Music Synthesizer

Candidate No: 149486

Games and Multimedia Environments (G6062)

Project Supervisor: Kingsley Sage

2019/20

Statement of Originality

This report is submitted as part requirement for the degree of Games and Multimedia Environments (G6062) at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I hereby give permission for a copy of this report to be loaned out to students in future years.

Signed:

1 CONTENTS

2 INTRODUCTION

2.1 ABSTRACT

The aim of this project is to create a software audio synthesizer that can be played like an instrument. The intended users of this synthesizer are music producers, composers and musicians. It utilises various C++ libraries to create a simple single voiced synthesizer with an effects chain to modify the sound further. It also allows for real-time modulation of the parameters within the synthesizer, allowing a user to create a wide range of sounds and switch between them on demand. It can be compiled as a standalone executable file, or as a plugin that can be opened within a larger audio program, meaning the synthesizer can be used in much larger electronic music productions. The method of synthesis chosen was subtractive synthesis for quick and easy shaping of sounds.

2.2 BACKGROUND

Music synthesis has become a popular creative tool in music creation as it allows for precise control over the final sound that is created. Users interact with the synthesizer by plugging a MIDI (1) keyboard into the computer to play notes and moving on-screen controls, typically knobs and sliders, to modify the sound. The sound is created by using oscillators or recorded samples to generate an initial audio signal, which is typically then sent through a Digital Signal Processing chain. Digital Signal Processing, or DSP (2), is the process of applying various effects to the audio signal through analysis and modification of the incoming signal. The various methods and techniques used within audio synthesis are explored further within the background research section, section 4.1, of this report.

Synthesizers are normally used within a real world setting as a single instrument within a larger composition, with software synths often being loaded into a larger program called a Digital Audio Workstation (commonly referred to as a DAW) (3). A DAW, among other things, provides the means of creating a musical timeline with multiple different audio tracks and mixing capabilities.

The synthesizer will be compiled as a VST (Virtual Studio Technology) (4) plugin, which is a standard format that allows it to be loaded and used as a single instrument within a DAW to help create a complete composition. VST is very commonly used, since every major DAW supports it. Because this synthesizer will be loaded with many other plugins and will be required to generate sounds in real-time, one of the major factors in this project will be optimization in order to lower the latency. Another major issue when it comes to software synthesizers is GUI (Graphical User Interface) design. A large amount of information and controls need to be displayed in a single window, potentially leading to the synth being cluttered or difficult to use. Another common feature of software synths is the ability to save the current sound as a preset, allowing the user to load the same sound in a different song.

The goal of this project is to create a synth that is relatively easy to use without sacrificing on customization options. The desirable features mentioned above, such as optimization and a solid UI will be core objectives of this project. As well as these, another objective is to distinguish it from other synths in various ways. The main distinctive feature is the ability to include an arbitrary number of oscillators as well as the ability to individually add effects to each oscillator. This will enable users to create much more dynamic sounds without having to load up multiple synths. This report will list out the requirements of this project in more detail, as well as provide a project plan to ensure the finished product is delivered on time.

2.3 Report Structure

Section 3 details the initial research that was undertaken before the design of the solution was started. This consists of a simple explanation of the theory behind the different types of audio synthesis, as well as outlining the research into existing synthesisers.

Section 4 is the professional considerations, which covers how the project will consider public interest, professional competence and integrity, and duty to relevant authority.

Section 5 outlines the requirements of the project in order to determine the scale and focus of the expected final project. This covers both the functional and non-functional requirements of the final system.

Section 6 explains how the final system is designed. The learning objectives for the project are discussed, as well as both the front and back end functionality. Finally, a high-level breakdown of how the class structure is designed and how each class functions.

Section 7 describes all the problems that were encountered during the implementation of the project. It outlines the general order in which the problems occurred and how they were dealt with.

Section 8 shows the evaluation and testing of the system. It covers both quantitative analysis to test the accuracy of the sound waves generated as well as a self-reflection of the overall outcome of the project and highlighting any other outstanding issues the system has.

Section 9 is the conclusion, summarising everything that this report has covered and discussing possible alterations that would be made had this project been done again.

Section 10 discusses the further work that could be done on this project in future, reflecting on the requirements that were not completed and explaining how they could be implemented.

3 Background Research

There have been various different methods of audio synthesis developed over the years, with the most popular ones including subtractive synthesis (6), additive synthesis (7) and FM (Frequency Modulation) synthesis (8). All three of these methods are closely related as they all rely on oscillators, as opposed to samples, to generate the initial signal. However, they all have distinct differences in function, workflow and performance. This makes them useful in different situations and are better at generating different types of sound.

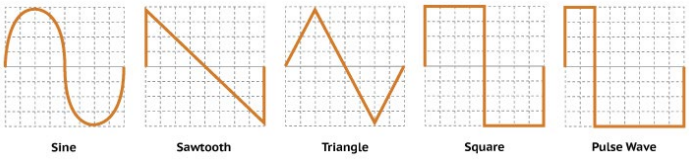


Figure 1. Oscillator wave types, from Szczepaniak, 2014 (6)

Subtractive synthesis works by using oscillators that create simple mathematically generated waveforms, such as sine waves, sawtooth waves and square waves (shown in Figure 1). These signals are then routed through the various DSP effects and filters to shape the sound. It is known as subtractive synthesis because typically, one or more cut-off filters are applied which reduce the volume of certain frequencies in the signal. Subtractive synthesis also makes use of LFOs (Low Frequency Oscillators) (6), which are used for control of other parameters in the synth. As Figure 2 shows, all these elements get routed into an amplifier at the end to regulate the amplitude of the final signal, with the LFO able to modify the amplitude over time. The amplifier is also controlled by an envelope generator (6), allowing a sound to fade in as the note is pressed and fade out as the note is released. Subtractive synthesis is the most widely used method, as it allows the user to create a wide range of sounds very quickly and intuitively.

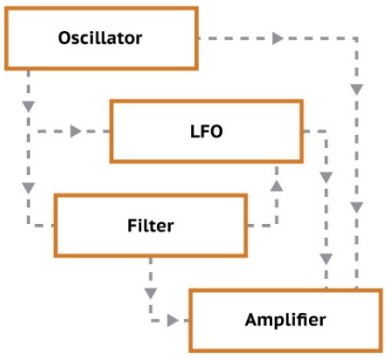


Figure 2. Subtractive synthesis pipeline, from Szczepaniak, 2014 (6)

Additive synthesis exclusively utilizes oscillators that generate sine waves and completely forgoes DSP. Sounds in an additive synthesizer are crafted by adding many sine waves together to generate a more complex tone. Due to the nature of sound, Additive synthesis theoretically allows any sound to be created. However, Additive synthesis remains less commonly used, as it is a less intuitive way of creating sounds and requires many oscillators, increasing the resources required to run.

FM synthesis works by assigning one or more oscillators as carriers and other oscillators act as modulators. The carriers are the waves that get routed out to the speakers, but modulators do not. The frequency of the carrier is rapidly moved up and down, or modulated, according to the value of the modulators. This allows for extremely complex and dynamic sounds to be created relatively

easily, particularly atonal sounds for percussion, but it requires a large amount of prior knowledge to get used to the workflow of FM synthesis.

There are numerous software synthesizers on the market; for simplicity, only two of the most popular examples will be discussed. Both are subtractive synthesizers, as the proposed system will be a subtractive synthesizer. These synths are solid examples of the most complex and powerful solutions available, but both suffer from various usability issues and information overload, meaning they both have a steep learning curve. They also suffer the same limitation when it comes to effects. Each effect is applied to every oscillator, which is useful when creating a unified distinct sound. However, if your goal is to create a full wall of sound or a soundscape, the only way of achieving that is by loading up multiple instances of the synth, slowing down the workflow and introducing unnecessary resource overhead.

3.1 MASSIVE

The first, pictured on the right, is Massive, created by Native Instruments (9). This synth allows for extensive customization and control over sounds. The window consists of many small modules, each of which corresponds to a different element of the synth. Oscillators are on the left-hand side, filters and other effects are on the top, envelopes and LFOs are below that and on the bottom right are knobs that the user can manually assign to change other parameters. This synth is very customizable and allows for a wide range of sounds to be created. The biggest drawback to this synth is within the UI design, which is very cluttered and difficult to understand without extensive prior knowledge of synths. It also has little visual representation of the inner workings and audio pipeline of the synth. This requires users to read documentation or follow tutorials in order to gain a deeper understanding of how to use it. It also has a fixed complexity, providing only 3 oscillators, 2 effects, 2 filters and one equalizer.

Figure 3. Screenshot of Massive, from Native Instruments, 2018 (9)

3.2 SERUM

Figure 4. Screenshot of Serum, from Xfer Records, 2018 (10)

The second example, pictured on the right, is Serum, developed by Xfer Records (10). This is also a very customizable synth allowing for incredibly complex sounds, but suffers from similar limitations to Massive. The most notable feature of Serum is that it is a wavetable synth. This means instead of simple mathematical oscillators, the user can load in custom waveforms and dynamically morph between waveforms as the sound is playing, with the current waveform being shown visually in real-time. This makes understanding the synth much more intuitive and helps users craft sounds more accurately. As well as this, Serum allows for immense modulation possibilities by clicking and dragging certain elements to other knobs. Serum can handle much more modulation possibilities than that, but any other connections have to be made within a separate matrix menu. Unfortunately, this matrix menu also suffers from the usability issue, where it is not immediately obvious how to use it without looking up tutorials. Serum also limits the complexity of the sound, being limited to three oscillators, one of which is much more limited than the others. It also only allows for one instance of each effect, with every effect being applied to the entire sound as opposed to allowing for different oscillators to have different effects applied to them. The overall learning curve for Serum is much less extreme than the learning curve for Massive, but it becomes a lot less intuitive once you dive deeper into the functionality of the synth.

4 PROFESSIONAL CONSIDERATIONS

The professional considerations for this project are ensuring conformity with the BCS code of conduct (5), as third-party libraries and graphics may be used. This section will break down the details of these considerations. All of these considerations were upheld fully during the course of this project.

4.1 PUBLIC INTEREST

This project will ensure that any third-party content will be open source/royalty free and appropriately referenced. Any licensing requirements for the use of third-party content will be upheld. The project will not require direct contact with the general public, so will not infringe on the public health, privacy, security or wellbeing of anyone.

4.2 Professional Competence and Integrity

The subject matter and scale of the proposed system does not go beyond the capabilities of an undergraduate Computer Scientist. Throughout this project, all legislation will be adhered to and efforts will be made to remain up to date on the developing technology. Any valid criticisms of the project will be respected and accepted.

4.3 Duty to Relevant Authority

The supervisor of this project will be made aware of the progress and any potential setbacks that occur, as well as being given a full and honest report of the performance of the final product at the end of the project. During the project, periodic meetings will take place in order to keep the supervisor up to date on the status of the project.

5 Requirements Analysis

A synthesizer's job is to allow the user to create a large amount of different sounds to be used within a song. Therefore, it is desirable to have as much control over the sound as possible, but it's also integral to allow rapid creation of sounds as many musicians are under time pressure to complete songs. The dilemma this causes requires a careful design in order to balance the speed of workflow and the amount of creative possibilities the synth can provide. This section will consider an ideal system, followed by outlining the proposed system taking into account time and resource limitations.

5.1 Overview

The research into existing synths has highlighted a number of unfulfilled features that an ideal synth could provide. It would be extremely helpful to have a synth that utilizes the power that each method of synthesis provides. A synth that allows an additive approach without limiting the oscillators to sine waves would minimize computational overhead. The same synth could also contain the filters and LFOs from subtractive synthesis within the pipeline to allow for intuitive and quick sculpting of sounds. The fully customizable waveforms that wavetable synthesis provides would also add another method of shaping the sound initially with more complexity. An ideal synth would allow not just for Frequency Modulation, but also near universal modulation, where any parameter within the synth could be used to modulate any other parameter. This would increase the power of the synth immensely. An ideal synth would also allow for an arbitrary number of oscillators and voices to be created, all with full control over the sound each one generates. Finally, a fully modular effects chain in which an arbitrary number of effects can be assigned to oscillators independently as well as within groups. This effects chain would lend itself to creating a sound that feels as though it is being created by multiple sources. All of these components together would give the user a multitude of creative options and would cater to every style of workflow, with every type of sound being easily achievable.

The biggest problem with this ideal synth becomes apparent the moment the UI design is considered; near complete modulation capability, multiple different possible types of oscillators and arbitrary scalability all lead towards a UI that is cluttered and dense, rendering it unusable. Another issue that arises is the limited speed of the hardware. Allowing the user to load in an unbounded number of oscillators and effects would very quickly drain the system of resources, meaning the latency would be unreasonably high. This is solved by adding upper bounds on the number of oscillators and effects. Another aspect to consider is the speed of workflow. Having so many options available can overwhelm the user or slow their progress down significantly.

For this project, the focus will be on creating a simple to use, intuitive synth with an arbitrary, but bounded, number of oscillators and effects. The architecture will separate each oscillator into its own mixer-style channel, allowing effects to be placed on individual oscillators, with functionality to copy effects over into other channels. It will take a subtractive approach, with a handful of simple waveforms the user can choose between and allowing for polyphony. The use case this has been designed around is pads, which are thick, complex and dynamic. It would also enable full ambient soundscapes to be crafted with finer control over how each part of the sound contributes to the overall sound. It would be preferable to provide more extensive modulation options, but this is unachievable within the timeframe, so only simpler modulation options are being considered.

5.2 Functional Requirements

The list of functional requirements are as follows:

- The system shall be able to take input from a MIDI source and use it to generate a sound of the correct pitch.
- The system shall allow the user to modify the nature of the sound using GUI controls.
- The system shall provide functionality for DSP effects to be applied to the oscillators.
- The system shall modulate the amplitude of the oscillator signal using an envelope generator.
- The system should allow the user to save the current sound as a preset.
- The system should provide controls for the attack, sustain, decay and release of each oscillator.
- The system could allow the user to edit the waveform generated by the oscillator.
- The system could have switchable quality modes to allow the choice between more responsive playback and higher quality output.
- The system could allow the user to assign a MIDI control, such as a knob or slider, to modulate oscillator or effect parameters.

5.3 Non-Functional Requirements

The list of non-functional requirements are as follows:

- The system shall use at least one oscillator in order to generate the sounds.
- The system shall be loadable into host applications via the VST format.
- The system shall be written in C++ using a third-party library for graphics.
- The system shall contain a simple delay effect and a simple cut-off filter.
- The system should allow for polyphony and multiple oscillators.
- The system should have a limit on the number of voices to keep resource usage consistent.
- The system should not exceed a latency of 100 milliseconds.
- The system should create a clean, accurate audio signal.
- The system should contain a longer list of effects, including, but not limited to:
    - Compressor/Limiter
    - Distortion
    - Complex Filters
    - Chorus
    - Flanger
    - Phaser
    - Reverb
    - Tremolo/Vibrato
    - Bitcrusher
    - Noise Gate
    - Stereo Effects
- The system could allow for modulation using LFOs.
- The system could allow an arbitrary number of oscillators and effects to be loaded at once

6 High Level Design

6.1 Learning Objectives

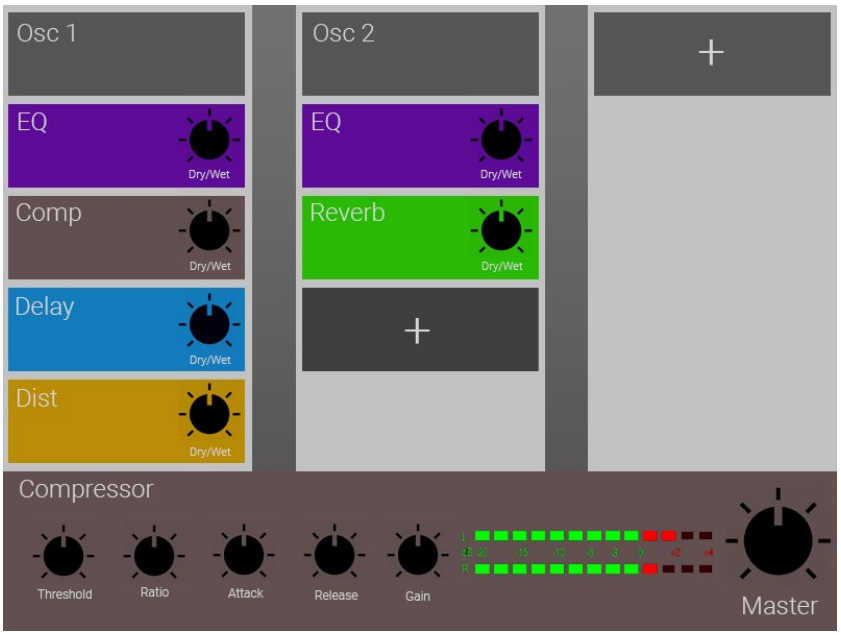
- Learn how DSP works
- Find and learn an easy to modify library that handles the front end for potential future projects
- Strengthen C++ knowledge
- Deepen sound design knowledge
- Experiment with different audio effects

The primary learning objective for this project was to learn how synthesizers and DSP works, as well as finding a decent library to build a synthesizer with and learning how it works so it might be used in potential future projects. This meant that the first major choice was finding an appropriate library that handles GUI, since writing the GUI from scratch was not important to the project. Another function the library needed to handle was interfacing with the audio drivers and the speakers, since this would be too complex and take up a large portion of coding time. The final feature, which was not essential but much desired, was the ability to compile as a VST plugin, so the program could be loaded into a DAW.

The choice of language was an easy one, as C++ is one of the only languages to be efficient enough to handle real-time audio playback, where samples need to be calculated at over 40,000 times a second. As a result, most of the libraries available are written in C++. That along with prior experience with coding in C++ made this the obvious option. That made another learning objective to strengthen proficiency in C++.

6.2 Front End

Figure 5. Early Prototype of the synthesizer GUI



The original design prototype for the UI is shown in Figure 5. This concept was to allow for items to be added and removed dynamically with buttons and dropdowns to select the effect to add. The idea was also to allow the effects to be dragged around to change the order of the effects chain.

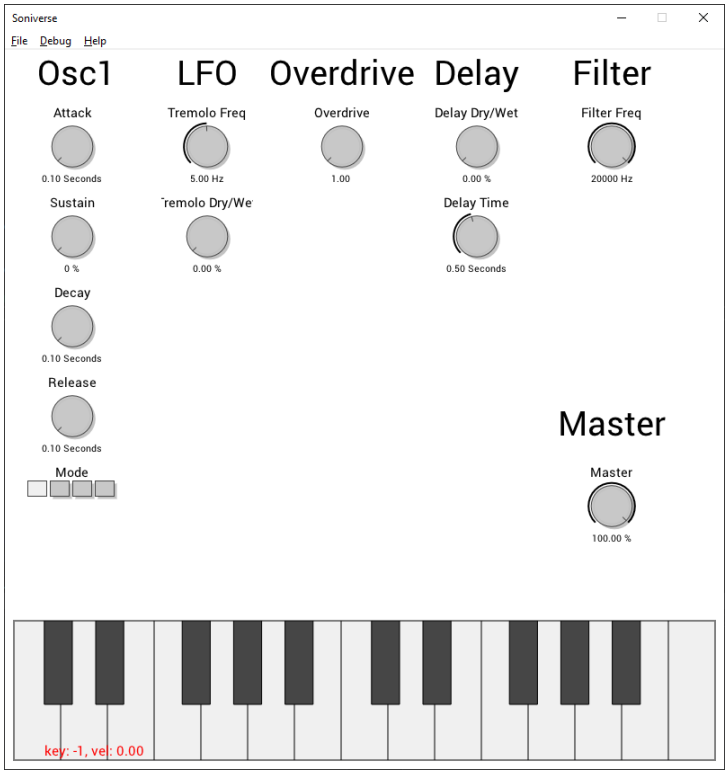


Figure 6. Screenshot of the finished Synthesizer GUI

The final UI (Figure 6) was stripped down due to the issues explained in section 7. The general layout of the UI is heavily influenced by the existing solutions researched in Section 3, consisting of a virtual keyboard on the bottom and various knobs to control all the parameters of the synthesizer, which will be explained in detail in the class breakdown (6.4). The UI is a fixed size window to allow for a consistent look and avoid the problems that arise with scalable UI. The knobs and buttons on the UI all change parameters of the synth and all of the associated effects. There is also a virtual keyboard so that the synth can be played without a MIDI input. Each control has a minimum and maximum value enforced in order to stop the user from breaking the program by changing values too high or low. These minimums and maximums are defined and justified in the class breakdown (6.4). The layout of the knobs is designed with large gaps between the columns to make it clear which part of the process is being changed.

6.3 Back End

Once the learning objectives were decided, preliminary research of possible solutions started. In the end, the list was narrowed down to two possible libraries for the final solution. Further research was done on these two solutions to determine which one would be more appropriate for this project.

6.3.1 JUCE

The first possible solution investigated was JUCE (12). JUCE initially seemed to tick all the boxes needed, providing a GUI implementation with an easy to use editing tool as well as dealing with audio drivers. However, upon closer inspection, the back-end features such as oscillators and envelopes were already built into it with no easy way to write new ones, meaning it was not a great choice for a coding project.

6.3.2 iPlug

The decided upon library was iPlug (13). iPlug is an open source C++ library, which provides a coding framework to handle compilation into many different formats, including VST, as well as handling audio output and providing GUI controls and tools to make building the GUI much easier. iPlug was chosen because it outputs high quality audio and doesn't require any extra work to port

the synth into the VST format, speeding up workflow considerably. Another benefit of iPlug is it requires no extra work to compile it into different formats, including a standalone .exe file, meaning the final program can be available to many more computers and DAWs.

6.3.3 SAVIHost

With the library chosen, the next step appeared to be finding a VST host to be able to test the code that was being written. The role of a VST host is to allow users to load up and test VST plugins, since VSTs are not standalone executables. SAVIHost was an easy choice for this, as it was a lightweight and easy to use VST host. The host acts as a wrapper for the VST and handles MIDI inputs. In the end, this ended up not being useful, as iPlug handled MIDI input anyway and allows for compiling the program as a standalone .exe file. As a result, SAVIHost was not used in this project.

6.4 Class Breakdown

On the right (Figure 7) is the class diagram, which shows the classes and how they interact with the library. Firstly there is an effect interface which all three effect classes implement. The LPFilter and Delay classes require a buffer, as explained later. All of these effects, as well as an oscillator and envelope generator are added into a rack class, which acts as a wrapper. As a result, the rack class is the only one the main Soniverse class needs to initialise. This was done to guarantee that only one instance of each class would be created, avoiding messy code and potential referencing issues. Every association is a 1 to 1 relationship, so no numbers were added.

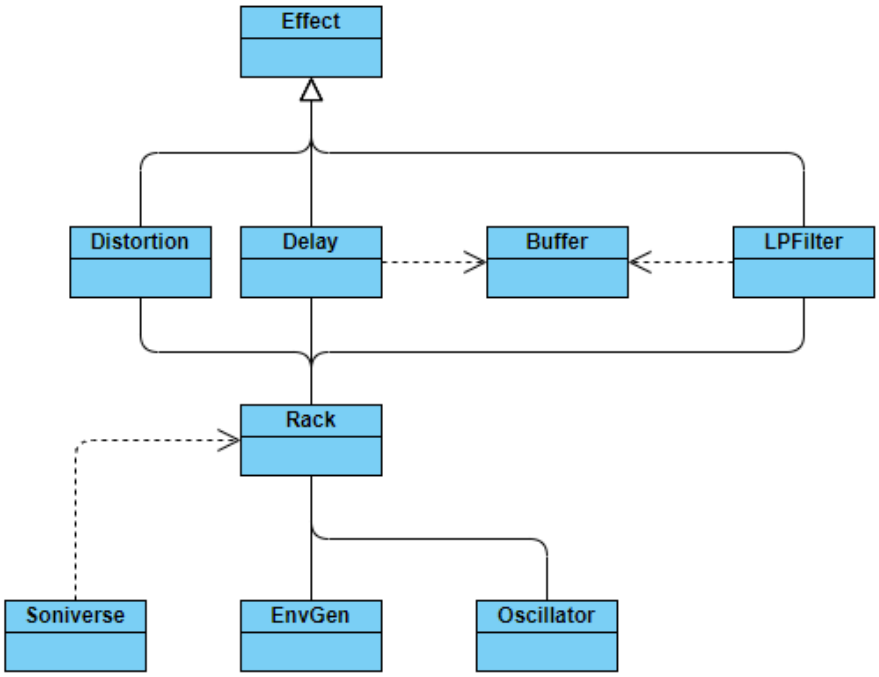


Figure 7. Class Diagram for the system

6.4.1 Oscillator

The oscillator class generates mathematical waves such as sines, squares and sawtooths. This is achieved by calling a process function every sample to calculate the next sample in the wave. It keeps track of where it is in the wave by using a phase value, which increases by the frequency of the note divided by the sample rate each sample. This means one cycle of the wave will cause the phase to increase from 0 to 1, at which point the phase is reset to 0 in order to stop the phase from

increasing indefinitely and causing overflow. The frequency of the note is calculated in iPlug, since it is based on the MIDI input. It works by choosing a note, in this case middle C at 440Hz, then multiplying based on the note that is being pressed. The mode of the oscillator is defined by an integer rather than a string, which means it both takes up less space and is easier to be accessed and edited by the GUI.

The sine wave is calculated using the C++ math library, whereas square and sawtooth waves are both calculated by doing simple modifications to the phase. Square just needs to be +1 for half of the phase and -1 for the other half, so the function just checks to see if the phase passes 0.5. The sawtooth wave is just a moved and scaled up version of the phase so it moves from -1 to 1 instead of from 0 to 1. There is also a white noise generator that just outputs a random number between -1 and 1.

6.4.2 Envelope Generator

The envelope generator defines the change in volume of the oscillator over time (as shown in figure 8) to make the sound more like a real world instrument. In this project, an envelope generator is implemented by creating a finite state machine, starting in an idle state. Once a key is pressed, it moves to the attack phase until the volume reaches the maximum value. Once that is reached, it transitions into the decay phase until the volume reaches the sustain value. After this happens, it remains in the sustain phase, maintaining the same volume until the key is let go. When the key is let go, it goes to the release phase, decreasing in volume until it reaches zero and the state moves back to idle until another key is pressed. The release phase can be jumped to from any other state as the key can be released at any point, but this doesn't change how it functions.

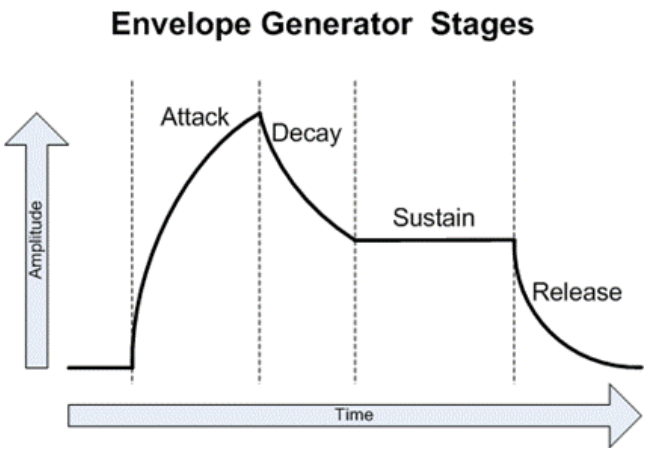


Figure 8. The states of an envelope generator (14)

The minimum values defined in the attack, decay and release variables are 0.1 seconds, because setting any of these to zero would most likely break the program or cause audio artefacts, so it is set close to zero instead. The maximum values are set to 10 seconds as any longer than that is unlikely to be needed in an instrument. The sustain variable remains between 0 and 1 as it's defining the volume of the sound and not the time it takes to reach a volume.

6.4.3 LP Filter

A low pass filter removes the high frequencies from an audio signal and is the main tool used to sculpt a sound in a subtractive synthesiser. There are many ways to achieve this, with complex implementations requiring Fourier analysis and transforms, but for this project, a simple

approximate implementation was chosen to save coding time and computational power. This works by storing a buffer of the last few samples and depending on the frequency cut-off, taking the mean of the last few samples. The way this is calculated is using the Nyquist frequency. The Nyquist frequency is the number of samples needed to accurately store sounds of a certain frequency, which ends up being double the frequency, as you need one sample for the peak of the sound when it reaches 1 and another for the peak when it reaches -1. The sample rate of approx. 40,000 samples per second allows for sounds up to 20,000Hz to be stored accurately, which is the highest frequency that humans can hear. This can be used to lower the highest possibly frequency. For example, averaging the last 2 samples will make the effective sample rate 20,000, meaning only sounds up to 10,000Hz are captured. Therefore, the number of samples to be averaged in the system is 20,000 divided by the maximum frequency desired. As mentioned before, this is not a precise conversion because the sample rate hasn't been changed, so some high frequencies may get through, but they will still be made much quieter overall.

The upper limit for the low pass filter is 20,000Hz, which as mentioned before, is the highest frequency the human ear can hear. The lower limit is set to 100Hz, because a low pass filter lower than that will be quickly approaching the lowest frequency a human ear can hear.

6.4.4 Distortion

Typically, errors in the audio engine were characterised and identified by clipping, which is when the value of the sample goes out of the defined boundary of -1 to 1. This means the value is capped at the boundary, which means any information louder than that is lost and replaced with a flat line at 1 or -1. The distortion effect plugin utilises this intentionally to create a gritty sound whilst allowing the use to bring it down to the desired value.

6.4.5 Delay

A delay stores the audio signal, plays it after a set amount of time, and is used in a synth to add dynamics to the sound. This is implemented in this project by adding a simple buffer using the same structure as the filter's buffer. This will store the input into a queue, which will be added to the main signal after the delay time has passed. Due to this implementation, the delayed signal will then also be stored into the buffer again, meaning it will keep being repeated infinitely if left unchanged. To stop this, the delayed signal is replayed at a lower volume, making it sound like it's fading away. Due to this implementation, changing the length of the delay is as simple as changing the size of the buffer. If a second delay were required, the size of the queue would be equal to the sample rate, meaning it is a simple multiplication to convert from seconds to samples.

The lower limit for the delay is 0.1 seconds, as the envelope generator's parameters also limit to 0.1 seconds, so any difference in delay time below that wouldn't be noticeable and just muddy the sound. The upper limit is set to 1 second due to buffer limitations discussed in section 7.

6.4.6 Rack

The rack was created as a container for all the various oscillators and effects, in order to make editing and adding functionality easier. It means that when the library asks for an oscillator, instead of making a new one every time, it just calls to the one stored in the rack. Another purpose of the rack is to make the effects chain a lot more efficient and easier to edit. All of the effects are stored in an array within the rack so that all the program has to do is iterate through each effect and call the process function of each one in turn.

7 Implementation Issues

The project came across many issues during development. Most of the problems were related to the iPlug library that had been chosen. Due to it being one of a very limited number of libraries available, using it was integral to the project. However, the documentation for this library was very limited and large sections of code were completely undocumented. This meant a large portion of the allotted implementation time was spent learning the structure and flow of the library before any coding could be started.

The first major issue occurred when revisiting the project after temporary withdrawal from university. The chosen library had a large update, changing how the structure of the library worked. This required some time to port over the earlier progress that had been made.

When creating the oscillators, there were many instances of issues with aliasing and incorrect waveforms. At first, the phase was not being calculated based on the frequency of the sound, only the time elapsed, which meant it was resetting every second rather than every loop and would cause a popping sound. After that problem was fixed, there was still some popping. This was being caused by the phase being reset to 0 instead of taking away 1 from it meaning that when the phase should have been a small number, it would get set back to 0 instead. When creating the sawtooth wave, playing different notes was causing the minimum and maximum values to move up and down, causing clipping. This was because the calculation was not taking into account the frequency, but instead assuming the frequency was 440Hz, which is the centre frequency that all the other frequencies are based on, as mentioned above.

The initial implementation of the envelope generator was to just divide the maximum level by the sample rate, which didn't allow for an editable time for attack, decay and release. A multiplier was added to this to allow for a variable envelope time. This highlighted another problem, however. The actual time taken to reach the desired value wasn't correct. It was trending towards the value on an exponential curve instead of a linear line as expected. This was because the increment was being calculated every sample, and as the last sample had moved it a fraction closer to the final value already, the newly calculated increment for the next sample would be smaller. In order to stop this from happening, a new system was put into place to store the increment for the given state once meaning that would always be the increment until the expected value is reached. The increment

was made equal to 1 divided by the sample rate, meaning a simple multiplication could be done to achieve the correct time.

Once development was under way, another issue was found when trying to set and change parameters, particularly in the Oscillator and Envelope Generator classes. This caused a substantial halt in development until the cause was identified, as being able to change values during runtime was essential to the project. After about 2 weeks of debugging and getting nowhere, the problem was identified when removing code from the library that wasn't needed. In the end, an oscillator and envelope generator class existed within the library, and they were so tightly integrated into the existing system and used the same variable names for parameters. Because of this, the existing classes were resetting the values immediately after the new classes changed them.

The low pass filter was a particularly troubling element of the project. The design had to undergo multiple iterations in order to function properly. The decided upon strategy was a simple averaging filter, which required a buffer to be constantly stored and accessed every sample. The initial design had a linked list as the storage medium for the buffer, but this was causing the output to become distorted and noisy. It wasn't initially obvious, but after close inspection, it was a side effect of the calculations taking too long to calculate and causing the program to output an incorrect value. Multiple other C++ data types were considered, namely vectors and deques, but a custom buffer class containing a simple array and pointers was chosen. This was the decision because it allows for very swift iteration through the buffer as well as allowing for random access in case another effect were to require it. Even after these more efficient buffers were implemented, there was a limit to the number of the buffers at approximately 50,000 samples, so 44,100 was chosen to match the sample rate of most devices. This caused the delay to stop functioning at a 1 second delay time. This was because the delay buffer was looking back 44,100 samples, which was looping back round to the same sample that had just been calculated, so it was just outputting a raw signal. The buffer was changed to 44.101 to counter this, meaning the maximum time for a delay is exactly 1 second.

Implementing the tremolo effect ended up being much more complicated than expected. The initial plan was to have the LFO be a separate modifier to the volume of the signal, happening after the envelope generator, but this ended up being hard to implement. Since it wasn't functioning like the other effects, it couldn't be easily added to the effect chain without reworking how the chain functioned, and all the other effects were already implemented. The compromise was to place the LFO into the envelope generator instead and multiplying the envelope output by the LFO's value.

Another problematic portion of the coding process came when trying to change the type of oscillator using the GUI. Multiple different button and switch types were used, but to no avail. After much testing and debugging, the solution to this problem was revealed to be the fact that the oscillator was being called directly from the rack, whereas the library seems to have its own voice system set in place. This voicing system was creating a separate oscillator for each voice, meaning

the wrong oscillator's mode was being changed. This was a problem that the rack was designed to eliminate, so this came as a great annoyance.

Adding polyphony proved to be very difficult, as it appeared to be tied into how the library handled input and possibly required more oscillators to be made. The first test was to change how the envelope generator worked. The next test was to make a retrigger function in the envelope generator, as that seemed to be how it is meant to work. In the end, the solution was not found in time, so only one note can be played at a time.

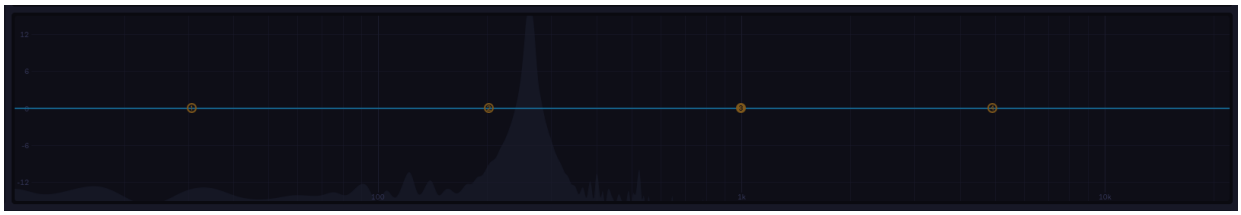
8 Evaluation and Testing

This section will contain self-reflection, as well as some quantitative analysis of waveforms to assess the quality of the finished sound. These were chosen over user testing because this project was more focused on the back-end functionality of the system rather than the end user experience.

8.1 Quantitative Analysis

The focus for this quantitative analysis is testing how close the waves are to the mathematical ideal wave. This is done by looking at a frequency analyser to visualize any erroneous frequencies that appear. For the more complex waves, a reference wave is shown for comparison. An important thing to note about this frequency analyser is that the scale is logarithmic on both axes, so any small differences in volume result in a large audible difference to a listener. This means any artefacts will look a lot worse than they will sound. It also means problems in the higher end of the spectrum will likely be more pronounced and look a lot worse, because higher frequencies can store much more information in them.

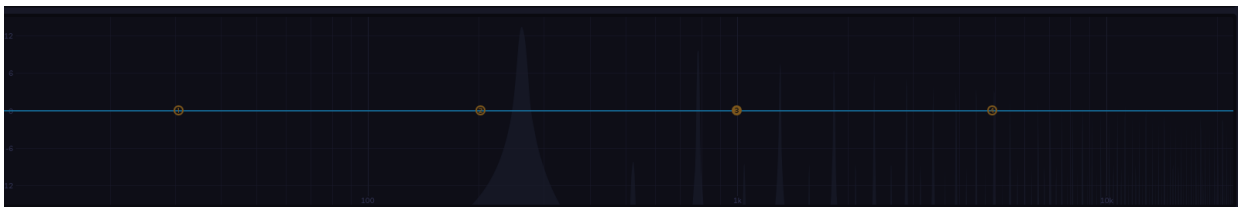
8.1.1 Sine Wave



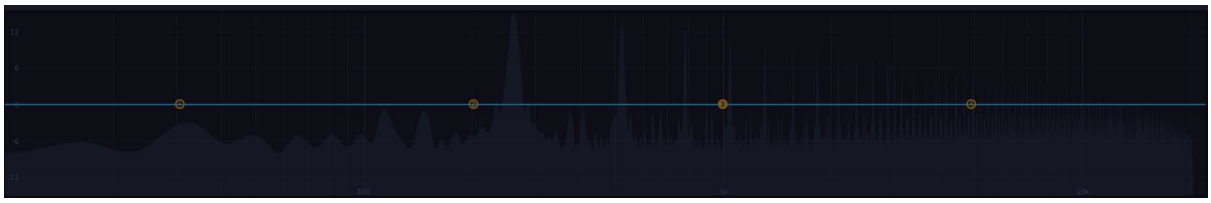
A clean sine wave should appear on a frequency analyser as a single peak with nothing else. As shown above, the sine wave generated by this synth is mostly clean, but it does have some overtones and a small amount of low frequency rumble. This is most likely to do with rounding errors in the sine calculations, so a higher bit representation would minimize this issue.

8.1.2 Square Wave

The square wave, much like the sine wave, has low-end rumble, but it is much more pronounced here. Above is the generated square wave and below is a clean square wave for reference. The tall peaks in the graph are the odd harmonics of the note, which this synth manages to produce, but there are many other noticeable peaks throughout the whole frequency spectrum. Overall this square wave could be a lot better, but it sounds close enough to be identifiable as a square wave.



8.1.3 Sawtooth Wave

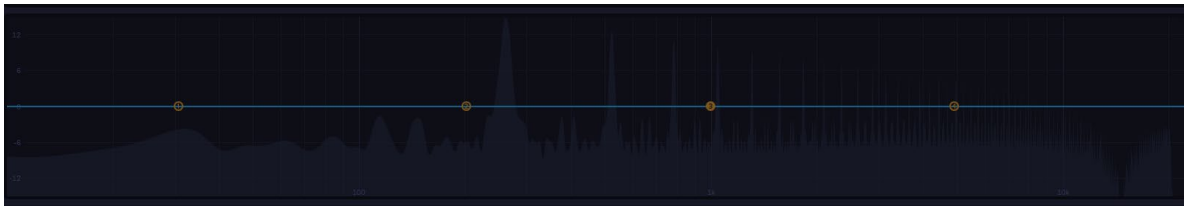


Above is the generated sawtooth wave and below is a clean sawtooth. Similar to the other two waves, there is a noticeable low frequency rumble and some unwanted overtones. However, there are a lot fewer than in the square wave. Overall, this one is in between the sine and square in terms of accuracy.

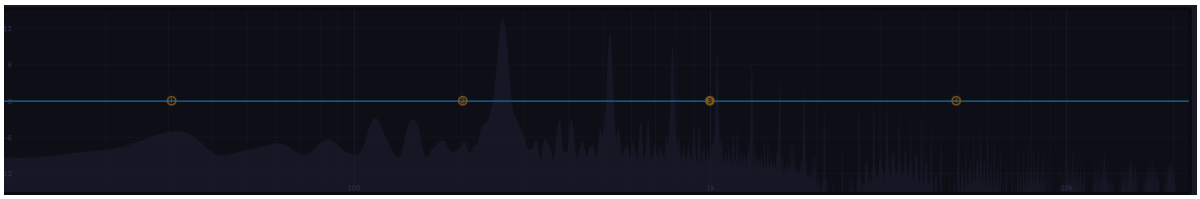
8.1.4 White Noise

The white noise signal generated is completely accurate, as it should average out to every frequency as being approximately the same volume, which is shown here. The small peaks and troughs are due to the nature of how random number generation works.

8.1.5 LP Filter



Shown above is the sawtooth wave with the low pass filter set to 10,000Hz. As expected, the volume does start dropping off at 10,000Hz, but not very steeply. Once it reaches about 15,000Hz, the volume reaches zero and bounces back up again until 20,000Hz. This is a side effect of having a simple averaging filter instead of a more thorough filter. It is much more noticeable in the example below where the filter frequency is set much lower. Despite the aliasing, it does reduce the volume of the higher frequencies by at least 6dB, which halves the perceived volume meaning there is a noticeable difference in tone to the sound as the low pass filter is added.



8.2 Self-Reflection

Due to the creative nature of this system, there are many issues with this synthesizer that aren't easily quantifiable, so in this section I will discuss the issues I have with the system as a potential user.

Firstly, the lack of polyphony is a massive downside, as only being able to play one note at a time is very limiting. On a similar note, the synth also has plenty of artefacts when you play two notes in very quick succession. Another issue with the system is how easy it is to cause clipping, as the oscillator on its own has the ability to reach the maximum values before any effects are even applied, so effects often push it past the maximum. It is also very limiting in features compared to other synthesizers. Only one oscillator is available, which severely limits the number of sounds that can be created. It also has no stereo functionality whatsoever, meaning there is no feeling of space to the sound. As a producer I also dislike that the more complex waves aren't cleaner, as it would make them harder to fit into a song without clashing.

9 Conclusions

Overall, I thought this project was very successful. Despite many setbacks and delays, I believe the finished product achieves what I aimed to achieve when I proposed this project. My initial requirements and expectations were very lofty, which meant many things were cut from the final program. Nonetheless, I managed to achieve all of my essential requirements and some of the extended goals such as extra effects and modulation in the tremolo effect. The final sounds that the synthesizer is generating are good enough to be usable by musicians, but it is unfortunate that some other aspects of the project stop this from being usable in some scenarios, such as the lack of polyphony. However, this project was successful at achieving my learning objectives. I managed to use a fairly lightweight and easy to use C++ library to implement a simple synthesizer. It helped me understand DSP better and gave me more insight into what goes into making synths. The design allows for further additions in the future should I decide to keep working on it.

The project was definitely at its most productive while I did not have to interface with the library, with my own rack structure being very quick to implement. By contrast, having to modify existing classes such as the IPlugInstrument class frequently caused unexpected behaviour resulting in lots of debugging.

iPlug as a library is very well thought out for the most part, but it definitely had its problems. The documentation was very poor, with large chunks having no documentation at all. This meant that in order to learn how it worked, I had to look up tutorials and examples instead. Despite that drawback, it ended up being a very competent library, with a large feature set built in. It meant that instead of worrying about MIDI input, GUI or audio drivers, I could focus on the important points of the project. Overall, I was very happy with my experience with iPlug.

If I were to do the project again, I would have focused the original requirements more and been clearer with what I was trying to achieve. I would also be more generous with coding time, as many unforeseen circumstances both inside and outside of the project had a large impact on what was possible in the time allotted. Adding more theory into the initial research would have been helpful, in case I wanted to, for example, attempt a simple reverb technique or do more complex filters that require Fourier analysis.

10 Further Work

Due to the complex and modular nature of synthesizers, there are countless possible additions that could be done as further work. Some of these additions were considered when designing the current solution and so would be relatively easy to implement, whereas some were too complex to even consider as a possible feature.

The most obvious feature is polyphony, as that would allow chords to be played, exponentially increasing the potential sounds. This appeared to be tied into how the library handled input and possibly required more oscillators to be made, but the final solution was not found in time.

The effects chain in the rack was designed to be able to arbitrarily add extra effects onto the chain whilst only needing to add a line of code to add it to the chain. This meant all manner of different audio effects could be added. A compressor or limiter could edit the dynamics of the sound; a chorus, flanger or phaser could make the sound morph over time, creating a sound with movement; or a reverb effect could make it sound as though it is being played in a real environment rather than a computer.

The rack structure also allows multiple oscillators to be made without major changes to code. This would add interesting dynamics where multiple different waves could be playing at different frequencies at the same time to make a fuller sound.

The tremolo in the envelope generator demonstrated the ability to modulate variables over time. With more time, an LFO or envelope could have been attached to nearly any other value in the synth to create interesting effects. Particularly, modulating the pitch of the main oscillator to create vibrato or an envelope attached to the filter cut-off frequency, which is a staple of subtractive synthesis.

Much like modulating with an envelope or LFO, a MIDI control could be attached to a parameter, meaning a hardware control could be used to modify values instead of the software controls in the GUI. This is a very desirable feature of a synthesizer, particularly for live performances.

Being able to save a preset was not considered in the design of this solution, but it would be obtainable if the parameters were all stored in a serializable object. It would add another layer to the project, as file saving and loading would need to be implemented.

Another addition that would require a significant change in the codebase is editable or custom waveforms. In order to achieve that, the oscillator would need to manually store all the samples of the custom wave in an array. Whilst a time consuming feature, it would arguably be the most effective when it comes to creating the widest range of sounds.

Adding stereo panning to the synth would create another dimension for the sound to sit in. It wouldn't be too difficult in theory, but the design for this project only works in mono, so this would require some reworking of existing code rather than just adding code on.

Further work could also be done to the UI, as the final UI for this project was very plain and not perfectly laid out. Adding more structure and colour to the UI would make for a much more pleasant user experience.

A very ambitious goal to add to this project would be dynamic oscillator addition, meaning an arbitrary number of oscillators could be loaded at once, each with their own effects chain. This would potentially require a different library or the code to be rewritten from scratch, as iPlug may have too much overhead and thus be unable to handle such an intensive and demanding feature.

Another large goal for the system would be to create a more accurate filter. There is also potential to create other types of filter, because a low pass filter is only one of many different types of filter. The notable common examples are a high pass filter, a notch filter to remove a specific frequency, or a comb filter which removes a much more complex series of frequencies. These would all be attached to the rack structure easily, but the implementation of the actual filters would be very time consuming and difficult.

11 References

1. Margaret Rouse. *MIDI (Musical Instrument Digital Interface)* Available from: https://whatis.techtarget.com/definition/MIDI-Musical-Instrument-Digital-Interface [Accessed 7th Nov 2018]
2. Analog Devices. *A Beginner's Guide to Digital Signal Processing (DSP)* Available from: https://www.analog.com/en/design-center/landing-pages/001/beginners-guide-to-dsp.html [Accessed 7th Nov 2018]
3. TechTerms. *DAW* Available from: https://techterms.com/definition/daw [Accessed 7th Nov 2018]
4. Joe Shambro. *VST Plug-ins: What They Are and How to Use Them* Available from: https://www.thoughtco.com/what-are-vst-plug-ins-1817748 [Accessed 7th Nov 2018]
5. BCS. *BCS Code of Conduct* Available from: https://www.bcs.org/category/6030 [Accessed 7th Nov 2018]
6. Lech Szczepaniak. *How Do Synths Work? An Introduction to Audio Synthesis* Available from: https://reverb.com/news/how-do-synths-work [Accessed 7th Nov 2018]
7. Gordon Reid. *An Introduction To Additive Synthesis* Available from: https://www.soundonsound.com/techniques/introduction-additive-synthesis [Accessed 7th Nov 2018]
8. Scott Rise. *FM Synthesis* Available from: http://synthesizeracademy.com/fm-synthesis/ [Accessed 7th Nov 2018]
9. Native Instruments. *Massive* Available from: https://www.native-instruments.com/en/products/komplete/synths/massive/ [Accessed 7th Nov 2018]
10. Xfer Records. *Serum* Available from: https://www.xferrecords.com/products/serum [Accessed 7th Nov 2018]
11. Hermann Seib. *VSTHost* Available from: http://www.hermannseib.com/english/vsthost.htm [Accessed 7th Nov 2018]
12. juce.com. *JUCE* Available from: https://juce.com/ [Accessed 24 May 2020]
13. Oli Larkin. *WDL / IPlug (Oli Larkin Edition)* Available from: https://github.com/olilarkin/wdl-ol [Accessed 7th Nov 2018]
14. Nerd Audio. *Envelope Generator (EG)* Available from: https://nerdaudio.com/blogs/news/envelope-generator-eg [Accessed 24 May 2020]

12 APPENDICES

12.1 APPENDIX A: WEEKLY LOGS

12TH NOV

Started design

Basic UI mock-up created

Started API researching

19TH NOV

UI mock-up redesigned

Continued API research

26TH NOV

Group Meeting – 26th: Asked about presets. Serialization was suggested.

Started class diagram

14TH JAN

No progress

21ST JAN

No progress

28TH JAN

Coding Phase started. Modified existing Oscillator class to include noise generator.

Individual Meeting – 29th: Discussed different API versions. Decided on sticking with older version.

TEMPORARY WITHDRAWAL

20TH JAN

Revisited code to re-familiarize myself.

Individual Meeting – 22nd: Discussed progress that had been made over the break. No major work done. Asked for the current prototype.

27TH JAN

Integrated library elements. Tagged sections to replace.

Individual Meeting – 29th: Demonstrated early MVP. Discussed advice on oscillators.

3RD FEB

Wrote sine oscillator and planned rack structure.

10TH FEB

Started envelope generator. Wrote more waveforms.

17TH FEB

Encountered UI parameter problems. Wrote distortion.

Group Meeting – 21st: Discussion about poster presentation.

24TH FEB

Fixed UI Issues. No other progress.

2ND MAR

Created effects chain. Added delay. Blocked UI

Individual Meeting – 5th: Got feedback on poster. Asked about possible filter methods.

9TH MAR

Started low pass filter. Encountered speed bottleneck.

Individual Meeting – 12th: Asked about speed bottleneck. Potential solutions were suggested.

16TH MAR

Tested different LP methods. Array pointer was the choice.

Individual Meeting – 20th: Enquired about shifting frequency of LP filter.

23RD MAR

LP modulation added. Tremolo added. Problems changing waveform in UI

External Issues

20TH APR

Fixed UI issues. Finished up UI. Implementation done.

Individual Meeting – 24th: Talked about structure for the report.

Final meetings for report progress on 11th and 18th May

12.2 Appendix B - Proposal Document

12.2.1 Aims and Objectives

Primary Objectives:

- Oscillator
- Envelope Generator
- Amplifier
- MIDI Control
- Simple Filters
- Effects Chain
- Delay
- Functional GUI

Extensions:

- Compressor/Limiter
- Distortion
- Complex Filters
- Chorus
- Flanger/Phaser
- Reverb
- Presets/Saving
- Tremolo
- Bitcrusher
- LFOs
- Automation
- Gates
- Stereo effects
- Polyphony
- Multiple Oscillators
- Editable Waveforms

Fall-back Strategies:

Stick to the points that result in a working synthesizer:

- Oscillator
- Amplifier
- MIDI Control
- Functional GUI

12.2.2 Relevance

This project has a large focus on coding in C++, both interfacing with existing APIs and engineering code from scratch. It will also require extensive mathematical knowledge to generate and manipulate the audio signal. It will require a solid design, particularly with the GUI, and will utilise many of the structures and functions provided by an OOP language. Optimisation and refactoring will also be integral to this project in order to minimise latency and CPU usage.

12.2.3 Resources Required

C++ IDE (Visual Studio)

MIDI Controller

12.2.4 Timetable

	Monday	Tuesday	Wednesday	Thursday	Friday
9AM					
10AM					
11AM					
12PM					
1PM					
2PM					
3PM					
4PM					
5PM					
6PM					
7PM					
8PM					

3D Animation

Video Production Techniques

Human-Computer Interaction

Project