Eötvös Loránd University

Faculty of Informatics

Department of Programming Languages and Compilers

Digital Signal Processing Plugin for Multilayered Synthesis

Supervisor:

Dr. Viktória Zsók

Assistant Professor

Author:

Evan Sitt

Computer Science BSc

Budapest, 2021

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:
Student's Name: Sitt Evan Hofung
Student's Neptun code: C3JI5D

Course Data:
Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Viktória ZSÓK*
Supervisor's Home Institution: *ELTE Informatikai Kar*
Address of Supervisor's Home Institution: *H-1117 Pázmány Péter stny. 1c Budapest*
Supervisor's Position and Degree: *Lecturer, Ph.D*

Thesis Title: Digital Signal Processing Plugin for Multilayered Synthesis

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Sound synthesizers have been pivotal to the development of the music industry ever since their inception. Software synthesizers offer advantages in flexibility and modularity, allowing for ease of customization by the end-user. Typical software music production involves digital signal processing (DSP) plugins usage. DSP plugins receive input data streams from Audio Stream Input/Output (ASIO) devices and Musical Instrument Digital Interface (MIDI) devices, which the DSP plugin then modifies and manipulates. DSP plugins are hosted by a Digital Audio Workstation (DAW). This is a digitalized version of an analog workstation featuring similar signal flow. The DAW provides the primary user interface while also handling communication with the ASIO drivers and with MIDI devices. The DAW handles final signal manipulation returning a final audio output stream. This project will implement a DSP plugin, using the Virtual Studio Technology 3 (VST3) interface standard. VST3 is an interface standard made to handle MIDI and ASIO while allowing the developers to create custom graphical user interfaces. The project will handle MIDI input and generate a polyphonic multilayered synthesizer waveform via the use of wavetables, combining both additive and subtractive synthesis. The implementation of the project will be accomplished with the use of the JUCE framework, a popular C++ development framework for audio processing plugins. The application will prioritize end-user customizability, cross-platform compatibility, and a strict low latency processing time. The application will be hosted by any VST3 compatible DAW, or used as a standalone synthesizer application. End-users will be able to integrate this DSP plugin into their workflow, either with multiple instances of this DSP plugin or in tandem with other DSP plugins. This DSP plugin will serve sound designers, audio engineers, and mixing engineers as an all-in-one, flexible, and easily customizable digital synthesizer.

Budapest, 2019.11.25.

Contents

Introduction

The thesis is an entry level professional ready digital synthesizer application for use by music producers and composers. This digital synthesizer application allows for flexible customization of the synthesizer sound with usage and integration into an existing workflow within a Digital Audio Workstation. It is possible to use the digital synthesizer application as a standalone as well. This application is made to be cross-platform compatible.

The application is built within the JUCE framework, a C++ application framework. It is packaged both as a Virtual Studio Technology 3 standard plugin, for use within a compatible digital audio workstation, and as a standalone application. Developers who are familiar with the JUCE application framework may also elect to build the application to create plugin versions compatible with the RTAS, AAX, and AU standards.

This digital synthesizer can be used with live MIDI input or MIDI notation. The synthesizer supports multilayer polyphony of waveforms generated up to hardware supported limits. Audio output is produced as PCM WAV via Audio Stream Input/Output drivers and supports low latency buffer sizes. The digital synthesizer can support additive and subtractive synthesis with up to three basic waveforms. A signal post processing chain contains the effects: LFO, Envelope, and Filters.

The chapters are as follows: *Background* (chapter 1) covers the theoretical background necessary for developing and using the application, the *User Documentation* (chapter 2) covers the user of the plugin within the FL Studio DAW, and finally the *Developer Documentation* (chapter 3) covers the necessary knowledge for implementing the code with the JUCE framework.

Chapter 1

Background

1.1 Audio basics

Before we delve into the background for Digital Synthesis and Digital Signal Processing, it behooves us first to have a brief overview of audio. Audio typically deals with the way we create a sound, the way our ears receives the sound, and the way our mind perceives the sound. While this chapter will provide a brief overview of audio basics, more detailed information can be found at [12] and [13].

1.1.1 Waveform

We start with the basic building block of sound, the waveform. A *Waveform* is an abstract way of representing a sound. Typically a waveform is visualized on the Time Domain with Amplitude as the range containing a single cycle of a waveform. Time Domain of a waveform is typically represented in seconds. The Amplitude range, on the other hand, can be represented in many different units. The most common unit used is the decibel (dB). A sample waveform representation with correspondence to sound pressure is illustrated by figure 1.1.

In the application, four waveforms are utilized: the *Sine*, *Sawtooth*, *Square*, and *Triangle* waveforms. These are explained in more detail in the *User Documentation* (chapter 2) and the *Developer Documentation* (chapter 3).

1.1.2 Decibel

A *Decibel* is a relative unit of measurement representing the ratio of the value of a power quantity to another. For example, if x represents the ratio of the power

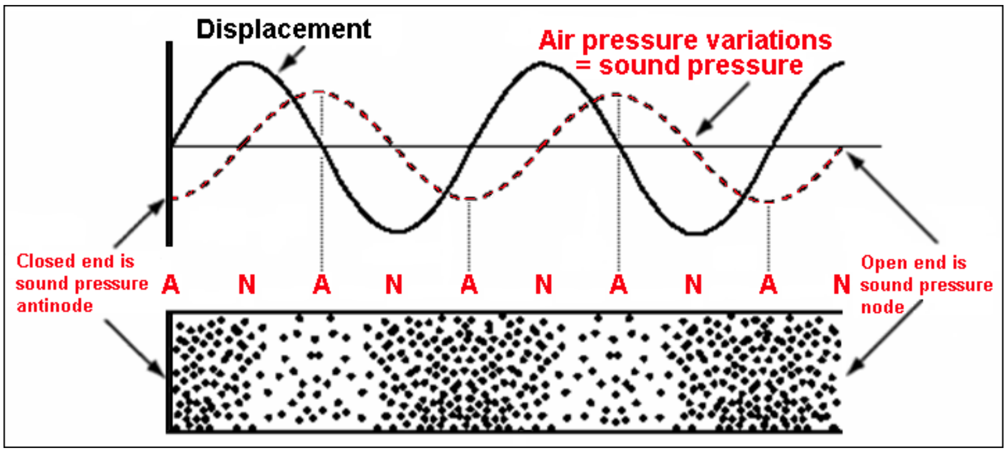
[h] 
Displacement
Air pressure variations = sound pressure
Closed end is sound pressure antinode
A
N
A
N
A
N
A
N
A
N
Open end is sound pressure node

Figure 1.1: Sound pressure and particle displacement [16]

quantity of one system to another system, then the decibels can be calculated via $10log_10x$. Since the decibel is a relative unit, it is always essential to establish what the reference measure of power is.

In colloquial usage, when one refers to the loudness of a jet-plane engine as +80dB, this is in reference to the lower threshold of sound pressure that is perceivable to the human hearing system. In the case that a waveform is referencing the lower threshold of human hearing, then the range is typically visualized with "0" of the range as 0dB and the values above the horizontal axis going up in decibel value and values below the horizontal axis going down in decibel value. A negative value is representative of the vibration of a membrane, such as an eardrum, in reaction to negative sound pressure. This is illustrated in figure 1.2.

1.1.3 Decibels in digital synthesis and audio engineering

In contrast to the colloquial usage of decibels, within audio engineering, the decibel value used is called the Decibels Relative to Overload. In this variant, 0dB is assigned to the maximal power level that can be output by a system before clipping occurs. *Clipping* is the term to describe the type of distortion that occurs when a signal overshoots the maximal output. For example, a speaker representing a sound pressure of 0dB will be outputting the signal at its maximum power, and representing the sound pressure at half power would be -6dB. In hardware, safeguards are implemented at certain decibel levels to curb a signal before it clips.

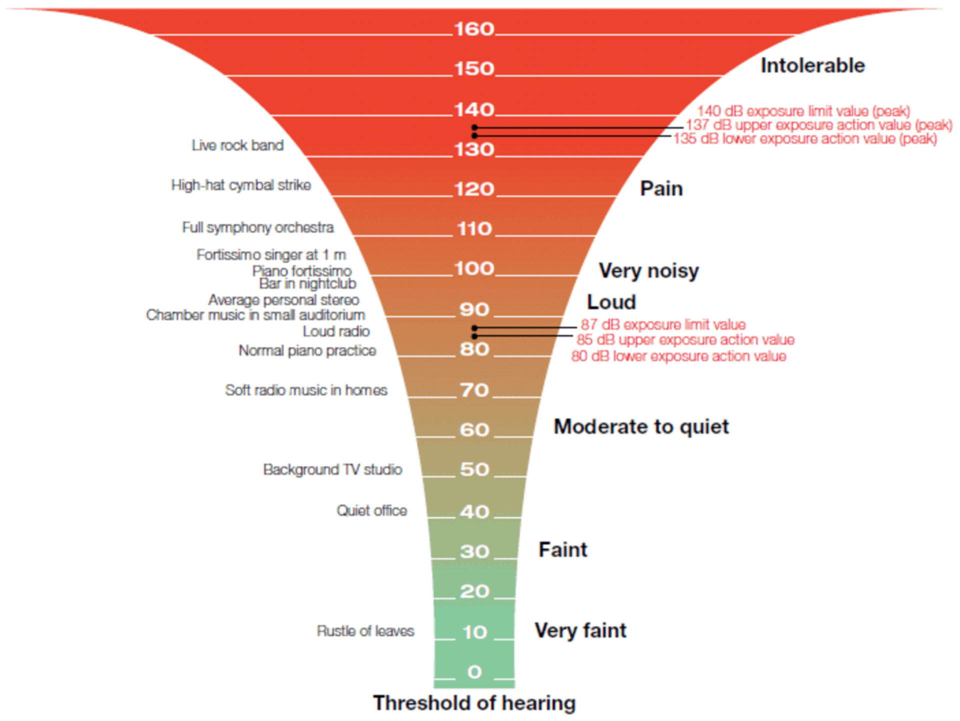


Figure 1.2: Decibels scale with typical comparisons [17]

1.1.4 Digital representation of decibels

Within a Digital Synthesis system, the power of a signal is represented using `float` data types. Typically, 1.0 would represent the maximal power output, 0.0 would represent the minimum power output, and -1.0 would represent the maximal power output with negative sound pressure. Sound pressure levels between maximal and minimal are represented as decimal values between 0.0 and 1.0. As with any floating-point value stored within a digital context, the precision of this value is highly dependent on the number of bits used to store the floating-point value. For the `PCM WAV` standard used for the majority of digital audio synthesis done on computer systems, 8-bit, 16-bit, and 32-bit floating-point values are commonly used.

This concept is used continuously during development, as described in chapter 3 and play a crucial part in the Testing section as well, most particularly during the *Value Tests* (subsection 3.7.2).

1.1.5 Frequency domain

The other commonly used representation of sound is the *Frequency Domain* representation. Here the frequency is the domain, while amplitude remains as the range. In contrast to the Time Domain, there are no time values within the Frequency Domain. As such, the view of the Frequency Domain graph represents the frequency spectrum and amplitude values at an instantaneous point in time.

Frequency refers to the rate at which a sound repeats a single cycle of its periodic function. The frequency is measured in hertz, which represents $cycles/seconds$. Human hearing is capable of perceiving sound with frequencies between 20Hz and 20kHz. While the actual range of any one person's audible range will vary depending on their age, health, and several other factors, the majority of sound reproduction systems will operate within the full range between 20Hz and 20kHz.

1.1.6 Frequency spectrum

The graphical representation of the Frequency Domain of a sound is usually referred to as the *Frequency Spectrum* or the *Spectral Graph.* These graphical representations will typically represent frequencies between 20Hz and 20kHz, with amplitudes from 0dB to -∞dB. The varying values of the Frequency Spectrum of any waveform control the timbre, or sonic characteristics, of the resulting sound.

A variation on the Frequency Spectrum that is used in digital audio synthesis is the *Harmonics Graph.* The Harmonics Graph also represents the Frequency Domain of a waveform. However, the domain represents values from 0 to ∞. These do not represent any specific frequencies; instead, the values in the domain represent multipliers of a base frequency. Integer multiplier values are referred to as the *harmonics* of a sound. A Harmonics Graph represents the unique character of a specific sound-producing object, such as a person's vocals chords, regardless of the frequency of the sound being produced. Harmonics Graphs are commonly used for physical modeling of instruments, which is then later used in reconstructing that sound in Digital Synthesis with the weighted summation of primitive sine waves. An example of a harmonics graph, in this case of a cello, is shown in figure 1.3.

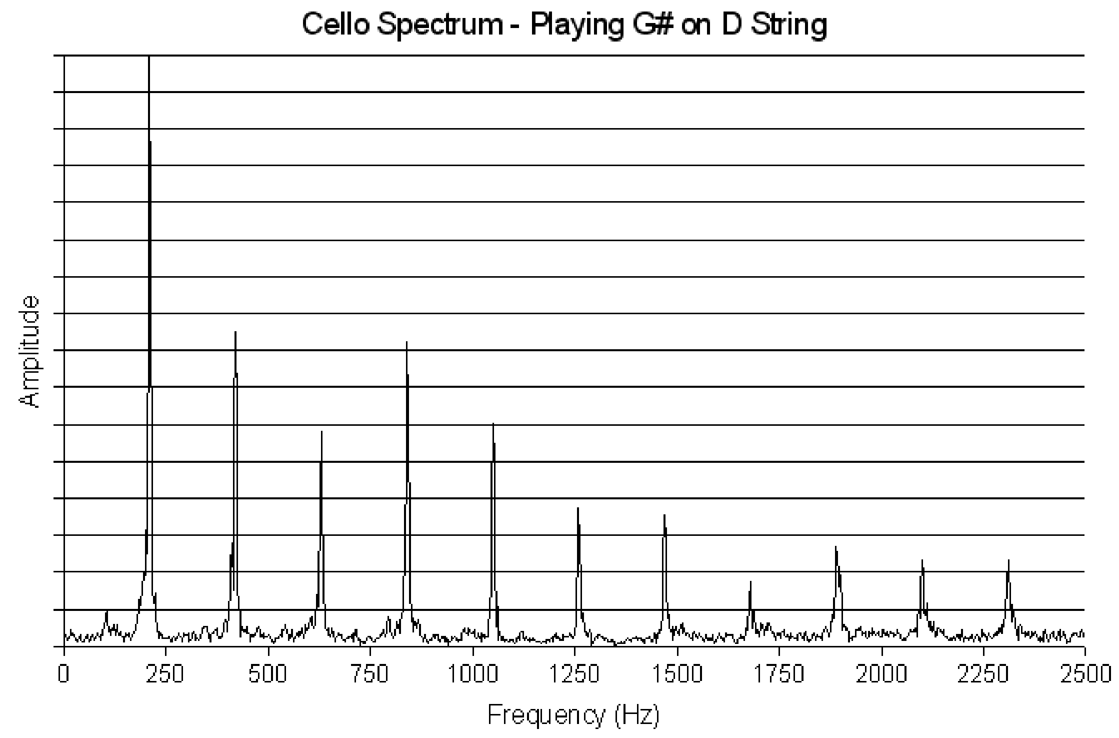


Figure 1.3: Harmonics graph of a cello [18]

1.2 Digital synthesis

Digital synthesis is a field in which an audio engineer utilizes digital signal processing techniques to make musical sounds. One of the most prominent early examples were the keyboards manufactured by Yamaha, which employed their proprietary FM Synthesis algorithms back in the 1970s.

A frequency synthesizer refers to a synthesizer with circuitry capable of generating a range of frequencies from a single reference frequency input. One typical application of a frequency synthesizer is the use of a Fourier Series to construct waveforms via Additive Synthesis.

1.2.1 Direct digital synthesis

When designing a digital synthesizer, there is a multitude of techniques available to the designer for generating frequencies, including Phase Locked Loop, Digital-Analog Conversion, and Direct Digital Synthesis. This project approaches the problem of digital synthesis by using *Direct Digital Synthesis.* Direct Digital Synthesis is a technique used by modern frequency synthesizers. Direct Digital Synthesis utilizes

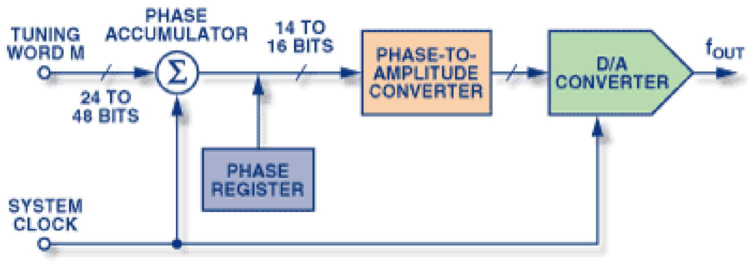


Figure 1.4: Direct digital synthesis signal flow [19]

a singular fixed-frequency reference clock as a stable time basis for the entire system. In contrast to the alternative Phase-Locked Loop system, where the output phase is locked directly to the phase of the input signal, Direct Digital Synthesis holds significant advantages in flexibility, the most prominent being better frequency agility. Frequency agility is defined as the capacity to switch the oscillator between different frequencies without introducing a significant delay in the signal. This advantage is offered as a result of Direct Digital Synthesis using discrete sample operations. Other advantages offered by the Direct Digital Synthesis technique include access to a broader spectrum of frequencies and finer frequency resolution. Additionally, a Direct Digital Synthesis device has the potential to have low overhead and power draw.

A Direct Digital Synthesis device typically consists of a few parts: a digital signal input, a phase accumulator, a phase register, a phase to amplitude converter, a digital to analog converter, and finally the reference oscillator. This signal flow is illustrated in figure 1.4.

This is particularly important during the development of the digital synthesis pieces of the application as discussed in section 3.6.

1.2.2 Digital signal input

The digital signal input acts as the Tuning Word for the system. The demanded frequency value is provided as an input to the system. Additional information that can be provided at this stage of the system includes the type of waveform to be generated, along with any MIDI instructions for the other signal processing modules further down the signal chain.

1.3 Phase accumulator

The *Phase Accumulator* holds the digital number, which represents the current phase of the Direct Digital Synthesizer. The Phase Accumulator iterates the phase with a value controlled by the Tuning Word input. The frequency and sampling rate make up the Tuning Word. Once the phase stored overflows a specified threshold value, modulus mathematics is used to reset the phase within the range specified. With a higher frequency, the Tuning Word will cause the Phase Accumulator to cycle through the phase range more rapidly, in essence accomplishing the task of converting the frequency input into a phase value. In a way, the Phase Accumulator can be thought of as a modulus counter.

1.4 Phase to amplitude converter

After the Phase Accumulator step, the phase value is then passed to the *Phase to Amplitude Converter*. The goal of the Phase to Amplitude Converter is to change the phase value to a digital amplitude value, which can then be further processed. This goal can be accomplished via two different methods: a Phase/Wavetable lookup, or a Waveform Function.

1.4.1 Phase/Wavetable lookup

The *Phase/Wavetable Lookup* method was developed in the late 1970s as a more efficient method of generating waveforms on low spec hardware devices. The concept relies on the efficiency advantage of accessing values from a stored array over repeated mathematical calculations of sinusoidal functions. While the hardware constraints are no longer an issue for modern digital synthesizers, the technique is still used for its relative flexibility when compared to the use of sinusoidal functions.

For the Phase/Wavetable Lookup method, a wavetable is stored within the system. This wavetable is named so since it contains a predefined number of precomputed amplitude values of the desired waveform. Typically this waveform is a fundamental sine wave, as other waveforms can be achieved via Fourier series reconstruction, which relies on additive and subtractive operations rather than trigonometric functions. Additionally, the wavetable is typically stored within a direct-access data structure such as arrays for even better efficiency.

For each phase value received from the Phase Accumulator, the Phase to Amplitude converter will use the Phase Register to convert the phase to its respective index value in the wavetable. The amplitude value stored at this index value is retrieved from the wavetable and returned. Since the index value generated typically falls between two integer values, usually some form of numerical interpolation is used. The final amplitude value will then be passed on to the rest of the digital signal processing chain.

The Phase/Wavetable Lookup method holds significant advantages that contribute to its popularity and continued usage. The efficiency of constant time complexity generation of each sample combined with the flexibility to store any waveform allows for nearly unbeatable processing speed for waveforms.

1.4.2 Waveform function

The other method for Phase to Amplitude Conversion is the use of a *Waveform Function*. Typically Waveform Functions are used whenever the waveform to be generated can be computed with simple additive and multiplicative computations. Examples of waveforms that typically fit this are the sawtooth and triangle waveform types, as they contain linear sections. Another use case for Waveform Functions is in the case of the square and noise waveform types. In both cases, the resulting amplitude values can be "computed" via without involving any mathematics.

An advantage of using a Waveform Function to convert the Phase to Amplitude is that there is generally no need for interpolation, which results in more accurate amplitude value results. However, as there is typically some form of calculation involved, this method is generally less efficient than the Phase/Wavetable lookup method by a small margin. Each of the waveform functions used are described in subsection 3.6.1.

1.5 Additive & subtractive synthesis

After the Phase to Amplitude conversion, the resulting amplitude value is passed down for *Additive or Subtractive synthesis*. Additive and Subtractive synthesis are exactly what their names imply; they involve the use of additive/subtractive operations between waveforms to create a new waveform. Applied in a systematic method, distinct types of waveforms can be formed, as is shown in figure 1.5.

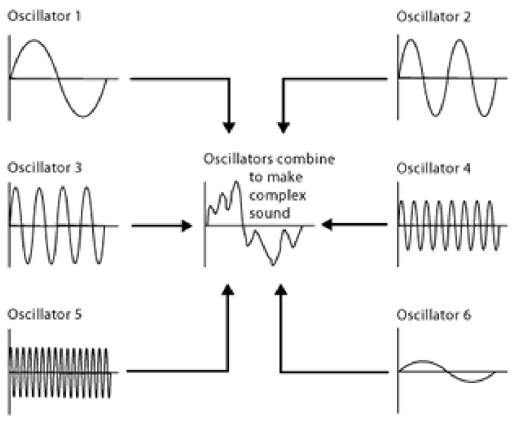


Figure 1.5: An example of additive synthesis of basic sine waves [22]

One method is to use the Fourier series approximation of a waveform. For example, the square wave can be generated via a weighted summation of harmonics of a sine wave. Starting with a fundamental sine wave at a base frequency, we add to this sine waveform each odd harmonic. A harmonic of a waveform is the waveform with a frequency at an integer multiple of the base frequency of the original waveform. The amplitude of each odd harmonic is multiplied by the reciprocal of the harmonic's frequency multiplier in relation to the base frequency as illustrated in figure 1.6.

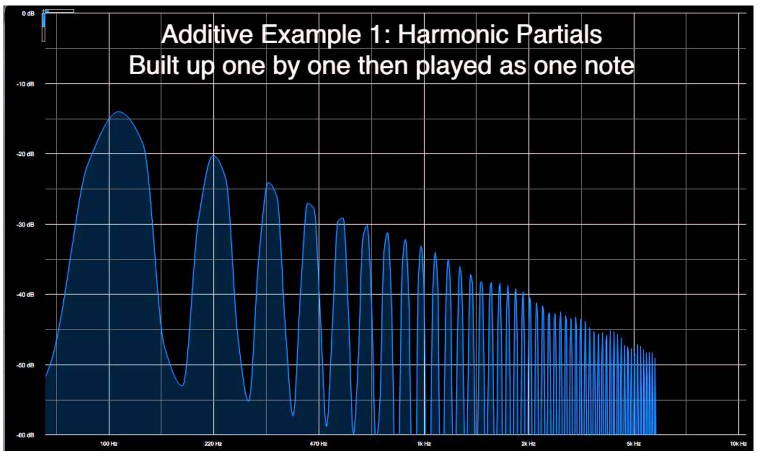


Figure 1.6: Additive synthesis of harmonic partials [20]

Another method for creating new waveforms is to use additive and subtractive synthesis to replicate sounds found in real life, a technique called physical modeling. For example, a piano's sound can be characterized by analyzing a spectral graph of its frequencies, from which is generated a set of base frequency multipliers and amplitude multipliers. An Additive/Subtractive synthesizer will then use this set of frequency and amplitude multiplier values to replicate the sound of the piano with high accuracy. This application of Additive and Subtractive synthesis for physical modeling is the technique used and popularized by the Yamaha synthesizers in the 1970s.

1.6 LFO

After the Additive and Subtractive synthesis processes generate a signal for the desired waveform, this is then passed on to the *Low-Frequency Oscillator Controlled Modulation* processes. Usually referred to as LFO's, these are modulation processes where a value produced by an oscillator modifies the value being modulated. This oscillator typically generates values using a basic sine wave. However, any waveform can be used, resulting in particularly interesting sound characteristics.

Of particular interest are the two most commonly used LFO's: the amplitude LFO and the frequency LFO. Both will modulate their respective parameters using the value from the low-frequency oscillator. An amplitude LFO will reproduce the effect of a tremolo, while the frequency LFO will, on the other hand, reproduce the effect of a vibrato. LFO's can be further controlled by processing the signal from the control oscillators in the same manner as a standard oscillator, although in general, this processing is typically limited to just an Envelope.

The LFO in the digital synthesis plugin is described in the User Documentation in subsection 2.3.4 and its implementation is discussed in the Developer Documentation in subsection 3.6.2.

1.7 Envelope

After the LFO processes the signal, it is then sent to the *Envelope*. An envelope is a generalized term describing a function or set of values that control the modulation value of a certain characteristic of a digital signal. Envelopes can be applied to many different types of modulation, including amplitude modulation and frequency modulation. An envelope can also be applied to the frequency and amplitude multiplier values used within additive and subtractive synthesis for further refined sounds. An example of a waveform with an applied envelope is shown in figure 1.7.

The use of the envelope module in the digital synthesis plugin is described in the User Documentation in subsection 2.3.6 and its implementation is discussed in the

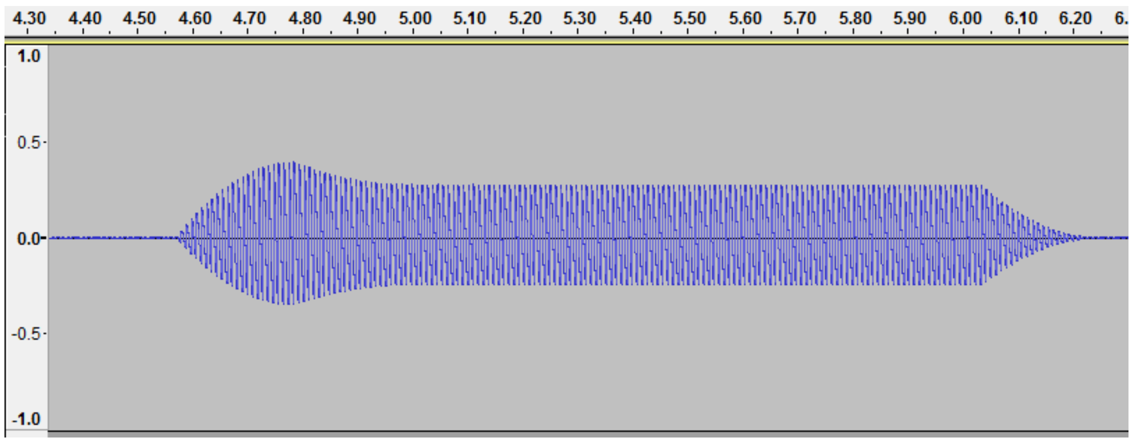
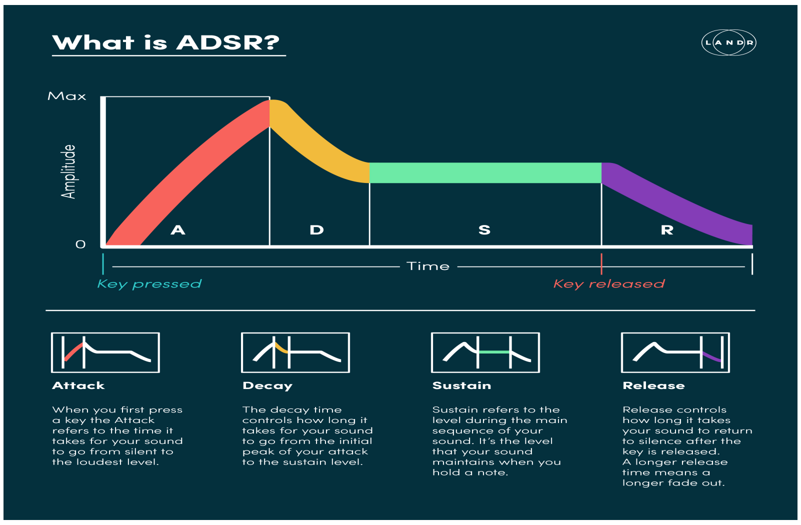


Figure 1.7: Sawtooth waveform with an applied amplitude modulation envelope [23]

Developer Documentation in subsection 3.6.3.

1.7.1 ADSR

The majority of digital synthesizers employ an Amplitude Modulation Envelope. These types of envelopes focus on modifying the amplitude of the sound over time, producing unique characteristics in sound, which can cause a synthesizer to produce the sound of a guitar vs. a violin. The simplest and most widely used envelope type is the *ADSR envelope*. ADSR stands for Attack-Decay-Sustain-Release, as these are the four parameters modified by the Envelope. A descriptive diagram is shown in figure 1.8.



When you first press a key the Attack refers to the time it takes for your sound to go from silent to the loudest level.

The decay time controls how long it takes for your sound to go from the initial peak of your attack to the sustain level.

Sustain refers to the level during the main sequence of your sound. It's the level that your sound maintains when you hold a note.

Release controls how long it takes your sound to return to silence after the key is released. A longer release time means a longer fade out.

Figure 1.8: ADSR envelope [21]

The Attack phase starts from the noteOn MIDI instruction and determines the amount of time required for the amplitude of the sound to transition from silence to its full value as determined by the MIDI velocity. After the Attack phase completes, the Decay phase begins, which acts in opposition to the Attack phase by determining the amount of time required for the amplitude to decrease down to its Sustain level. The Sustain level controls what percentage of the maximum amplitude the sound should be allowed to reach for the duration of the note. And finally, upon receiving a noteOff MIDI instruction, the Release phase begins, which determines the amount of time required for the sound amplitude to decrease back to silence.

These four values can drastically shape the character of a synthesizer's sound. For example, let us say we have produced a proper approximation of a string instrument sound via Additive and Subtractive synthesis. Using an Envelope with a long Attack phase, short Decay phase, high Sustain level, and long Release phase could produce a sound akin to a Cello played legato. On the other hand, using an Envelope with an almost instantaneous Attack phase, a minuscule Decay phase, low Sustain level, and a medium-length Release phase could produce a sound closer to that of an Acoustic Guitar. With proper control of the Envelope parameters, it is possible to replicate many different instruments and styles of playing.

Typically, the Envelope is considered the end of the Digital Signal Generation chain. The Digital Signal Generation chain will then accumulate the samples into an Audio Buffer that is sent to the Digital Signal Processing chain after it is filled.

In the Digital Signal Processing Chain, the order of the processes within the chain is entirely flexible and typically customizable by the end-user to best fit the sound they wish to generate. Additionally, the processes can be applied either in series or in parallel, depending upon the architecture of the synthesizer and the underlying framework. Setting up the Digital Signal Processing circuits in parallel holds an advantage for normalizing audio data to avoid artifacts and peaking, the term used for when the power or amplitude audio signal overloads the design capacity of the system. However, setting up the Digital Signal Processing circuits in series can avoid significant phase-shift issues that can occur from the asynchronous nature of parallel processes. The Digital Signal Processing effect types discussed in the following pages are presented in their serial order within the application.

Digital Signal Processing effects can be categorized into three primary archetypes: Frequency-Based, Time-Based, and Amplitude-Based. Digital signal processing effects within each archetype deal primarily with that domain of the digital signal;

however, these are frequently overlapping. For example, a simple reverberation effect is a Time-Based effect. However, more sophisticated reverberation effects will incorporate Frequency-Based effects to create a "shimmering" capability for the reverberation. A Multiband Compressor is also a common Amplitude-Based effect that will incorporate Frequency-Based filters to selectively modulate the amplitude from different parts of the frequency spectrum.

1.8 Filters

Multiple types of *Frequency Domain Filters* can be implemented. Each of these filters operates within the Frequency Domain, best visualized with the instantaneous point in the digital audio signal represented within the Frequency vs. Amplitude domains. In the Frequency Domain, modulation takes place to the amplitude of various frequencies along the frequency response spectrum.

The various Frequency Domain filter types include the Low, High, and Band Passes, the Low and High Shelves, and the Peaking filter. Each of these types has a different characteristic of how they affect the incoming audio signal. These types, in turn, changes the timbre, or characteristic quality, of the sound produced. The details will be discussed in the following parts. However, it is good to note that despite the multitude of variations that exist, these are all additive or subtractive combinations of the basic Pass filters.

Additionally, there are a variety of designs for audio filters. These designs, while initially made for analog systems, work exceptionally well for Digital Signal Processing as well. Each of these designs has differences in their properties that affect the character of the resulting audio signal filter. The most important properties are the Frequency-Domain Ripple factor, the Roll-Off, and finally, the Resonance.

The Frequency-Domain Ripple factor describes the amount of periodic variation in insertion loss with the frequency of a filter, an unavoidable side-effect of certain mathematical functions such as the notable Chebyshev polynomials which commonly used in polynomial interpolation. Some filter designs will utilize mathematical functions that can minimize or completely eliminate the Frequency-Domain Ripple as these ripples in frequencies causes undesirable audio artifacts in the resulting filtered audio signal.

The Roll-Off property of an audio filter design refers to the steepness of the transfer function with frequency, or in other words, the rate at which the amplitude changes with respect to frequency. Differing audio filter designs will have different capabilities for Roll-Off, with some mathematical functions such as the Elliptic functions being capable of accomplishing much steeper Roll-Off in contrast to other mathematical functions that can only support gentle slow Roll-Off. While a Roll-Off with a steeper value can be desirable for precise audio signal filters, a Roll-Off with a gentler value can be desirable for the more natural-sounding resulting audio signal.

The most common audio filter designs used are the Butterworth filter, the Chebyshev filter, and the Elliptic filter. Each of these audio filter designs has its advantages and trade-offs for each property. For example, the Butterworth filter design, also known as the maximally flat magnitude filter, holds the sole advantage over the other design types in that the Butterworth filter design has no Frequency Domain Ripple in the frequency response graph before the Roll-Off, however, the Butterworth filter design can only support slow and gentle Roll-Off with low Resonance values. By drastic contrast, the Elliptic filter design can support the steepest Roll-Off and the highest Resonance values. However, the Frequency Domain Ripple for Elliptic filters can quickly produce undesirable artifacts. Figure 1.9 shows typical frequency response graph of these audio filter design types.

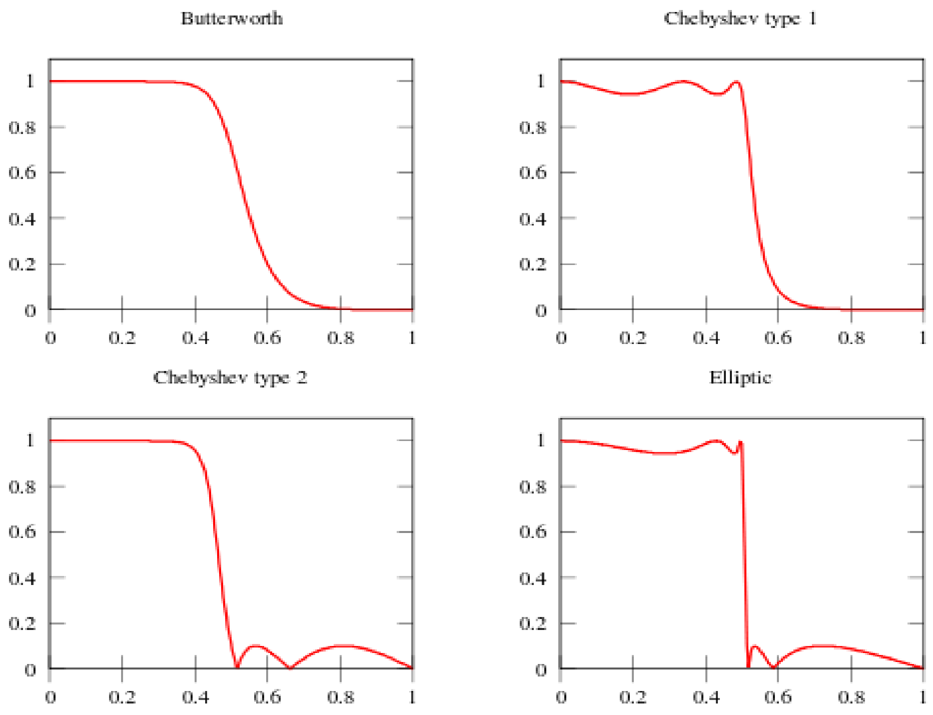


Figure 1.9: Butterworth, Chebyshev, Chebyshev2, and Elliptic filter frequency response graphs [24]

Typically a synthesizer will be designed around whichever filter design best fits the sound design goal and hardware limitations. More sophisticated and capable synthesizers may utilize complex Audio Filters, which combine multiple filter designs to produce a hybrid design that can mitigate the disadvantages of each design. Within this project, the Butterworth filter design is used exclusively for all Audio Filter types for its ease of implementation, lack of audio artifacts, and efficiency.

1.8.1 Low pass filter

A *Low Pass* is the most commonly used variant of the Pass type filters. A Low Pass Filter will attenuate the amplitude of signals with frequencies higher a set Cutoff frequency to $-\infty$ dB and pass signals with frequencies lower than the Cutoff frequency. The resulting sound will have a "darker" or "smoother" timbre. A Low Pass filter can also be instrumental in helping remove certain high frequencies that can be too harsh for some people. A frequency response graph of a low pass filter can be seen in figure 1.10.

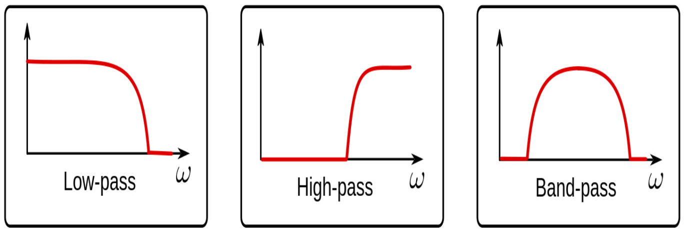


Figure 1.10: Filter types [25]

An everyday use of a Low Pass filter within a music workflow is to apply it to a synthesizer that is being utilized as a bass synth, with the cutoff frequency set to somewhere around 100hz. In this manner, the Low Pass filter will pass through the relevant bass frequencies while filtering out the higher frequencies that may interfere with the desired frequencies of the other instruments.

1.8.2 High pass filter

The *High Pass Filter* works in exact opposite to how the Low Pass filter operates. While a Low Pass filter will pass frequencies lower than the target cutoff frequency while filtering out those frequencies higher than the target cutoff frequency, the High Pass filter will do the opposite. A High Pass filter can help eliminate common low-frequency rumbles within recordings. A frequency response graph of a high pass filter can be seen in figure 1.10.

High Pass filters typically are used for vocal recordings where the low-end frequencies caused by vibrations to the microphone or recording setup produce undesirable sounds. Alternatively, a High Pass filter can be used to isolate specific high-frequency instruments, a triangle, for example, so that they can be processed separately.

1.8.3 Band pass filter

When a High Pass filter is combined with a Low Pass filter, a *Band Pass Filter* is created. A Band Pass filter will filter out frequencies within a certain range of the "cutoff" frequency while passing all frequencies outside of this range. The Band Pass filter is desirable for its ability to filter out a specific set of frequencies, and with a small enough range, even remove close to one single frequency. A frequency response graph of a band pass filter can be seen in figure 1.10.

This functionality of the Band Pass filter is commonly used to eliminate noise and clean up synth waveforms. In recordings, it is common that a particular machine will cause noise at a specific frequency throughout the duration of the recording. By determining the frequency of this noise and filtering it with a Band Pass filter, it is possible to remove noise effectively. The Band Pass filter can help with Additive Synthesis as well, as there are many instances when two waveforms will have constructive interference at a specific frequency that may cause distortion. The Band Pass filter will help remove these distortions.

By combining a variety of the three types of pass filters, an Equalizer is created, which can drastically alter the character of a waveform and is a staple of Digital Sound Processing platforms, including consumer-level products.

1.9 The MIDI standard

MIDI [7] is an acronym for *Musical Instrument Digital Interface*. Originating in 1983 from a panel of music industry representatives, it is a technical standard that describes a universal digital communications protocol for connecting any type of audio-related hardware and software. Analog hardware relies on converters that translate MIDI messages into analog control voltages.

A single MIDI link is capable of carrying up to sixteen channels of MIDI information. The routing of each channel can be completely independent of each other, allowing for flexible routing to and from different devices. Modern hardware and software will extend this functionality by using ports, with the current standard being 255 ports supported. Each port will support a full set of sixteen MIDI channels, making for a total of 4,080 available MIDI channels useable by the device.

Each MIDI channel is responsible for transmitting MIDI event messages. These event messages typically contain data that specify instructions for music such as a note's notation, pitch, velocity, panning, pitch bend, and tempo. However, in many cases, devices will send and receive proprietary MIDI event messages that can contain instructions for anything, ranging from automating and triggering other devices to controlling performance lights and pyrotechnics. One notable example is the use of MIDI to connect the music playing to a crowd to a fireworks display in a manner that synchronizes well and is automated.

1.9.1 MIDI event messages

MIDI event messages consist of 8-bit byte-words transmitted in series at a rate of 31.25 kilobits per second, a rate chosen for being an exact divisor of 1 MHz. Each byte-word starts with a bit indicating if the word is a status byte or a data byte. The next seven bits encode the relevant information or instruction. On top of the 8 bits forming the byte-word, a start bit and a stop bit are used to frame the byte-word, summing up to a total of ten bits per byte-word in transmission [8].

Each MIDI event message contains an instruction for which the receiving device may choose to act upon. The two main categories of MIDI event messages are Channel event messages and System event messages. A device can choose to act upon or ignore any or all MIDI event messages received.

Channel event messages contain instructions that usually are relevant to the audio to be produced. A `noteOn` message will indicate to the device that a note has started. Similarly, the `noteOff` message indicates that the note has ended. On a keyboard, these two MIDI event messages would correspond to the pressing of a key and the release of the key. Channel event messages can also convey pitch information in the form of MIDI notation. MIDI notation assigns the integers from 0 to 127 to all possible notes within the music. This range of integers corresponds to the range from $C1$ up to $G9$ according to the Helmholtz pitch notation standard, covering a range of frequencies from 8.175798916 Hz to 12,543.85395 Hz. This range of possible notes and frequencies extends well beyond the range of the majority of acoustic instruments. Another important piece of information that Channel event messages can convey is the `velocity` value of a note. The `velocity` value corresponds to how forcefully a note was played, corresponding to the *piano* and *forte* notation instructions on traditional sheet music. The `velocity` value typically will control the loudness of the sound generated, but may also control other parameters such as resonance and vibrato. For digital synths, the `velocity` value of a note can be used to control different modulations. The Channel event messages can also contain tempo and time signature instructions. These correspond to their musical notation equivalents and combine with the beat value of a note to determine the time value of the note. The beat value is conveyed in the Channel event messages via the `duration` value. Each `duration` unit within MIDI notation corresponds to $1/24^{th}$ of a beat value. There is also a large variety of other Channel event messages; however, the essential Channel event message types have been discussed.

System event messages are used to promote the flexibility and longevity of the MIDI communication standard by allowing manufacturers to create proprietary event messages. These System event messages allow for devices to communicate instructions typically corresponding to other software or hardware control parameters. For hardware and analog systems, a converter will translate the MIDI System event message to control voltage values. There are no universal standards for System event messages. However, with modern MIDI software and hardware systems, there are commonly used standards established by widely used software.

1.9.2 MIDI channels and ports

Each MIDI link between devices is capable of carrying up to sixteen channels of MIDI information. The MIDI channels are identified with a binary encoding with values from 0 to 15. However, to the End-User, these are presented as the numbers 1 to 16. MIDI ports extend a device's MIDI link capacity. These MIDI ports are identified with hexadecimal encoding with values from 0 to 255. MIDI-compatible hardware and software support 16 MIDI channels of information, but they may choose to listen or ignore any number of these MIDI channels. On the other hand, the implementation of MIDI ports is dependent upon the constraints of the software or hardware and is more commonly found on modern devices.

The application of this thesis project supports one MIDI link of sixteen channels, with all MIDI channels being handled at the same time. System event messages are ignored, as the application is not device or platform-specific, nor does it use proprietary interfaces. However, most MIDI event messages are handled by the application.

More details about MIDI specification can be found at the MIDI specification website [8].

1.10 Background conclusion

The background info presented in this chapter does not comprehensively cover the entirety of digital signal processing knowledge regarding the topics discussed, however the prerequisite minimum information necessary for developing the application is presented.

Chapter 2

User Documentation

In this chapter, the application will be explained from an End-User perspective with explanations on how to use each part of the application within their workflow to achieve the desired sound. This section will rely heavily on technical and musical terminology discussed in the previous chapter 1, and as so, will focus primarily on describing in detail the operation and effects of the application.

This chapter will assume that the user will be familiarized with the use of DAWs and has one installed. For demonstration purposes, *FL Studio* will be the DAW shown. More information can be found on the Image-Line website [9], and details on installation and usage of FL Studio can be found on the user manual website [11]. Guided tutorials can also be found on the tutorial site [10].

2.1 Installation

The installation of the software is relatively straightforward and simple. This section will walk the End-User through the Build, Installation, and Setup process on different platforms. For those who prefer to download the precompiled binaries for the VSTi plugin, please read the following subsection. However, for those who prefer to build the application from the source code, please refer to section 3.

2.1.1 Installing the VSTi

The latest release builds of the VST3 plugin can be found at the Github page[4]. Once downloaded, the installation of the plugin within the preferred Digital Audio Workstation will proceed as specified by the specific Digital Audio Workstation.

Typically, the process consists of placing a copy of the VST3 plugin binary within the designated system folder for VST3 plugins and registering the new plugin within the Digital Audio Workstation. The Digital Audio Workstation must support ASIO drivers, MIDI input, and VSTi 3.0 plugins.

For the purposes of demonstration, the *FL Studio* 20 Digital Audio Workstation running on the *Windows* 10 operating system will be shown and referenced in the instructions.

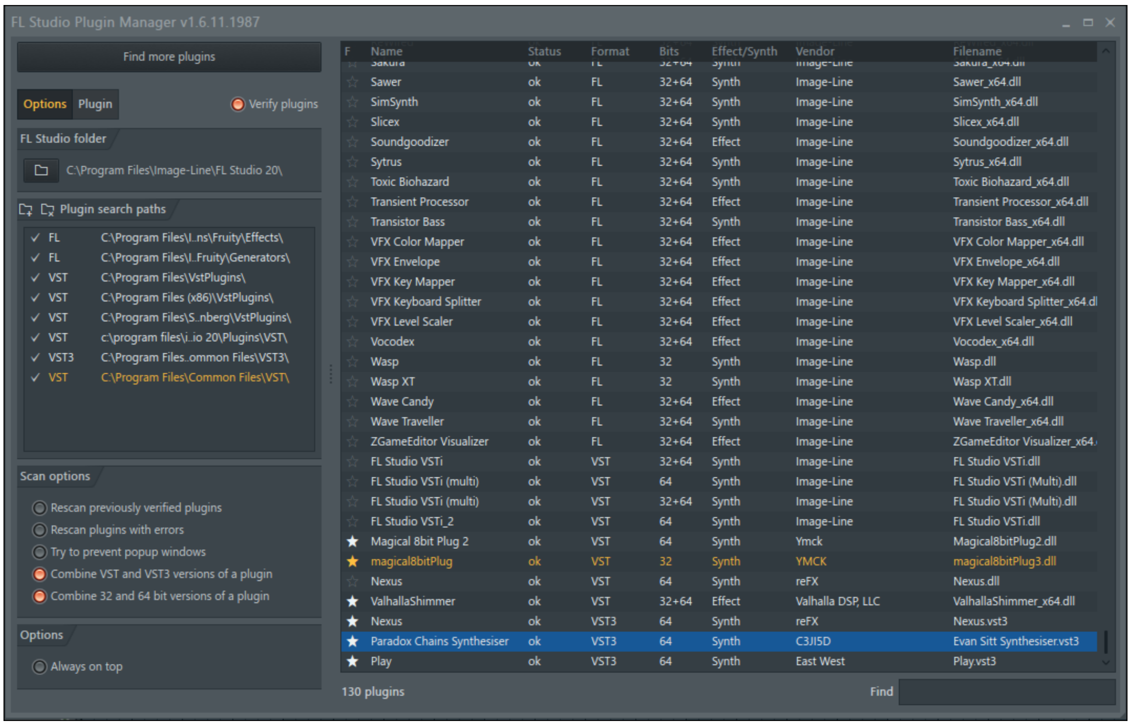
Figure 2.1: FL Studio logo [9]

2.1.2 Registering the VST3 plugin

The first step to installing the VST3 plugin is to copy the `.vst3` file into the appropriate directory set in the *Plugin Manager > Plugin search paths* field. By default, the directory set for 64-bit plugins on 64 bit Windows 10 is typically found in the `'../Program Files/Common Files/VST3'` directory.

Afterward, the user should open the *Plugin Manager* via the *Options* by going to *File settings* and then clicking on the *Manage plugins* button. Figure 2.2 shows an example of what this window should look like.

After ensuring that the *Verify plugins* radio button is turned on, the next step is to use the *Find Plugins* button to scan the preregistered VST plugin folders for the new VST3 plugin. The window will then show the new plugin, named *Paradox - Chains Synthesiser*, within the plugin list on the right panel. By selecting the star button next to the plugin name, the plugin will be added to the favorites to ensure ease of finding the plugin in the future. This last step completes the registration of the VST3 plugin within the FL Studio 20 Digital Audio Workstation. Figure 2.3 shows the menu from where the plugin can be added into the project.



FL Studio Plugin Manager v1.6.11.1987

Find more plugins

Options | Plugin | Verify plugins

FL Studio folder

C:\Program Files\Image-Line\FL Studio 20\

Plugin search paths

✓	FL	C:\Program Files\I..ns\Fruity\Effects\
✓	FL	C:\Program Files\I..Fruity\Generators\
✓	VST	C:\Program Files\VstPlugins\
✓	VST	C:\Program Files (x86)\VstPlugins\
✓	VST	C:\Program Files\S..nberg\VstPlugins\
✓	VST	c:\program files\i..io 20\Plugins\VST\
✓	VST3	C:\Program Files..ommon Files\VST3\
✓	VST	C:\Program Files\Common Files\VST\

Scan options

- Rescan previously verified plugins
- Rescan plugins with errors
- Try to prevent popup windows
- [✓ Combine VST and VST3 versions of a plugin]
- [✓ Combine 32 and 64 bit versions of a plugin]

Options

- Always on top

F	Name	Status	Format	Bits	Effect/Synth	Vendor	Filename
☆	Sakura	ok	FL	32+64	Synth	Image-Line	Sakura_x64.dll
☆	Sawer	ok	FL	32+64	Synth	Image-Line	Sawer_x64.dll
☆	SimSynth	ok	FL	32+64	Synth	Image-Line	SimSynth_x64.dll
☆	Slicex	ok	FL	32+64	Synth	Image-Line	Slicex_x64.dll
☆	Soundgoodizer	ok	FL	32+64	Effect	Image-Line	Soundgoodizer_x64.dll
☆	Sytrus	ok	FL	32+64	Synth	Image-Line	Sytrus_x64.dll
☆	Toxic Biohazard	ok	FL	32+64	Synth	Image-Line	Toxic Biohazard_x64.dll
☆	Transient Processor	ok	FL	32+64	Effect	Image-Line	Transient Processor_x64.dll
☆	Transistor Bass	ok	FL	32+64	Synth	Image-Line	Transistor Bass_x64.dll
☆	VFX Color Mapper	ok	FL	32+64	Effect	Image-Line	VFX Color Mapper_x64.dll
☆	VFX Envelope	ok	FL	32+64	Effect	Image-Line	VFX Envelope_x64.dll
☆	VFX Key Mapper	ok	FL	32+64	Effect	Image-Line	VFX Key Mapper_x64.dll
☆	VFX Keyboard Splitter	ok	FL	32+64	Effect	Image-Line	VFX Keyboard Splitter_x64.dl
☆	VFX Level Scaler	ok	FL	32+64	Effect	Image-Line	VFX Level Scaler_x64.dll
☆	Vocodex	ok	FL	32+64	Effect	Image-Line	Vocodex_x64.dll
☆	Wasp	ok	FL	32	Synth	Image-Line	Wasp.dll
☆	Wasp XT	ok	FL	32	Synth	Image-Line	Wasp XT.dll
☆	Wave Candy	ok	FL	32+64	Effect	Image-Line	Wave Candy_x64.dll
☆	Wave Traveller	ok	FL	32+64	Synth	Image-Line	Wave Traveller_x64.dll
☆	ZGameEditor Visualizer	ok	FL	32+64	Effect	Image-Line	ZGameEditor Visualizer_x64.
☆	FL Studio VSTi	ok	VST	32+64	Synth	Image-Line	FL Studio VSTi.dll
☆	FL Studio VSTi (multi)	ok	VST	64	Synth	Image-Line	FL Studio VSTi (Multi).dll
☆	FL Studio VSTi (multi)	ok	VST	32+64	Synth	Image-Line	FL Studio VSTi (Multi).dll
☆	FL Studio VSTi_2	ok	VST	64	Synth	Image-Line	FL Studio VSTi.dll
★	Magical 8bit Plug 2	ok	VST	64	Synth	Ymck	Magical8bitPlug2.dll
★	magical8bitPlug	ok	VST	32	Synth	YMCK	magical8bitPlug3.dll
☆	Nexus	ok	VST	64	Synth	reFX	Nexus.dll
★	ValhallaShimmer	ok	VST	32+64	Effect	Valhalla DSP, LLC	ValhallaShimmer_x64.dll
★	Nexus	ok	VST3	64	Synth	reFX	Nexus.vst3
★	Paradox Chains Synthesiser	ok	VST3	64	Synth	C3JI5D	Evan Sitt Synthesiser.vst3
★	Play	ok	VST3	64	Synth	East West	Play.vst3

130 plugins

Find

Figure 2.2: Module registration screen

2.2 Initialization and workflow integration

While the previous steps stated in the *Installation* section 2.1 will only need to be completed once, the steps in this section and the following *Workflow Integration* section 2.2 will be the steps necessary to be taken for every new project established.

To add an instance of the VST3 plugin to a project, simply go to the *Add* dropdown menu in the top left, and then select the *Paradox Chains Synthesiser* option. This action will automatically add an instance of the VST3 plugin to the *Channel rack*. The *Channel rack* can be accessed via *'View > Channel rack'* or via the *F*6 shortcut key. By clicking on the VST3 plugin within the *Channel rack*, the user can access the graphical user interface of the synthesizer.

Each outlined portion of the graphical user interface of the VST3 plugin contains the controls which correspond to a process within the Digital Signal Processing chain. The specifics of each module will be discussed in the *Module Details* section 2.3. Controls on the main user interface can be set during a project or modified in real-time during a performance as the situation demands. The Digital Synthesis Processes will automatically read in the new parameters and adjust the sound accordingly.

The full list of detailed controls, with labels and fine controls, can be accessed

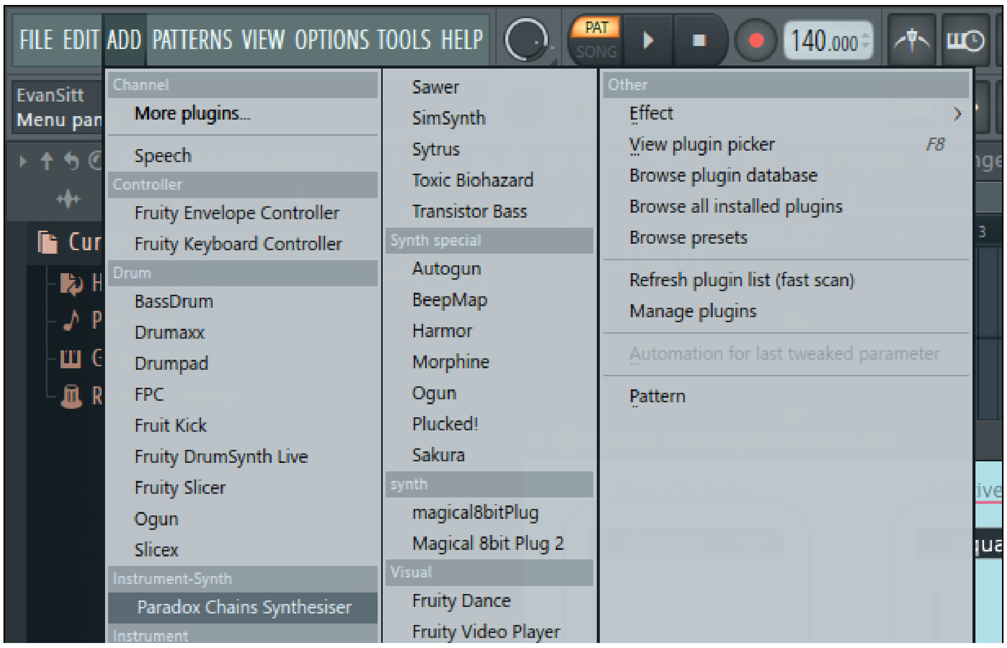


Figure 2.3: Menu for adding plugins

by advanced users via the *Browser* panel on the left side of the screen. By using the center *Current project* tab, opening up the *Generators* listing, and then opening up the *Paradox Chains Synthesiser* listing, the advanced user will have access to all possible MIDI values for the VST3 plugin. A section of what the *Browser* panel looks like is shown in figure 2.4.

Clicking on any one of them will bring up a detailed rotary slider that will also display a precise value to pass to the VST3 plugin. Note, the value displayed by the Digital Audio Workstation is from a range between 0.0 and 100.0, corresponding to the percentage of the range of the MIDI parameter. However, as detailed within the *Module Details* section 2.3, the actual value range of the MIDI parameters will be different depending upon the parameter. This modifying rotary slider is shown by figure 2.5.

Right-clicking on any of the detailed MIDI parameters will bring up advanced user options. The "*Init song with this position*" option will allow the advanced user to set the current value as an initialization value. These detailed MIDI options are shown in figure 2.6. The "*Edit events*" option will allow the advanced user to enter a detailed MIDI event graphical editor window. The "*Create automation - clip*" option will allow the advanced user to create tracks to automate the MIDI parameter. Lastly, but certainly not least, the "*Link to controller*" option will allow

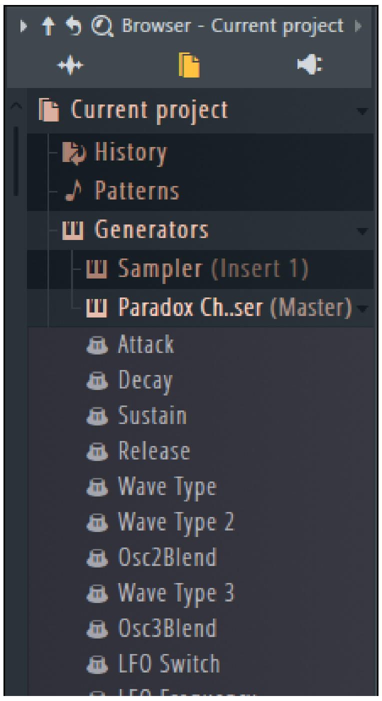


Figure 2.4: Browser panel

the advanced user to link the MIDI parameter, or even the entire VST3 plugin, to either a software control panel, such as FL Studio's *Patcher*, or even a hardware MIDI device. The "*Link to controller*" option is shown as an example in figure 2.7.

The final step to integrating the *Paradox Chains Synthesiser* into a workflow is to route the digital audio output to the mixer. By opening up the VST3 plugin's graphical user interface, there is an option to click on the Settings button to bring up the detailed settings window. This is shown in figure 2.8. There is a multitude of options, such as modifying the MIDI input and output ports; however, the relevant option is to change the Track Assignment in the top right corner of the window. By changing this value, the VST3 plugin will be assigned to the corresponding mixer track for further processing and routing to the final output. Figure 2.9 illustrates these options.

2.3 Module details

In this section, we will discuss each module of the digital synthesis plugin. The plugin user interface is shown in figure 2.10, and each of the following subsections will reference their respective module from left to right. The name of the module is

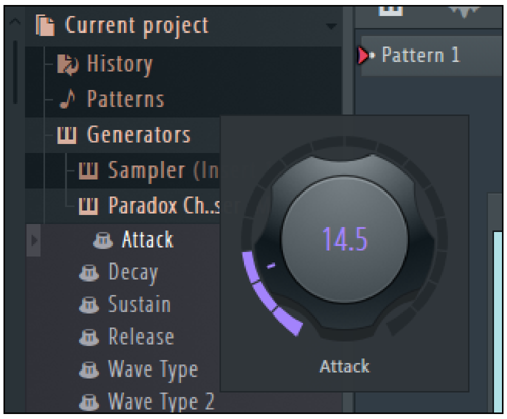


Figure 2.5: Modifying MIDI details

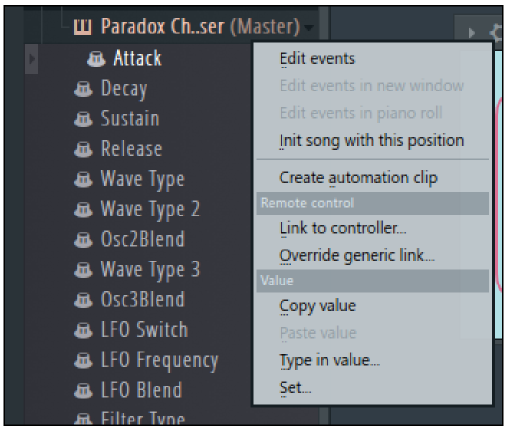


Figure 2.6: Detailed MIDI options

also labeled on top of each module in the user interface.

2.3.1 Main oscillator

The *Main Oscillator* module controls the primary oscillator. This oscillator is the one which will play at full volume. The dropdown menu allows for the user to control which waveform should be generated by the primary oscillator.

2.3.2 Additive oscillator

The *Additive Oscillator* module controls the secondary oscillator. This oscillator will generate a waveform which will be combined with the primary waveform via additive synthesis. The dropdown menu allows for the user to control which waveform should be generated by the secondary oscillator. The slider controls the mix amount of the

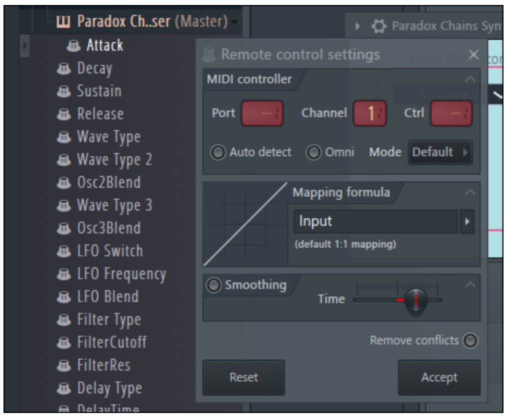


Figure 2.7: Link to controller menu

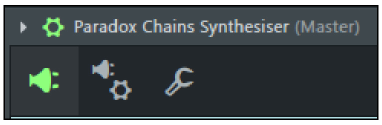


Figure 2.8: Settings and plugin options buttons

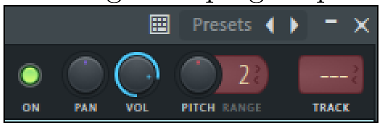


Figure 2.9: Plugin settings, with track setting on the right

secondary waveform to be used for additive synthesis.

2.3.3 Subtractive oscillator

The *Subtractive Oscillator* module controls the tertiary oscillator. This oscillator will generate a waveform which will be combined with the primary waveform via subtractive synthesis. The dropdown menu allows for the user to control which waveform should be generated by the tertiary oscillator. The slider controls the mix amount of the tertiary waveform to be used for subtractive synthesis.

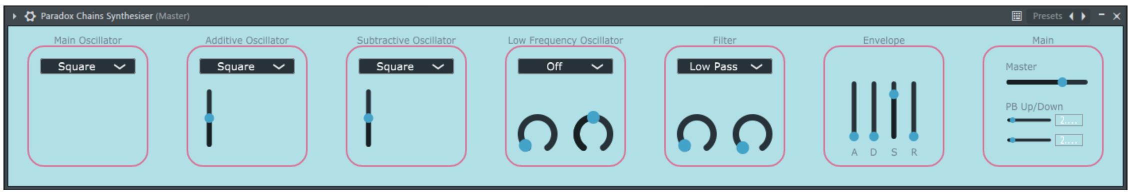


Figure 2.10: Synthesizer user interface

2.3.4 Low frequency oscillator

The *Low Frequency Oscillator* module controls the oscillator which generates the waveform controlling the amplitude modulation. The dropdown menu allows for the user to control whether or not the LFO amplitude modulation should be turned *On* or *Off*. The left radial dial controls the frequency of the waveform generated by the oscillator. The right radial dial controls the amplitude of the waveform generated by the oscillator.

2.3.5 Filter

The *Filter* module controls the filter module which performs post processing filtering on the resulting signal from the three oscillators. The dropdown menu allows for the user to control which filter type is applied. The options available are *Low Pass*, *High Pass*, and *Band Pass*, each corresponding to their respective filter types. The left radial dial controls the cutoff frequency of the applied filter. The right radial dial controls the resonance of the applied filter.

2.3.6 Envelope

The *Envelope* module controls the envelope controlled amplitude modulation. This is applied to the generated signal from the combined output of the three oscillators. The four sliders are labeled with *A*, *D*, *S*, and *R*, corresponding to the *Attack*, *Decay*, *Sustain*, and *Release* values of the envelope.

2.3.7 Main

The *Main* module controls the Master volume of the application along with pitch bend. The slider labeled with *Master* controls the master volume, which is the final post processing step in the signal chain. The two sliders below the *PBUp/Down* label control the pitch bend up and pitch bend down respectively.

2.4 User results

After setting up the digital synthesis plugin and routing it properly within the workflow of the digital audio workstation, MIDI note input from either a MIDI controller such as a keyboard or from MIDI notation should produce the expected sound from the digital synthesis plugin. This audio should be routed through the mixer panel which should allow for additional mixer signal post processing modules to be added onto the signal chain.

Chapter 3

Developer documentation

This chapter will cover the basics and the process for developing the digital synthesis plugin. The developer is presumed to have the prerequisite knowledge of C++ and familiarity with Visual Studio. Visual Studio can be downloaded from the main website [5], and usage information can be found through the documentation [6].

3.1 Prerequisites

Before writing the code, the developer must have access to a few necessary tools.

- The Project Source Code, available from Github [4].
- The JUCE C++ Application Framework [1]
- A development environment of your choice. JUCE natively supports Visual Studio, XCode, and Linux Make.
- A Digital Audio Workstation of your choice. It must support VST3 plugins.

3.1.1 Building the software from source code

The stable source code of this application can be downloaded from the `master` branch of the public repository found at Github: https://github.com/ParadoxChains/-C3JI5D-Evan-Sitt-BSc-Thesis [4]. The source code includes all of the $C++$ header and CPP files that are necessary for the application. However, the repository does not include the JUCE framework that is required for building the project.

3.1.2 JUCE framework

The JUCE framework is a partially open-source, cross-platform C++ application framework. It is particularly notable for its robust support of the VST3 SDK providing developers with an easy toolset for interacting with VST3 applications. Additionally, the JUCE framework provides graphical user interface functionality, supports a large variety of integrated development environments and compilers, and is designed to compile and run compatibly on the Windows, Mac OS X, and Linux platforms. More information regarding JUCE can be found on the homepage [1].

The JUCE framework is required for the project as it provides the base framework for interacting with the VST3 SDK. The JUCE framework can be found on the Roli JUCE website at [1]. The project for this application will work with both paid, free, and GPL licenses for JUCE. The Projucer application, a necessary extension of the JUCE framework, can be downloaded along with the framework. Further details on setting up the JUCE framework and Projucer application can be found on the JUCE website [1]. Additional information regarding JUCE development basics can be found on various tutorial sites [3].

3.1.3 VST3 standard

The JUCE framework's main strength is the robust support for working with the VST3 SDK. This acronym stands for the Virtual Studio Technology audio plugin interface developed by Steinberg Media Technologies.

Figure 3.1: Virtual Studio Technology logo [26]

VST Plugins are designed to work in conjunction with a Digital Audio Workstation, or with a similar standalone wrapper application providing the same functionality. VST Plugins are generally categorized into two categories: Generators and Effects. A subcategory of the Effects category exists for the sole purpose of visualization and data representation called Visualizers. VST Plugins typically have an integrated graphical user interface for ease of use, presenting the End-User with controls that mimic their physical equivalents on audio hardware. However, many VST plugins have controls for advanced users that do not have a graphical user interface and rely on the host application or Digital Audio Workstation for the user interface.

The Generator category of VST plugins typically consists of digital synthesizers. These are software simulation emulations of audio synthesizer hardware, typically incorporating similar signal flows and processes, however, usually, optimizations are made in consideration to the nature of digital synthesis.

The Effects category of VST plugins differs from the Generator category in that they do not generate new audio signals. Instead, they modify an existing audio signal. As such, these VST plugins typically do not take MIDI input or produce MIDI output, but only deal with audio signal input and output. Similar to the Generator category of VST plugins, Effect VST plugins are software simulation emulations of their respective audio synthesizer hardware equivalents.

Many modern VST3 plugins are a hybrid of the Generator and Effect types. These VST3 plugins will typically receive MIDI input, generate new audio signals from the MIDI instructions. Afterward, the VST3 plugin would process those audio signals in tandem with signals received from the audio input before passing the finalized audio signal to the audio output. The application developed for this thesis falls within the Generator/Effects hybrid category of VST plugins and provides a simplified graphical user interface for the entry-level user while making the advanced controls accessible from the Digital Audio Workstation.

The VST standard specifies a standard for communication between Digital Audio Workstations and the VST Plugins. The standard includes communication of MIDI input and output, audio signal input and output, and parameter value input and output. The VST Plugin will specify to the Digital Audio Workstation which inputs are available. Based upon the inputs received from the Digital Audio Workstation, the VST Plugin will process this information and then send various output messages back to the Digital Audio Workstation. These output messages can go directly to the Digital Audio Workstation or can be directly passed on to other VST Plugins, creating a signal chain. As such, *chaining* is a useful technique for connecting VSTs to form deeply layered and sophisticated sounds.

3.2 Process flow

The central object which handles the flow of data is the `AudioProcessor` found within the `PluginProcessor` header and CPP files. The `AudioProcessor` is responsible for directing MIDI input and Audio output throughout the different objects. Additionally, the `AudioProcessor` is responsible for handling the MIDI buffer and Audio buffers. The `AudioProcessor` will contain instances of the other classes and assists them in interacting with each other [2].

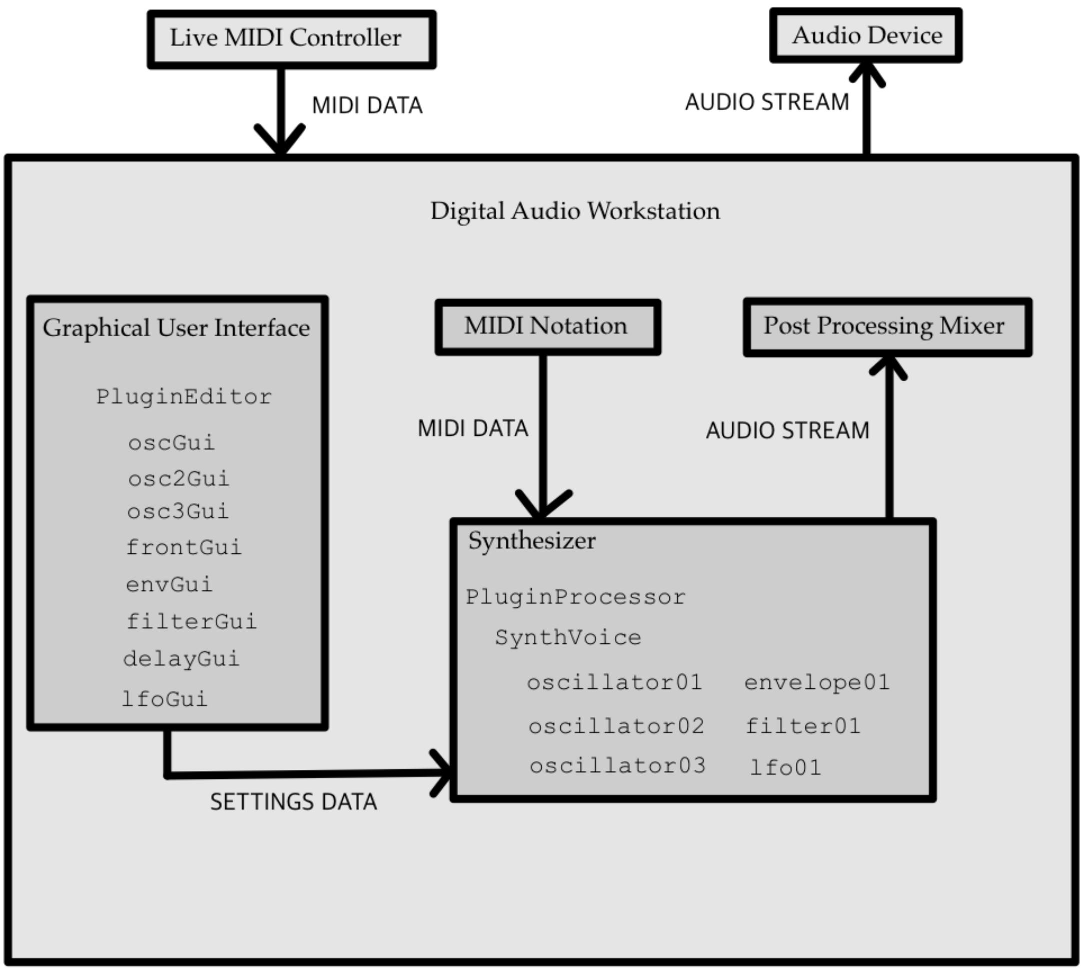


Figure 3.2: Application data flow

The front-end of the `AudioProcessor` that will be used by the End-User is handled by the `AudioProcessorEditor`. This `AudioProcessorEditor` object contains each of the individual graphical user interface objects and controls the overall appearance of the VST3 plugin. These individual graphical user interface objects are `processor` objects that each will contain the individual graphical user interface elements to assist the End-User in changing the state values of the VST3 plugin, such as sliders and rotary dials.

The `AudioProcessor` also contains an array of `SynthesiserVoice`. Each of these objects is a Synthesiser object that will perform the computation of values that will be handed back to the `AudioProcessor` for sending out to the audio buffer. The `AudioProcessor` is responsible for handling the MIDI buffer and sending those sets of instructions to the Synthesiser objects for processing. These are illustrated in figure **??**.

This process flow is illustrated briefly in figure 3.2.

3.3 Development flow

While development can follow the flow of the process, it is not intuitive to develop in this manner. To develop the `AudioProcessor` first would require a complete understanding of which control and state values would be required for each synthesizer component.

First, the developer must implement the functions for handling the MIDI buffer first and the corresponding synthesizer functions. The audio buffer functions must also be developed afterward to handle the output correctly.

Then at this point, the synthesizer components can be created. After this, developing the `AudioProcessor` to manage the control and state values is much more straightforward. From here, it is advised to develop the graphical user interface for the new synthesizer component, which will connect to the `AudioProcessor`, and then finally adding it to the `AudioProcessorEditor`.

Special care must be taken to ensure that all functions are thread-safe, as the JUCE framework makes full use of VST3's multithreading support for performance, especially when handling high polyphony counts [2].

The following section will walk through the development flow as described and provide detailed instructions on each portion.

3.4 Initializing the synthesiser

The first step is to set up the `Synthesiser` controlled by the `AudioProcessor`. It is sufficient to declare the `Synthesiser` within the private members and allow it to use its default constructor. Within the constructor of the `AudioProcessor`, we then create a loop that will add new instances of the `SynthesiserVoice` class and new instances of the `SynthesiserSound` class to the `Synthesiser`. The following is the code snippet that performs this action.

```
mySynth.clearVoices();

for (int i = 0; i < 5; i++)
{
    mySynth.addVoice(new SynthVoice());
}
mySynth.clearSounds();
mySynth.addSound(new SynthSound());
```

Listing 3.1: Instantiating Voices

The `clearVoices` and `clearSounds` methods ensure that we are starting with a fresh `Synthesiser`.

3.5 Handling the MIDI and audio buffers

There are two parts to the handling of MIDI input and audio output. The `Audio-Processor` must implement a `processBlock` function. This function is called by the framework at every tick of the program, and is passed the references to the `AudioSampleBuffer` and the `MidiBuffer`. The following is the declaration required for this function:

```
void JuceSynthFrameworkAudioProcessor::processBlock (AudioSampleBuffer& buffer, ↩
    MidiBuffer& midiMessages)
```

Listing 3.2: `processBlock` Declaration

Within the `processBlock` function, the developer may choose to implement any sort of system-wide functionality that must occur in real-time. One of these functionalities is the retrieval of control and state values from the `AudioProcessor-ValueTreeState`, which will be discussed in detail in subsection 3.6.4.

The first step is to ensure that a call is made to the `renderNextBlock` method of each `SynthesiserVoice` that was added to the `Synthesiser`. These calls are done within a loop that iterates through each `SynthesiserVoice`. The function call is as follows:

```
mySynth.renderNextBlock(buffer, midiMessages, 0, buffer.getNumSamples());
```

Listing 3.3: `renderNextBlock` function call

Within the `renderNextBlock` function call, the audio buffer and MIDI buffer are both passed in by reference, followed by the index of the starting sample and the index of the last sample.

Within the `SynthesiserVoice` derived class we must implement an override on the `renderNextBlock` method. The function declaration is as follows:

```
void renderNextBlock(AudioBuffer<float>& outputBuffer, int startSample, int ↩
    numSamples) override
```

Listing 3.4: `renderNextBlock()` override

This method must accomplish the task of rendering and adding the audio output sample values to the provided `AudioBuffer`. The code within this consists of two nested loops, one which iterates through each sample of the `AudioBuffer`, and the other which iterates through each audio channel that needs a sample. Within the loops we call an `AudioBuffer` method, `addSample`, to add the new sample to the `AudioBuffer`. The code snippet is as follows:

```
for (int sample = 0; sample < numSamples; ++sample)
{
    for (int channel = 0; channel < outputBuffer.getNumChannels(); ++channel)
    {
        outputBuffer.addSample(channel, startSample, giveSample());
    }
    ++startSample;
}
```

Listing 3.5: `addSample()`

The first argument is the channel index, the second argument is the sample index, and the last argument is the value to be written. To keep the code readable and maintainable, the value to be written is provided by the function `giveSample`.

In addition to the `renderNextBlock` method, additional methods are required in order to process the relevant MIDI event messages passed in from the MIDI buffer. The `startNote` and `stopNote` methods of the `SynthesiserVoice` class must be overridden by the developer. The `startNote` method is called whenever the *noteOn* MIDI event is encountered within the MIDI buffer while the `stopNote` method is called whenever the *noteOff* MIDI event is encountered in the MIDI buffer.

The `startNote` method must perform any function required whenever a new note is started. The `trigger` value of the `envelope` component, described in subsection 3.6.3, is set to the value of 1 to indicate the start of the *Attack* phase. The position of the pitch wheel, a general MIDI hardware component, is then passed to the function `setPitchBend` to set the `pitchBend` value. The `midiNote-Number` is then converted to the corresponding frequency value in Hz and stored as the `frequency` value which will be used by the `oscillator` components, as described in subsection 3.6.1. Finally, the `velocity` value from the MIDI event instruction is stored into the `level` value. The following is the code snippet for the `startNote` method:

```
noteNumber = midiNoteNumber;
envelope01.trigger = 1;
setPitchBend(currentPitchWheelPosition);
frequency = noteHz(noteNumber, pitchBendCents());
level = velocity;
```

Listing 3.6: `startNote()`

The `stopNote` method must perform all functions required whenever the current note is ended. The `trigger` value of the `envelope` component is set to 0 to indicate the start of the *Release* phase. The `allowTailOff` value is set to `true` to indicate to the *JUCE* framework that the release tail should be allowed to persist in the audio output buffer. Finally, after the `velocity` value is equivalent to 0, then the `clearCurrentNote` method is called which ensures that the current note is cleared from the instructions received from the MIDI buffer. The following code snippet is the instructions involved within the `stopNote` method:

```
envelope01.trigger = 0;
allowTailOff = true;
if (velocity == 0)
    clearCurrentNote();
```

Listing 3.7: `stopNote()`

3.6 The Digital synthesis Components

The next parts for the developer to build are the individual Digital Synthesis components. The usage and purpose of each component have been discussed in the *Module Details* section 2.3, and this section will focus on the implementation and technical details.

3.6.1 Waveform oscillators

Designing an oscillator consists of two primary parts: designing the *Phase Accumulator* as described in 1.3 and designing the $Phase\!-\!to\!-\!Amplitude\ Converter$ as described in section 1.4.

The *Phase Accumulator* is accomplished by declaring a `phase` variable for the `oscillator` class. This value will be used by each of the $Phase\!-\!to\!-\!Amplitude$ - *Converter* functions. Since each $Phase - to - Amplitude\ Converter$ function uses the same `phase` value, it is ensured that the transition between different waveforms can be done with no phasing issues and with relative smoothness.

The `resetPhase` methods allow for the `oscillator` to reset the phase to 0 or to set it to a specified value. The `iteratePhase` method is responsible for iterating the value of `phase` by the formula: $\dfrac{\texttt{waveTableSize}}{\left(\dfrac{\texttt{sampleRate}}{\texttt{frequency}}\right)}$

The `waveTableSize` and `sampleRate` are constants set by the constructor of the `oscillator`. The `waveTableSize` represents the how many precalculated values are stored for a single waveform cycle within a Waveform Lookup Table as described in subsection 1.4.1. The `sampleRate` represents the number of samples to be rendered per second for the audio output. The `frequency` is a value that will be provided to the oscillator when it is called by the `SynthesiserVoice` class. By iterating the `phase` value by the calculated value from the `waveTableSize`, `sampleRate`, `frequency`, the `phase` can not only accurately represent the phase but also represent the wavetable lookup index.

With the *Phase Accumulator* completed, the next step is to create the $Phase - to - Amplitude\ Converter$. In this project, both $Waveform\ Lookup\ Tables$ and $Waveform\ Functions$ are used. Each of the $Phase - to - Amplitude\ Converter$ functions follow the same conventions in regards to input and output. Each of the functions takes `frequency` as an argument, which acts as the $Tuning\ Word$, and returns a `double` representing the resulting amplitude from the conversion. These will be explained in the following paragraphs.

The `sine` function operates using a $Waveform\ Lookup\ Table$ in order to bypass incurring the repeated computation cost of sine calculations. A precalculated wavetable, `double sineWaveTable[ ]` is stored within the CPP file. This array contains a predetermined number of precalculated values representing values of one cycle of the primitive sine function with an amplitude of 1. The size of the wavetable is 4800 values. This number, 4800, is calculated by dividing the highest possible sampling rate for $PCMWAV$ encoding, $98,000Hz$, by the lowest audible frequency, $20Hz$. While a lower number of precalculated values may be sufficient, some systems use a wavetable with as few precalculated values like 128, having a larger wavetable size ensures for much higher accuracy with less reliance on interpolative accuracy. Additionally, the extra overhead incurred by the memory allocated for a larger array of `double` values is negligible by the standard of modern computing hardware and is typically not a limiting factor.

The `sine` function first checks the `phase` value to ensure that it is within the bounds of the wavetable size. If the `phase` value is not within these bounds, then modulus mathematics is applied to the `phase` value to shift it to a value within those bounds. This `phase` value, as calculated by the `iteratePhase` method, serves as the access index of the wavetable. As this access index typically will not fall on an integer value, but rather some real number index, the value can not be simply accessed from the wavetable. Instead, the $floor$ of the `phase` value is calculated and four values are accessed. These values are the value at the floored index, the value immediately following that position, and the two values immediately preceding that position. With these four values, a $4 - PointPolynomialInterpolation$ algorithm is used to calculate the final value. The following is the code for the $4 - PointPolynomialInterpolation$ algorithm.

```
decimalPlaces = phase - floor(phase);
a = sineWaveTable[(long)phase - 1];
b = sineWaveTable[(long)phase];
c = sineWaveTable[(long)phase + 1];
d = sineWaveTable[(long)phase + 2];
x = 0.5f * (c - a);
y = a - 2.5f * b + 2.0f * c - 0.5f * d;
z = 0.5f * (d - a) + 1.5f * (b - c);
outputValue = double(((z * decimalPlaces + y) * decimalPlaces + x) * ↩
    decimalPlaces + b);
```

Listing 3.8: 4-Point Polynomial Interpolation implementation

The final `outputValue` is then returned from the function, which will be used for further Digital Signal Processing.

Since the calculation of values for the Saw, Square, and Triangle waveforms involve only elementary addition, subtraction, multiplication, and division, the cost of repeated calculations is far cheaper than repeated sine calculations. For this reason, the $Phase-to-Amplitude\ Converter$ functions for these waveforms are implemented as $WaveformFunctions$.

The `square` function contains a conditional block which will set the `outputValue` to either -1.0 or 1.0 depending on the current `phase` value. It will also perform modulus mathematics to ensure that the `phase` value is within the value range. Below is the core code block of the `square` function:

```
if (phase < 0.5 * waveTableSize)
    outputValue = -1.0;
else if (phase >= 0.5 * waveTableSize)
    outputValue = 1.0;
if (phase >= waveTableSize)
    phase -= waveTableSize;
```

Listing 3.9: The `square()` waveform function

The `triangle` function involves a conditional block controlling the linear transformation of the `phase` value to the `outputValue`. As with the previous functions, modulus mathematics is also performed on the `phase` value to ensure it remains within the valid range. The following is the main logic block of the `triangle` function:

```
if (phase <= 0.5 * waveTableSize)
    outputValue = ((phase - (0.25 * waveTableSize)) * 4)/waveTableSize;
else
    outputValue = (((waveTableSize - phase) - (0.25 * waveTableSize)) * 4)/↩
        waveTableSize;
```

Listing 3.10: The `triangle()` waveform function

Lastly, the `saw` function is the simplest of the *Waveform Functions*, involving a linear transformation of the `phase` value to the `outputValue`. The modulus arithmetic is present within this function in the same manner as the previous *Waveform Functions*. The logic block of the `saw` function is accomplished by this single line:

```
outputValue = phase/waveTableSize;
```

Listing 3.11: The `saw()` waveform function

The `outputValue` returned from the *Phase − to − Amplitude Converter* functions is sent from the `oscillator` class to the other Digital Signal Processing components. For each of the components, they all follow a similar input/output format. Each of the components take a `sample` value as a parameter, and return a `double` as their output. The `sample` value represents the output sample value from the previous component, and the returned `double` value corresponds to the processed output of the signal. Within this VST3 Plugin, the signal is routed from the `oscillator` component to the *Low − Frequency Oscillator Controlled - Amplitude Modulator*.

3.6.2 Low-frequency oscillator controlled amplitude modulator

The *Low − Frequency Oscillator Controlled Amplitude Modulator* acts similarly to an amplitude envelope on the signal. The main difference is that the amplitude modulation is not based on ADSR phases, as described previously in subsection 1.7.1, but based on the output values of an oscillator. Typically the output values of the oscillator are used as multipliers on the sample values from the input signal. In order to avoid silence at the nodes of the oscillator, the modulated signal is mixed back in with the dry signal to create the final output signal.

The main logic block of the code is as follows:

```
sampleMix = sample * (1 - lfoMixLevel01);
lfoMix = sample * lfo01.sine(lfoFrequency01) * lfoMixLevel01;
newSample = sampleMix + lfoMix;
```

Listing 3.12: LFO Mix

The parameters that the *Low − Frequency Oscillator Controlled Amplitude - Modulator* uses are `lfoSwitch`, `lfoFreq`, and `lfoBlend`. These three parameters are passed in by the `AudioProcessorValueTreeState`, which will be discussed in subsection 3.6.4. The `lfoSwitch` parameter controls whether or not the *Low−Frequency Oscillator Controlled Amplitude Modulator* will be bypassed or not. The `lfoFreq` parameter is assigned to the `lfoFrequency01` variable and controls the frequency of the Low-Frequency Oscillator. And finally, the `lfoBlend` parameter is assigned to the `lfoMixLevel01` variable which controls the final mix.

3.6.3 Amplitude modulation envelope

The `envelope` component modifies the amplitude of the signal value by the parameters set for the `envelope` component. The implemented type of *Envelope* component is the 4-step *ADSREnvelope* as described in subsection 1.7.1.

Within the `envelope` class, the most important values are the `attack`, `decay`, `sustain`, `release`, and `trigger` variables. The `attack`, `decay`, and `release` correspond to the rates of their respective phases. The `sustain` value represents the sustained amplitude during the *Sustain* phase. And lastly the `trigger` variable corresponds to the *noteOn* and *noteOff* MIDI event messages and is set by the `startNote` and `stopNote` methods as described in subsection 1.9.1.

The `envelope` component tracks its internal state via four switch variables. These variables are `attackphase`, `decayphase`, `holdphase`, and `release-phase`. These four switch variables correspond to their respective *ADSR* envelope phases. The envelope uses these switch variables to determine which state it operates in to process the input signal sample value.

3.6.4 AudioProcessorValueTreeState

One of the first crucial steps is to build the `AudioProcessorValueTreeState`, a JUCE framework tree object designed to facilitate the storage of state values of the VST3 plugin and the access and communication of this information between the

Digital Audio Workstation and the VST3 plugin.

The `AudioProcessorValueTreeState` is declared within the header file, and is initialized in the constructor list of the `AudioProcessor`. The following code block is an example of initializing the filter parameters:

```
tree(*this, nullptr, "PARAMETERS",
        {
            //Filter Parameters
            std::make_unique<AudioParameterChoice>("filterType",
                                                    TRANS("Filter Type"),
                                              StringArray("Low Pass","High Pass","Band ↩
                                                  Pass"),
                                                   0),
            std::make_unique<AudioParameterFloat>("filterCutoff", "FilterCutoff",↩
                 NormalisableRange<float>(20.0f, 10000.0f), 400.0f),
            std::make_unique<AudioParameterFloat>("filterRes", "FilterRes", ↩
                NormalisableRange<float>(1.0f, 5.0f), 1.0f),
        })
```

Listing 3.13: Initializing Filter Parameters

This block will initialize the specified MIDI control value and state values that the VST3 plugin will need to make accessible to the Digital Audio Workstation. Any state value that will only be used and modified internally does not need to be added to the `AudioProcessorValueTreeState` object.

In each initialization call, a call is made to a `std::make_unique<>()` object constructor. This template class constructor will be given a JUCE framework type corresponding to the VST3 standard type. For example, a frequency range for the Low Pass Filter is of type `AudioParameterFloat`, while a dropdown menu selection for selecting between filter types is of type `AudioParameterChoice`. The JUCE framework later converts these types to their respective VST3 SDK equivalents.

The `AudioProcessorTreeValueState` is connected the other classes in order to provide access to the VST3 plugin state. Most notably, the `processor` objects which contain the graphical user interface elements are initialized with references to state values stored within the tree object.

3.7 Testing

The unit testing for the project is implemented using `GoogleTest`, Google's in house C++ testing framework. The public Github repository, including the full documentation, for `GoogleTest` can be found at [14].

The unit tests written are focused on the waveform synthesis module of the application as this module is at the centerpiece of the direct digital synthesis and any issues in the synthesis module would create increasingly distorted artifacts in the final output.

The unit tests written for the application utilize the *Text Fixtures* feature of `GoogleTest`, which allows for tests to be grouped into a test fixture to utilize the same resource without incurring the cost of rebuilding and tearing down for each test.

Additional documentation and resources for writing GoogleTest test suites and test cases can be found on their primer [15].

3.7.1 Phase tests

The first set of unit tests are focused on testing the phase accumulation of each waveform function generator. The phase should be consistent and match expected outputs with the given frequency input. Additionally, the phase should never exceed 360 degrees and should return to 0 degrees after a full cycle at the given frequency. The phase values are obtained by calling the `getPhase()` function. Figure 3.3 shows a sample subset of the expected values used for the tests.

The C++ class `OscillatorPhaseTest` constructs the `oscillator` required to run the tests. The phase tests are part of the `OscillatorPhaseTest` test fixture which utilizes the class.

The `TestResetPhase` and `TestResetPhaseSpecificValue` functions test the `resetPhase()` method. These tests ensure that the oscillator can properly reset the phase value back to 0 degrees or to a specific degree value.

The `TestResetPhaseAfterSynthesis` test operates by running the `sine()` function for a whole cycle, then checking if the phase has reset as required back to 0. The frequency used for the test is 480hz with 1000 function calls made, which would complete a single cycle.

The next set of phase tests check to ensure that the phase value stays within expected values each and every time a synthesis function is called. The `Sine-SynthesisPhaseTest` calls the `sine()` function 99 times. In each iteration, an `EXPECT_FLOAT_EQ()` is called to check the value of the phase returned by `getPhase()` against the expected value. Should the test fail at any iteration, the iteration is reported back in the test failure message. The `SineSynthesisPhase-`

`ResetTest` is similar, however it runs the `sine()` function for 100 times. After the 100th iteration, it runs `EXPECT_FLOAT_EQ()` to check if the value of the phase reset back to 0 degrees as expected.

These tests are also made for each of the different waveform function generators, i.e. *Square*, *Saw*, and *Triangle*. The respective tests are called `SquareSynthesis-PhaseTest` & `SquareSynthesisPhaseResetTest`, `SawSynthesisPhase-Test` & `SawSynthesisPhaseResetTest`, and `TriangleSynthesisPhase-Test` & `TriangleSynthesisPhaseResetTest`. One additional test exists for the *Saw* waveform generator, the `SawSynthesisPhaseFlipTest`. As the *Saw* waveform generator modifies the phase value between $[-360, 360]$ degrees, this test checks if the phase value flips at the expected iteration.

3.7.2 Value tests

Value unit tests are crucial to ensure that each waveform function generator will create the waveform as expected. The best way to test for this is to check the generated values at specific phase values with known output values. Additionally, the generated values should never exceed the range of $[-1.0, 1.0]$. Figure 3.3 shows a sample subset of the expected values used for the tests.

The C++ class `OscillatorValueTest` constructs the `oscillator` required to run the tests. The phase tests are part of the `OscillatorValueTest` test fixture which utilizes the class.

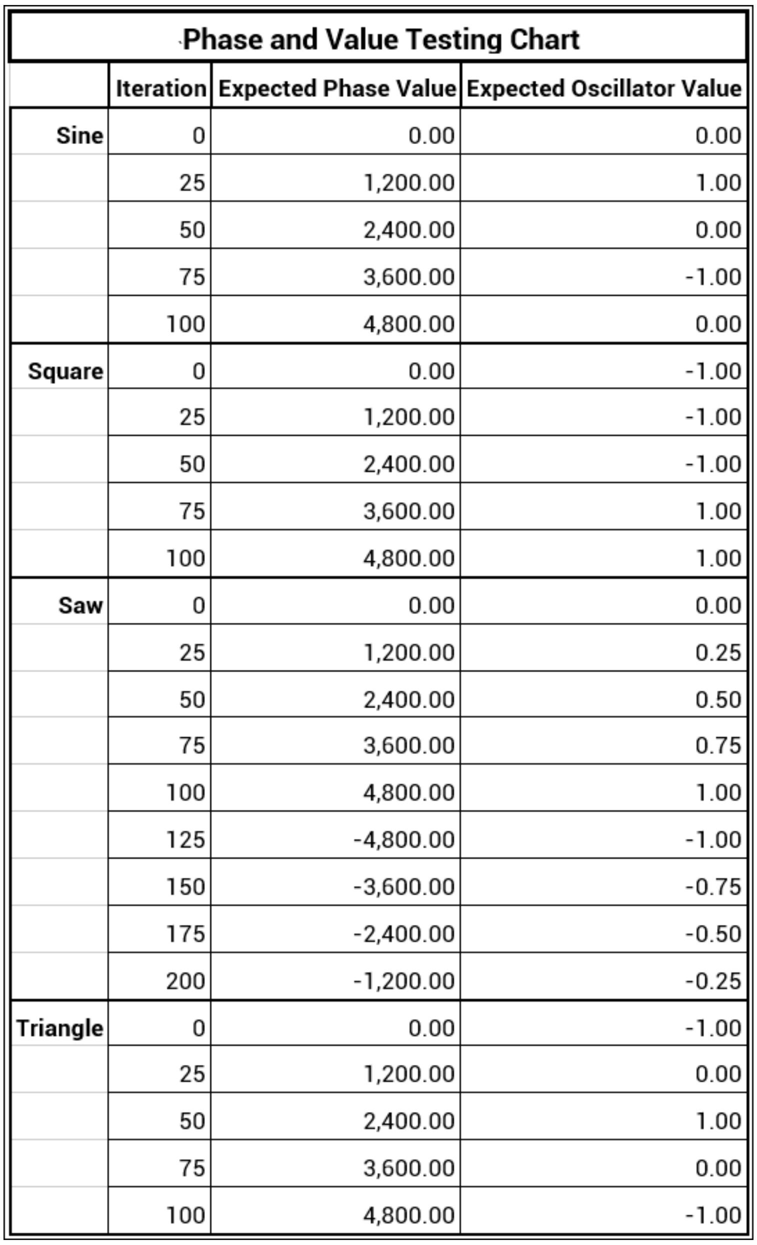
The `SineSynthesisValueTest` test operates by setting the phase to 0, 90, 180, 270, and 360 degrees, and checking the value generated at each phase value by the `sine()` function to the expected values from the respective phase values. Any value not matching the expected values is reported in the test failures along with the phase which it failed at.

The functions are similar for the *Saw*, *Square*, *Triangle* waveform generators, and are tested via the `SawSynthesisValueTest`, `SquareSynthesisValueTest`, and `TriangleSynthesisValueTest` tests respectively.

The `SineSynthesisValueRangeTest` performs 100 iterations of the `sine()` function. At each iteration, the value generated is checked to ensure that it is within the range of $[-1.0, 1.0]$. If any iteration fails the check, the iteration is reported back in the test failure.

Each of the other waveform function generators also have their respective value range tests. These are `SquareSynthesisValueRangeTest`, `SawSynthesis-`

`ValueRangeTest`, and `TriangleSynthesisValueRangeTest`.



Phase and Value Testing Chart

	Iteration	Expected Phase Value	Expected Oscillator Value
Sine	0	0.00	0.00
	25	1,200.00	1.00
	50	2,400.00	0.00
	75	3,600.00	-1.00
	100	4,800.00	0.00
Square	0	0.00	-1.00
	25	1,200.00	-1.00
	50	2,400.00	-1.00
	75	3,600.00	1.00
	100	4,800.00	1.00
Saw	0	0.00	0.00
	25	1,200.00	0.25
	50	2,400.00	0.50
	75	3,600.00	0.75
	100	4,800.00	1.00
	125	-4,800.00	-1.00
	150	-3,600.00	-0.75
	175	-2,400.00	-0.50
	200	-1,200.00	-0.25
Triangle	0	0.00	-1.00
	25	1,200.00	0.00
	50	2,400.00	1.00
	75	3,600.00	0.00
	100	4,800.00	-1.00

Figure 3.3: Phase and value testing chart

Conclusion

The project was successful and performs not only to my original specifications, but even surpassed it as I was able to include more features than originally intended. The current application features VST3 standard compatibility, as demonstrated by its seamless integration into a digital audio workstation. Coming with this integration is the capability of accepting MIDI input from either a live MIDI controller or prescripted MIDI notation. The plugin then is able to produce synthesized sounds with low latency, and output the audio via the ASIO drivers.

The plugin allows for flexible customization of the synthesizer sounds. The combinations possible with the primary oscillator, additive secondary oscillator, and subtractive tertiary oscillator allow for sophisticated waveforms, taking the sound far past the 8-bit aesthetics of single oscillator synthesizers and into the sophistication of 1980's synthesizer aesthetics. Additionally, I was able to implement the Envelope, Filters, LFO, and Delay signal post processing functionality to the synthesizer, allowing for even more customization of the resulting synthesizer sounds. The VST3 integration into the DAW allows for deep integration into the workflow, opening up the possibility for a much more professional sounding result.

As a music producer myself, I was extremely satisfied with the final result. The plugin permitted me the level of control and flexibility to achieve a customized sound from my digital synthesis plugin, and I was capable of creating an entire workflow built solely around instances of this digital synthesis plugin within FL Studio, my DAW of choice. From this workflow, I was able to easily produce music without any problems.

Additional work can be done in the future to further expand the capabilities of the digital synthesizer. Frequency Modulation, both static envelope and low frequency oscillator controlled, could be added for a more 21st century sound and the capability to create dubstep synths. A visualizer can be added to display the output waveform to allow the user to better understand the sound being generated and how the signal post processing being applied modifies the audio.

Bibliography

[1] Made With JUCE | JUCE
https://juce.com/discover/made-with-juce.
Retrieved 15 December 2020.

[2] JUCE: Class Index
https://docs.juce.com/master/index.html.
Retrieved 15 December 2020.

[3] JUCE for VST Development
http://www.redwoodaudio.net/Tutorials/juce_for_vst_development__intro.html.
Retrieved 15 December 2020.

[4] Source Code on Github
https://github.com/ParadoxChains/C3JI5D-Evan-Sitt-BSc-Thesis.
Retrieved 15 December 2020.

[5] Visual Studio 2019 IDE
https://visualstudio.microsoft.com/vs/.
Retrieved 15 December 2020.

[6] Visual Studio documentation
https://docs.microsoft.com/en-us/visualstudio/windows/?view=vs-2019.
Retrieved 15 December 2020.

[7] MIDI.Org Home
https://www.midi.org/.
Retrieved 15 December 2020.

[8] MIDI Specifications
https://www.midi.org/specifications.
Retrieved 15 December 2020.

[9] FL Studio Overview
https://www.image-line.com/fl-studio/.
Retrieved 15 December 2020.

[10] FL Studio Tutorials
https://www.image-line.com/fl-studio-learning/.
Retrieved 15 December 2020.

[11] FL Studio Manual
https://www.image-line.com/fl-studio-learning/
fl-studio-online-manual/.
Retrieved 15 December 2020.

[12] The Beginner's Guide to Audio Synthesis
https://www.blackghostaudio.com/blog/
the-beginners-guide-to-audio-synthesis
Retrieved 15 December 2020.

[13] Oswinski, B.: The Mixing Engineer's Handbook, 4th Edition Bobby Owsinski Media Group, 2017.

[14] GoogleTest Github Repository
https://github.com/google/googletest.
Retrieved 15 December 2020.

[15] GoogleTest Primer
https://github.com/google/googletest/blob/master/
googletest/docs/primer.md.
Retrieved 15 December 2020.

[16] Standing Waves
http://230nsc1.phy-astr.gsu.edu/hbase/Waves/standw.html.
Retrieved 15 December 2020.

[17] Noise Level Chart: Decibel Levels of Common Sounds With Examples
https://boomspeaker.com/noise-level-chart-db-level-chart/.
Retrieved 15 December 2020.

[18] Cello Tone Analysis
https://vobarian.com/celloanly/index.html.
Retrieved 15 December 2020.

[19] All About Direct Digital Synthesis
https://www.analog.com/en/analog-dialogue/articles/
all-about-direct-digital-synthesis.html#.
Retrieved 15 December 2020.

[20] Additive Synthesis Basics
https://www.youtube.com/watch?v=ZAgiC0vtc8M&ab_channel=
DanHosken.
Retrieved 15 December 2020.

[21] ADSR Envelopes: How to Build The Perfect Sound [Infographic]
https://blog.landr.com/adsr-envelopes-infographic/.
Retrieved 15 December 2020.

[22] Planet of Tunes - Synthesis types
http://www.planetoftunes.com/synthesis/synthesis-types.
htm.
Retrieved 15 December 2020.

[23] STM32F4-Discovery MIDI input and basic synthesis
https://minddumpdotnet.wordpress.com/2013/05/23/
stm32f4-discovery-midi-input-and-basic-synthesis/.
Retrieved 15 December 2020.

[24] Linear filter
https://en.wikipedia.org/wiki/Linear_filter.
Retrieved 15 December 2020.

[25] What Is a Low Pass Filter? A Tutorial on the Basics of Passive RC Filters
https://www.allaboutcircuits.com/technical-articles/
low-pass-filter-tutorial-basics-passive-RC-filter/.
Retrieved 15 December 2020.

[26] Virtual Studio Technology
https://en.wikipedia.org/wiki/Virtual_Studio_Technology.
Retrieved 15 December 2020.

List of Figures

Listings