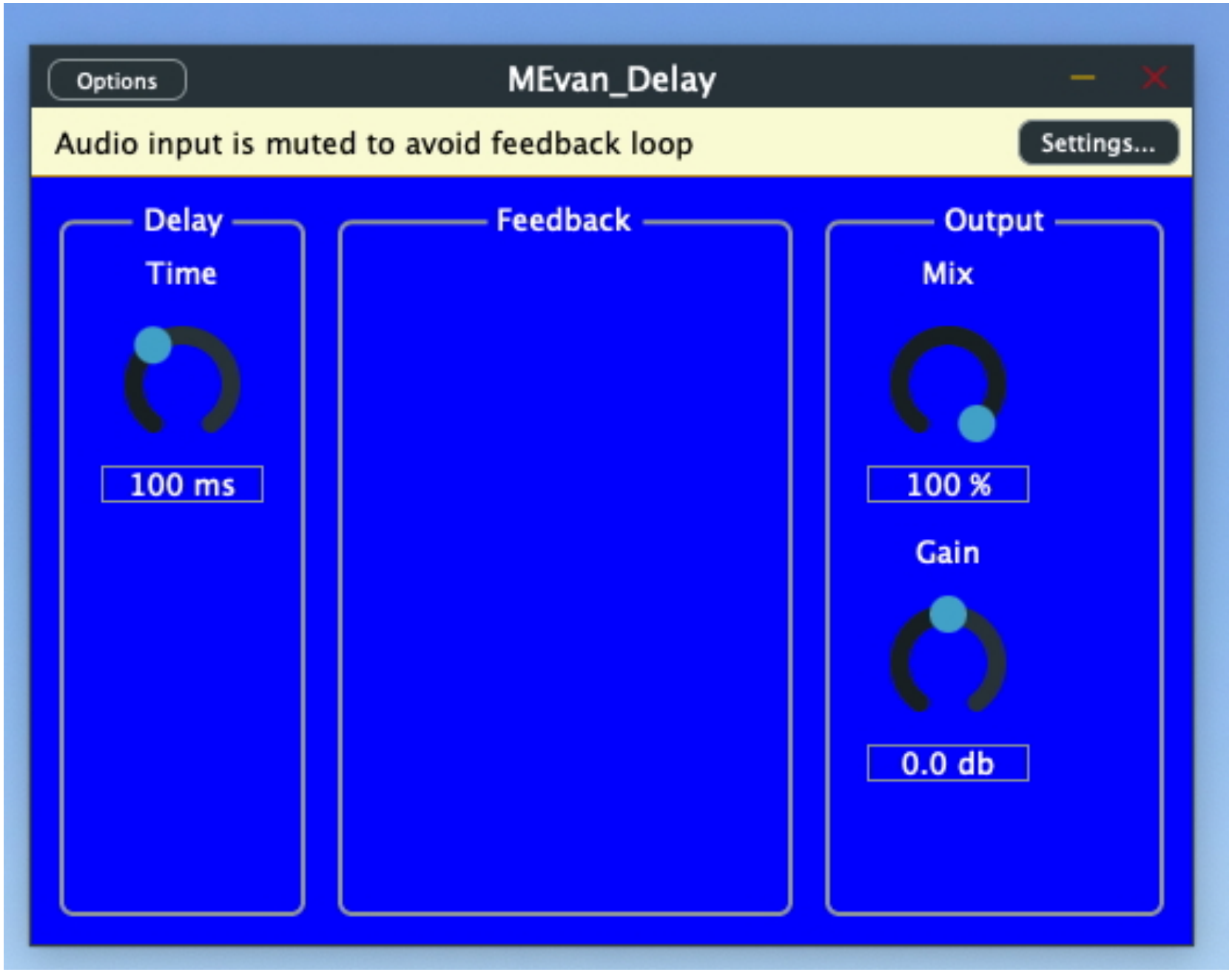

Options
MEvan_Delay
Audio input is muted to avoid feedback loop
Settings...
Delay
Time
100 ms
Feedback
Output
Mix
100 %
Gain
0.0 db

Figure 4):


Stereo Out
Manual
Compare
Editor
MEvan UMassD CIS 595
Delay
Note
1/4
Sync
Feedback
Feedback
52 %
Stereo
100 %
Low Cut
20 Hz
High Cut
20.0 k
Output
Mix
84 %
Gain
-1.3 dB
6
0
-6
-12
-18
-24
-30
-36
-42
-48
-54
-60
MEvan_Delay