Microbiome: A real-time audio effects plugin

Darwin Do

Advised by Dr. Scott Petersen

Department of Computer Science, Yale University

May 2024

Abstract

In this report, I present *Microbiome*, a VST3/AU/LV2 real-time audio effect plugin that uses looping and delay to aid in the creation of interesting soundscapes for use in music production and sound art. *Microbiome* was written in C++ with the JUCE framework, allowing for multi-platform exports and rapid development with the supply of many basic pre-built DSP algorithms. *Microbiome* features a user interface for parameter control and allows for parameter automation/modulation from the host DAW. The core audio processing algorithm consists of a series of "colonies" that resample audio from a global delay line managed by the "Microbiome Engine". Each colony has its own processing pipeline that outputs data back to the engine to be combined into the output signal. If used within the context of a host DAW, the output signal can then be fed into further audio processing plugins. *Microbiome* was tested on the Windows version of Ableton Live 11 but maintains compatibility with any DAW that supports VST3 on Windows, LV2/VST3 on Linux, and AU on macOS. The macOS AU was built on macOS Monterey version 12.3. Code for *Microbiome* is completely open-source and all binaries can be found on the GitHub page linked in the conclusion section. The following sections describe an introduction to digital audio plugins, my design goals in creating *Microbiome*, the implementation details of the plugin, and a discussion of factors I had to consider when designing for use in a real-time audio host application. Overall, *Microbiome* is another audio tool, in the plethora of many, that sound artists and musicians can use to generate sounds to their liking.

Introduction

The introduction of software Digital Audio Workstations, or DAWs, has allowed for a greater democratization of music production. No longer is it required to have access to an expensive physical recording studio to create quality production-level sound. With the right software and enough creativity, motivated individuals can create professional-quality music right in their bedrooms[6].

The popularity of software DAWs such as Ableton, Logic, FL Studio, etc. has also brought a demand for the development of the audio plug-in. The software instruments and audio effects pre-loaded in each DAW can only provide so much creative liberty. Audio plug-in architectures such as Steinberg's Virtual Studio Technology (VST), Apple's Audio Units (AUs), or the open-source LV2 framework allow developers to create software instruments/audio effect units in a standardized interface and export them for use in any DAW that supports said interface.

While the multiple plugin architectures vary in API and implementation, they all follow the same basic principles for processing (or generating) audio. Audio I/O is streamed through the plugin in a series of multichannel blocks of samples. Additional control information such as plugin-specific parameters and MIDI data are also supplied alongside the audio blocks. The audio plugin performs the desired digital signal processing (DSP) on the audio and returns a view of the modified data the host environment can use[3].

An unfortunate side effect of the numerous amount of plugin architectures is that there are a plethora of conflicts between software licenses, ease of use, and host compatibility. While the VST3 library is open-source, distributing a VST3 in binary form requires signing the *Proprietary Steinberg VST 3 License*[1]. Apple's many DAW offerings such as Garageband or Logic Pro only support the Apple Audio Unit architecture, requiring plugins in other architectures to be rewritten specifically for Apple devices[7]. While there are numerous open-source standards such as the Linux Audio Developer's Simple Plugin API V2 (LV2) or the more recent CLever Audio Plug-In API (CLAP)[8], there is no overarching global standard that guarantees compatibility with the majority of DAW hosts.

JUCE is a C++ audio framework that solves many of the aforementioned problems. Users building audio applications can implement the processing logic within JUCE's API and target multiple architectures and operating systems for export. This allows developers to

focus their efforts on the signal-processing portion of their algorithms instead of worrying about maintaining compatibility across different environments[4].

The goal of this project is to create an audio plugin via JUCE that can be used in any mainstream DAW (Ableton, Arduor, Live Pro, etc.) on any platform (Windows, MacOS, Linux). This is intended to be used as an audio effect plugin for any music producer or sound artist to use in their workflow as a way to synthesize interesting sounds.

Design Goals: The Plugin-as-an-Instrument

One interesting paradigm within the audio plugin world is the concept of a plugin instrument versus a plugin effect. Instruments generate sound. They use oscillators to produce a stream of audio data that is output back to the host in a series of blocks. Traditional effects on the other hand do not generate sound by themselves. They take in audio data from the host, apply some processing, and output that processed data back for playback or further consumption by the host.

Plugin architectures support both the *effect* and *instrument* paradigm. Within JUCE, the core of all audio plugins, both effects and instruments, are implemented in the *AudioProcessor* class. With how the preprocessor tags are set, switching between the instrument and effect paradigms is done easily by toggling a single flag in the Projucer, the JUCE project configuration manager.

One of the main design methodologies of *Microbiome* is to explore this paradigm even further. Similar to how musicians may tweak their guitar pedals to create a more dynamic sound during a live performance, could a virtual audio effect plugin be used as an instrument? *Microbiome* attempts to create a plugin-as-an-instrument by treating the plugin's input as its sound generation source. The plugin's name comes from the idea of viewing sound as an evolving creature viewed through a biological lens. As audio is played, snippets of sound "branch off" from the main source and forge their own paths. *Microbiome* acts as an instrument by playing these sound branches in combination with the original, continually changing sound source.

Implementation

Implementation - Engine

At a high-level overview, *Microbiome* can be seen as a sort of hybrid between a delay and a looper effect. When the host supplies audio data to the effect plugin, the audio buffer goes through the *MicrobiomeEngine*. The engine is composed of a 30-second audio buffer alongside an array of *Colonies*. Each Colony contains a 10-second buffer and a series of processing parameters/states. There are a maximum number of four colonies, and each colony can be enabled/disabled at will.

The audio buffer within the engine is used as the global delay line for each colony to pull from. As audio passes through the engine, samples are continuously pushed onto the delay line in a ring-buffer-like fashion. The size of the buffer determines how many samples in the past the delay line can hold. While each sample is stored in the delay line, the current sample is combined with the output of every enabled colony. The ratio of the original signal versus the processed signal from the colony can be controlled via the **engine wetness** parameter. Once each sample in the current processing block has been combined with the output from each of the colonies, a reverb effect is applied to the combined signal to mesh everything together. The reverb algorithm is implemented by JUCE and is based on the "FreeVerb" Schroeder reverberator[5]. The room size and wetness of the reverb can be controlled via the **engine reverb** and **engine lush** parameters respectively.

Implementation - Colony

Each colony populates its internal buffer by resampling data from the global delay line in the engine. The **sample speed** parameter determines the sampling rate of the colony audio, allowing the user to speed up/slow down the playback from each colony. The point in time the colony reads from the delay line is determined by the **resampler seek** parameter and the manner of resampling is determined by the **colony cultivation mode** parameter. There are three different cultivation modes:

1. LOOP
2. REGENERATE
3. FOLLOW

LOOP mode fills the colony buffer with resampled data until it is full. The buffer remains static and is not resampled unless the resampling speed or seek are changed.

REGENERATE mode is similar to LOOP mode with the exception that after a certain amount of time, the buffer will "regenerate" and resample fresh data from the delay line regardless of any user parameter changes. The triggering of a buffer regeneration is determined randomly at the beginning of each block's processing cycle within the colony.

FOLLOW mode continuously resamples the delay line without stopping. It effectively turns the colony's output into a sped-up/down version of the audio signal at a previous point in time.

The colony keeps track of the sample index to output from its internal buffer. Once each colony has finished reading the last sample in its buffer, it loops back to the beginning of the buffer read point and starts to iterate from there again. The start/end points of the buffer read indices can be controlled via the **end** and **start** parameters within the Colony Controls panel. This gives the freedom of either using the full length of the buffer or looping through a smaller subset of it. Note that while playback of the colony buffer may only loop through a certain subsection, buffer resampling is done at the scale of the full buffer length.

There are additional effects that are applied to the resampled audio in each colony. A Moog 24db lowpass ladder filter is applied to the signal followed by a compressor to keep levels reasonable. The filter's cutoff frequency can be modulated via the **filter cutoff** parameter.

When the colony returns a single sample to the engine, it can also combine its output with multiple other samples from the colony buffer as "ghost" samples. Each ghost sample is chosen at a predetermined random index before the colony's internal buffer read index. These predetermined indices are randomly shuffled around at the beginning of each colony processing cycle to create a sense of life and continuous evolution within the colony's output. When indices change, they are linearly smoothed to prevent abrupt disruptions within the audio signal. Since decimal indices can't be used in the context of discrete samples, the value at each ghost sample index is linearly interpolated between the two closest samples. The number of ghost samples played back in each colony output sample is determined by the **sample ghosts** parameter.

Finally, a **gain** parameter is provided to allow the user to control the volume of each colony's output.

Implementation - UI

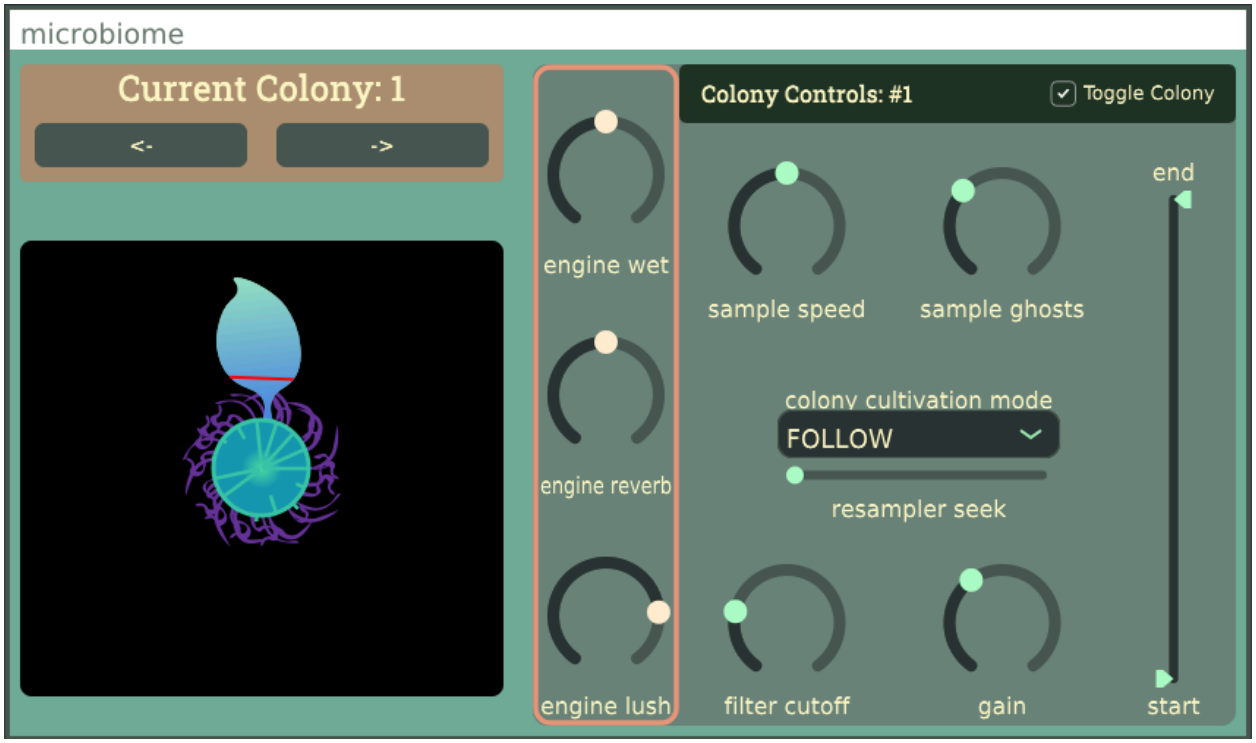

Figure 1.1: Microbiome editor UI with focus on an enabled Colony #1

The user interface visualizes the current state of the Microbiome engine and allows for user control of the different plugin parameters. The right side of the UI contains the engine control parameters and the colony control box. The control box contains all parameters regarding the colony's processing algorithm with a toggle switch on the top that allows the user to enable/disable the currently selected colony. Most parameter controls are implemented as single-value rotary sliders except the cultivation mode and buffer start/end offsets which are controlled via a menu box and a 2-value slider respectively.

The currently selected colony can be cycled through by clicking on the arrow keys in the colony selector box at the top left of the interface. Disabled colonies have their controls disabled underneath a semi-opaque filter, indicating that the colony does not affect the

output. The visualization window below the colony selector box shows a pattern reflecting the current state of the engine.

Figure 1.2: Microbiome editor UI with focus on a disabled Colony #1

Figure 1.3: Microbiome editor UI with three enabled colonies at varying resampling seek positions and buffer start/end points

Enabled colonies are depicted as leaf-like objects branching off the central node, suggesting a living, evolving creature. The start and end limits of the colony's buffer are visualized through a color gradient on the leaf object. A red "playback" line sweeps through the gradient sections of the leaf, symbolizing how each colony is simply looping through a section of its internal buffer. Changing the **resampler seek** parameter for a colony changes its rotational position around the central node. When manipulating these important colony parameters, these graphics provide visual clarity to the user in a unique and thematically consistent manner.

All graphics and animations are implemented via built-in JUCE functionality. The main audio processor class creates a PluginEditor object which contains child component objects to draw. Every child component contains an overridable paint method that is called when the component needs to be redrawn in the window. The Microbiome editor is split into two main child components: the MicrobiomeWindow and MicrobiomeControls. The controls object handles drawing and initialization for all the parameters while the window object handles drawing the visualization window. To animate the window, the paint method is explicitly called on a timer callback that occurs 24 times a second.

Discussion

Many small details had to be taken into account when designing the architecture of *Microbiome*. The processing of each sample in the engine and colony is done by the audio thread which must complete its block processing as fast as possible. This means expensive operations such as buffer allocations have to take place in the prepare() functions which take place when the plugin is initialized.

Maintaining a fast audio thread also means blocking locks have to be avoided in the processing functions. However, the plugin parameter values need to be accessed by both the audio thread for use in processing as well as the UI thread for allowing user interaction. To prevent any potential race conditions, we use the built-in AudioProcessorValue tree to store the underlying values of all our parameters. This uses the ValueTree data structure provided by JUCE to synchronize access to our parameters, allowing for lock-free atomic loads of values in the audio thread and automatic UI updates for parameter controls. These automatic UI updates allow the interface to reflect any parameter changes from host automation.

Figure 2.1: Microbiome used on an "Acid Bass" instrument in Ableton Live. The red lines show the automation path of the "Resonance" parameter of colony 1.

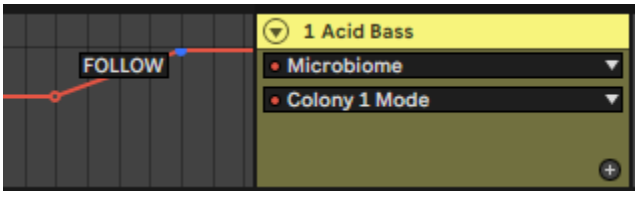

Figure 2.2: Microbiome used on an "Acid Bass" instrument in Ableton Live. The red lines show the automation path of the cultivation mode parameter of colony 1. Ableton maps the discrete values of each mode onto the continuous automation space.

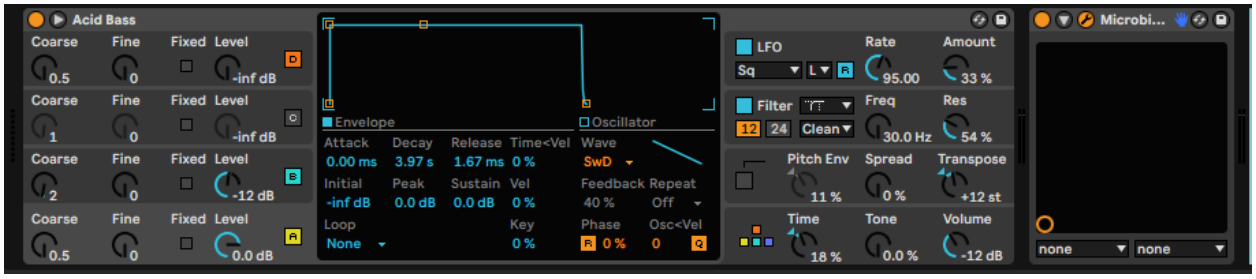

Figure 2.3: The effects rack of the Acid Bass MIDI instrument in Ableton Live with Microbiome at the end of the processing chain. Pressing the orange wrench in the Microbiome window opens the plugin editor.

For the synchronization to be successful, every UI element needs to register an attachment with the underlying parameter object that it is connected to. JUCE provides a set of attachments to standard UI elements such as rotary sliders, toggle buttons, linear sliders, etc. The **start/end** colony parameters, however, are controlled via a two-value slider which has no attachment class provided by JUCE. After browsing the JUCE forums, I ran into user Nikolay Tsenkov who shared his implementation of a two-value slider attachment class[2]. I was able to modify Nikolay's code to work within the context of the Microbiome parameters, saving me lots of time.

Another benefit of using the ValueTree objects is that the parameters are easily transformed into XML for saving state. This allows hosts to save the values of each parameter so that its values are not lost every time the host closes. State saving and loading are implemented in the MicrobiomeAudioProcessor class.

Conclusion

Microbiome is a VST3/LV2/AU audio plugin made with JUCE that uses delays and looping to create unique and interesting soundscapes from an input audio source. The incoming audio is treated as a living, evolving creature. Snippets of the past can branch off and form their own little “colonies”, feeding back into the source a modified signal that is closely related, but not quite the same.

Each colony contains a processing pipeline consisting of audio resampling, low-pass filtering, compression, and addition of noise. All colony output is combined with the original signal and fed through a reverb processor for output. Many of the parameters in this pipeline are modifiable by the user through the plugin UI or host application.

There is a lot of room for creative liberty. *Microbiome’s* results range from long drawn-out ambient drones to fast-paced jittered noise. It is one more tool, in the plethora of many, that musicians and sound artists can equip to fully utilize the power of digital audio processing.

A video demo of the plugin can be found at:
https://www.youtube.com/watch?v=Ihh65Blad9o

The source code for the plugin can be found at:
https://github.com/dsmaugy/microbiome/tree/main

References

1. What are the licensing options [Internet]. 2024;Available from: https://steinbergmedia.github.io/vst3_dev_portal/pages/VST+3+Licensing/What+are+the+licensing+options.html

2. Tsenkov N. Using the slider attachment class with 2-value sliders [Internet]. Available from: https://forum.juce.com/t/using-the-slider-attachment-class-with-2-value-sliders/18411

3. Goudard V, Muller R. Real-time audio plugin architectures a comparative study [Internet]. 2003. Available from: http://recherche.ircam.fr/equipes/temps-reel/movement/muller/xspif/pluginarch.pdf

4. JUCE [Internet]. Available from: https://juce.com/

5. Smith JO. Physical Audio Signal Processing [Internet]. Available from: https://ccrma.stanford.edu/~jos/pasp/Freeverb.html

6. Fagnoni F, Morales J. Digital Audio Workstations—The Infrastructure of Music Production [Internet]. 2019;Available from: https://mastersofmedia.hum.uva.nl/blog/2019/10/24/digital-audio-workstations-the-infrastructure-of-music-production/

7. About third-party Audio Units and external device compatibility in Logic Pro and Final Cut Pro on Mac computers with Apple silicon [Internet]. 2023;Available from: https://support.apple.com/en-us/102082

8. CLAP: The New Audio Plug-in Standard [Internet]. Available from: https://u-he.com/community/clap/