Sheffield Hallam University

College of Business, Technology and Engineering

Department of Computing

Project (Technical Computing)

[55-604708]

2020/21

Author:	**Ruari Molyneux**
Student ID:	**28010709**
Year Submitted:	**2021**
Supervisor:	**Christopher Bates**
Second Marker:	**Dave Thormley**
Degree Course:	**Computer Science**
Title of Project:	**Developing Virtual Studio Technology Plugins**

Confidentiality Required?
YES / NO

I give permission to make my project report, video and deliverable accessible to staff and students on the Project (Technical Computing) module at Sheffield Hallam University.
YES / NO

Developing Virtual Studio Technology Plugins
By Ruari Molyneux
Marker: Please mark this work for content and ideas and not for accurate spelling and punctuation unless this is a requirement of assessment.

Acknowledgements

I would like to thank Chris Bates for his support and invaluable insight during the project.

Abstract

This project contains a suite of plugin implementations on the JUCE audio app development framework. This project aims to provide understanding of the algorithms and technologies involved in the implementation of an audio application. The goal of this project is to implement a synthesizer and sampler Virtual Studio Technology Plugin that can be used to write pieces of music in a digital environment. First some background information about the components of a sampler and synthesizer instruments will be provided. After this a selection of audio app development frameworks and implementation on these frameworks will be discussed. A design will then be created for implementing the components that have been identified in the research section. After this a discussion into the implementation of this design on the selected framework will take place. During the development process an iterative and incremental methodology will be used.

Contents

1. Introduction

1.1 Project overview

Nowadays, music is pronominally produced using software instruments instead of physical hardware. This is due to the flexibility and power of using a software approach. As a music enthusiast who has used software instruments to produce music, I was curious to bring my software engineering knowledge to this area and develop one myself. Also, since this was an area that is not covered in my degree it would extend my knowledge of software engineering in a music technology context. This project will offer insight into how virtual studio technology, VST, plugins work and provide a breakdown of their constituent parts in the context of a synthesizer and sampler plugin built using an audio app development framework. To give a more in depth understanding of how these plugins work information will be provided about digital signal processing and synthesis fundamentals.

1.2 Aims and objectives.

The aims of this project are to

- Research digital synthesis methods.
- Research synthesizer signal paths and architecture.
- Identify the current methods for Virtual Studio Technology development
- Identify a set of libraries and a development platform for Virtual Studio Technology development
- Implement a suite of instruments that can be used inside a digital audio workspace, DAW.

This project would be considered a success if the set of instruments created are playable and the sound, they create is able to be manipulated and output to a DAW. This will be evaluated with an exploratory testing approach that involves trying to break the plugins and an acceptance approach that involves using the plugins for their intended purpose.

2. Research

Due to the nature of this project research will cover Digital signal processing as well as computer science concepts. This is used to help give context to the software implementation.

2.1 Introduction to digital signal processing

The research in the section is guided by The Scientist and Engineer's Guide to Digital Signal Processing book (Steven W. Smith, 1999).

The term digital signal processing in the context of VST's refers to the manipulation of audio signal data through use of mathematical functions. Through using these functions, traditional synthesis methods can be implemented in software.

2.2 Synthesizers

Fig 1. Image of a Roland Juno-106 analog synthesizer.

Traditionally a synthesizer would have been a piece of hardware that uses a voltage-controlled oscillator, VCO, to generate an audio signal. A voltage is input into the oscillator and depending on this input a different frequency would be output. The voltage that is input is generally controlled by which key is pressed on the keyboard. In a software implementation of a synthesizer the oscillators are algorithms, and the input is a midi note that will be converted to a frequency. A different waveform can be generated depending on the algorithm that is implemented.

2.2.1 Digital Oscillators

An Oscillators is the component in a synthesizer that handles tones generation. Oscillators can produce a variety of waveforms depending on the design of the generator function. The function

takes a pitch that has been selected from the keyboard and generates a waveform for the selected pitch. In a traditional synthesizer a VCO would be used to generate tones, although over time the oscillators in these synthesizers would go out of tune and need to be retuned by the user. The digital oscillator was developed as a response to the tuning instability of VCOs. Tuning stability is one of the main advantages of using a Digital Oscillator.

Some common examples of waveforms are shown below.

2.2.2 Sinewave

A smooth sounding wave based on the sine function. Like its appearance it has a smooth sound. Also, it has a strong emphasis on its base frequency in the sound and should have no overtones if it is pure.
Generally, this wave is used to make deep bass sounds and strong lead sounds.

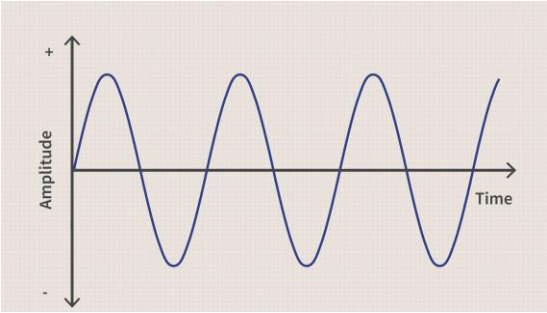

Fig 2. Image of a sine wave

2.2.3 Sawtooth wave

This wave is composed of a sequence of ramps.
Generally used to make aggressive leads.

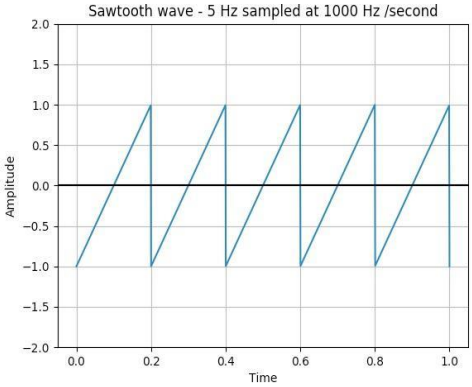

Fig 3. Image of a sawtooth wave

2.2.4 Square wave

This waveform is composed of a binary sequence.

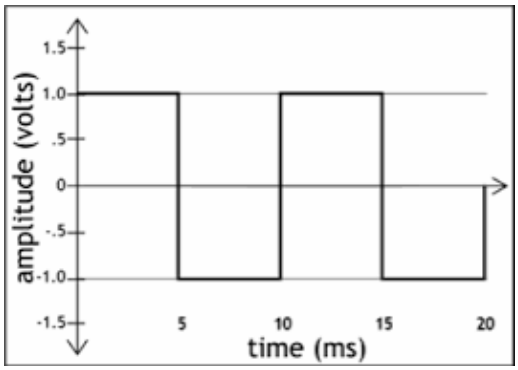

Fig 4. Image of a square wave

2.2.5 Synthesizer Voice

The voice of a synthesizer contains all the components (signal path) necessary to create a sound for a single note. The number of voices a synthesizer has is equal to the number of sounds that can be played in parallel.

2.2.6 Signal paths

For a synthesizer to affect the signal it produces from its oscillators the signal must pass through various components. The combination of these components is called a signal path. Common components in the signal path include.

2.2.7 Mixer

Component of a synthesizer which controls the level of each individual sound source.

Fig 5. Image of a Roland SH-101 mixer

2.2.8 Envelope

An envelope is used to modulate how a sound changes over time. This can be applied to a variety of aspects of the sound like the amplitude, the filter or the pitch. For this example, how they affect the amplitude of the sound will be used.
Generally, an envelope is described using the parameters.
Attack - The time it takes the sound the reach its highest amplitude from silence.
Decay - The time it takes the sound to drop from the peak to a sustain level.
Sustain - The main level of the sound after the attack and decay duration have passed.
Release - The amount of time it takes the sound to return to silence from the sustain level after the key has been released.

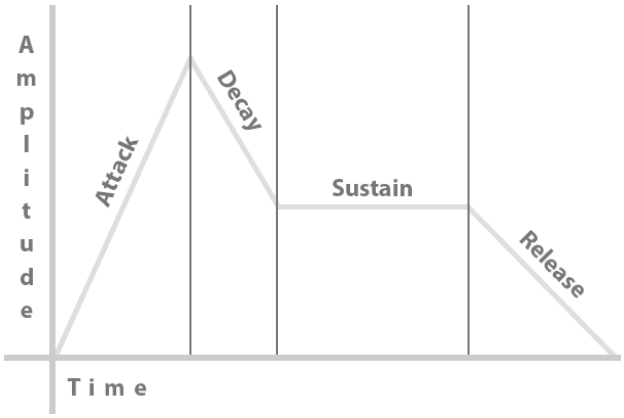

Fig 6. Image of an envelope curve

2.2.9 Filters

A filter is used to remove certain constituent frequencies from a sound. A cutoff value is used to denote the points at which these frequencies start to get removed. Also, some filter designs may include a resonance level which amplifies the sound at the cutoff frequency depending on its level.

Examples of filter types include.
Low Pass filter - removes frequencies above the cutoff frequency.

High Pass filter - removes frequencies below the cutoff frequency.

Bandpass filter - removes frequencies outside a certain frequency range.

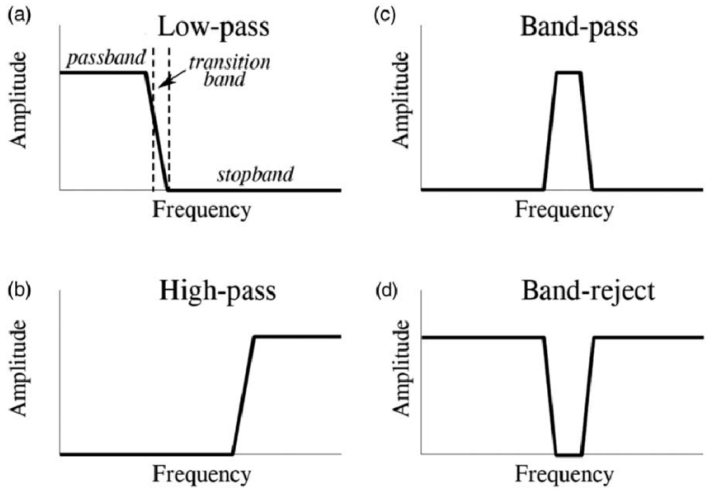

Fig 7. Image of common filter types.

2.2.10 Low frequency oscillators - LFO

Used to create long oscillations to modulate the sounds as whole and add movement to the sound. The depth of an LFO is used to describe the range of frequencies that it should oscillate through.

2.2.11 Arpeggiators - generally shortened to ARP.

This type of modulation is different to the aforementioned types as it does not directly modulate the signal of the sound. Instead, it converts the keys of a chord into a sequence of notes. Depending on the configuration of the arpeggiator it may play keys in ascending, descending or a random order.

2.3. Samplers

Fig 8. Image of an Akai S950 sampler

Fig 9. Image of a Mellotron sampler

In addition to synthesizers another form of electronic instrument is a sampler. In this instrument design a sample is used as the sound source instead of a synthesized waveform. Besides the sound source, their signal paths will be very similar to a synthesizer, usually including at least a filter, an envelope and a low frequency oscillator. Some may include more sample specific effects like looping, changing the start point or a way of pitching the sound.

2.4. Audio application research

2.4.1 Musical Instrument Digital Interface (Midi)

Midi data is created by a midi sequencer and used to describe the start of a note, its pitch, length, volume and timing. This information can then be used to control an electronic or software instrument. In the context of this project a DAW is used as a midi sequencer to create and pass midi data to a plugin.

2.4.2 Audio sampling/Sample rates

The process of audio sampling in the context of digital to analog conversion is the process of converting a discrete set of audio samples into a continuous audio signal via approximation. A digital to analog converter, DAC, is a piece of a hardware that is used to perform this task. A sample rate is used to determine how often the DAC should process the set of samples to a continuous signal. In order to represent the full range of human audio perception a sample rate of 20khz or higher must be used. Although generally a sample rate of 44.1 kHz is used to allow for more headroom. As the sample rate drops lower the quality of the sound will reduce since more information is being left out of the conversion to an analog signal. In the context of a VST audio data is output to the DAW as a sequence of samples which will then be passed on to a DAC for output to the speakers.

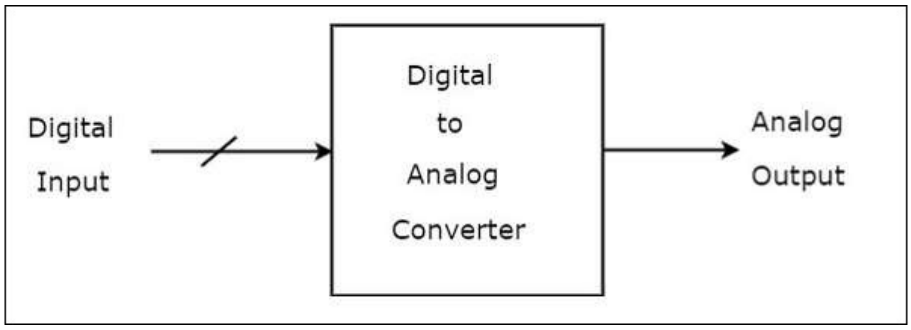

Fig 10. Image of DAC functionality

2.4.3 What is a Digital Audio Workspace?

The majority of contemporary music production is based around an application called a digital audio workspace, DAW, an environment for digitally recording and editing audio. These environments are frequently used to produce and record musical performances much like a traditional recording studio. In this environment VST plugins can be loaded and used to create music. The DAW acts a clock for everything that is loaded so multiple different plugins can be used in conjunction.

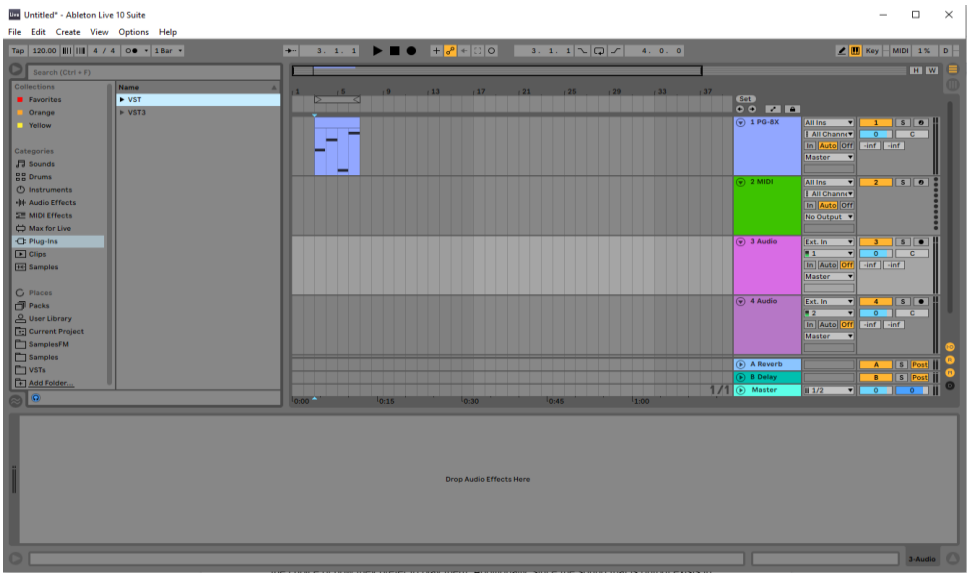

Fig 11. Image of a DAW GUI with a plugin loaded and midi data created.

2.4.4 What is an instrument plugin?

One implementation of a software instrument is a Virtual Studio Technology (VST) instrument plugin. Throughout this report this will be referred to as a VST. This is an application that is loaded into a DAW. Once loaded the DAW acts as a host that controls what midi data is input to the plugin. The plugin then uses this data as input to digital signal processing algorithms that dictate the audio data that is output back to the host environment. This disconnection from how the data that is input to the plugin is created is what makes them so flexible. Additionally, since the sound that is output exists in the DAW environment, sound effects plugins can be applied to sculpt the final sound. Also, through the DAW automation be applied to components like the filter cutoff.

Fundamentally, these plugins perform the same role as a physical instrument, like a synthesizer, would in a music studio. Although, they provide some advantages over traditional instruments. Some of these include.

- Scalability. Software instruments are not limited by physical space requirements.
    - Can have many more voice/signal paths than hardware synth.
- Reliability. Software instruments cannot regularly break down.
- Flexible interface provided by MIDI.
- Convenience. A laptop is easier to transport than a Grand Piano.
- Extensible.

Some **limitations** include.

- Performance limited to host environments.
- Playability is limited without additional hardware.

Fig 12. OB-Xd plugin GUI

2.5. Software framework research

The table below outlines some of the software instruments that are currently in development.

	Dexed	OB-Xd	Surge
Programming language	C++	C/C++	C/C++
Synth engine	music-synthesizer-for-android(google)	Bespoke	Bespoke
Framework	JUCE	JUCE	VSTGUI

Most software instruments that are currently in development are written in C/C++. This is due to the potential performance gains that well written and optimized C++ can yield. Optimizing for performance is very important in audio applications since they usually constantly process audio at 44.1khz. Also, C++ provides ample tools for concurrency which is generally used in these applications due to the complexity of the signal path and its integration with a front end in real time.

2.5.1 Technology comparisons

Below is a comparison table which has been used to determine which technology has the appropriate characteristics for creating a complete set of software instruments.

	Python with NumPy and SoundDevice libraries	**C++ on the Juce framework with Maxi Library**	**Pure data/Max/Max Signal Processing**	**Sonic Pi/Tidal Cycles & SuperCollider/ChucK**	**Faust(Functional audio stream)**
GUI options	A python GUI framework would have to be added as a dependency.	Built in GUI library that is integrated efficiently with the backend	Built in GUI options that can be attached to elements in the signal chain. Limited option for developing custom GUI components	This approach is based around a command line, so everything is controlled via text input.	Primary used as a backend Sample level DSP language although it could be connected to a web front end via Faust web.
Level of abstraction from DSP fundamental algorithms.	Very close to fundamental. Requires writing most of the synthesis and effects processing from scratch.	Moderate level of abstractions. Most synthesis algorithms will not need to be written but it is possibly to write a bespoke solution if required.	Highly abstracted from the actual synthesis. Although still maintaining access to the audio data signal path.	Highly abstracted from lower-level synthesis. Audio generation options are limited to the ones provided by the framework.	Moderately low level with built in functions allowing access to DSP algorithms that are required to create bespoke filters and other complex real time audio components from scratch.
DSP algorithms provided	no	yes	yes	yes	yes
Project can be compiled to a VST file	no	yes.	no	no	Not inherently, the faust VST library could be used to do this.
Type of programming language	Object oriented	Object oriented	Visual	Live coding	Functional

2.5.2 Technology requirements

The correct technology for the job should provide a sufficient suit of pre-written digital signal processing, DSP, algorithms for use throughout the project. It should also provide a quick and customizable way to connect GUI components to the data being sent to these algorithms. The technology should provide an optimized way to work with both synthesized sounds and prerecorded sounds to allow for the development of a synthesizer and a sampler with potential to be scaled up to a more complex solution. Additionally, the correct technology should provide the capability to compile the project to a VST architecture allowing the plugin to be built and used in any compatible DAW.

2.5.3 JUCE framework

JUCE is an audio application development framework built in C++ and is an industry standard for audio application development. A benefit of using this framework is the documentation, one developer noted “the JUCE docs often give very clear explanations of the available classes and functions, as well as useful code examples” (Chowdhury, 2020). This along with their useful prebuilt implementations make this library a good choice. One of the key features it provides is a wrapper for VST applications. This allows an audio application to interface with a DAW host meaning the plugin can be loaded with other plugins and reap the benefits of DAW environment. Additionally, JUCE provides a suite of GUI components and functionally to map these components to DSP algorithms. An example of this could be a slider that is mapped to an algorithm that affects the volume of the audio signal. JUCE also includes a debug environment called the JUCE Plugin host. This can be used to mimic a DAW environment when debugging the VST, while still allowing the use of breakpoints and the call stack.

2.5.4 Maximilian library

The Maximilian library is an audio synthesis and signal processing library written in C++.
It provides pre-built implementation for lower-level DSP algorithms like the synthesizer components mentioned in the above background information. It seemed to make the most sense to us a library instead of implementing this functionality myself….

2.6. Methodology

When picking a development approach, it is important to take the characteristics of the project. This project's development can be broken down into implementing each of the components that have been displayed in the research section. Due to this an iterative and incremental methodology made the most sense.

An iterative design and incremental methodology is an approach to software development in which a project is broken down into a series of cycles where each iteration is informed by the previous cycle. In each of these cycles the full development process is completed including, first planning the iteration through creating requirements, then designing and implementing the iteration followed by testing and finally an evaluation of the pros and cons of the iteration. Each phase of a cycle can also be broken down into iteration if it is necessary for the task. A benefit of this methodology is that each iteration “is used as the learning process with the objective to correct the way the next iteration is done or modify the scope” (Farcic, 2014). A prototyping approach has been used in this project to allow for rapid development and refinement of the implementation.

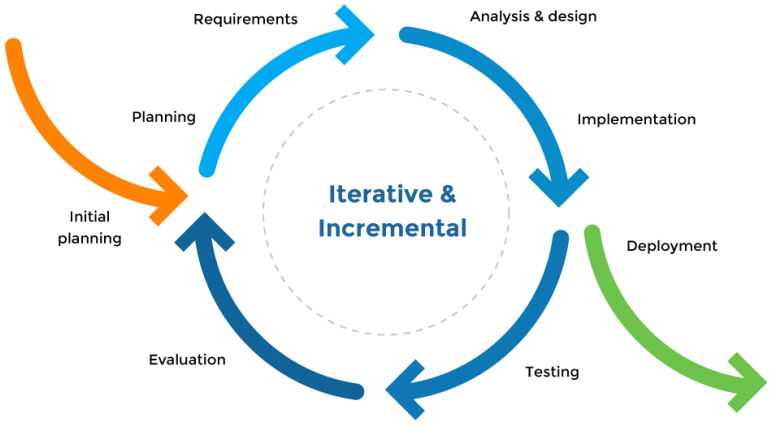

Fig 13. Image of the iterative and incremental model

2.7. Tools

2.7.1 Trello

Trello is a kanban style project management website that was used to organize and keep track of tasks that needed to be completed throughout the implementation process.

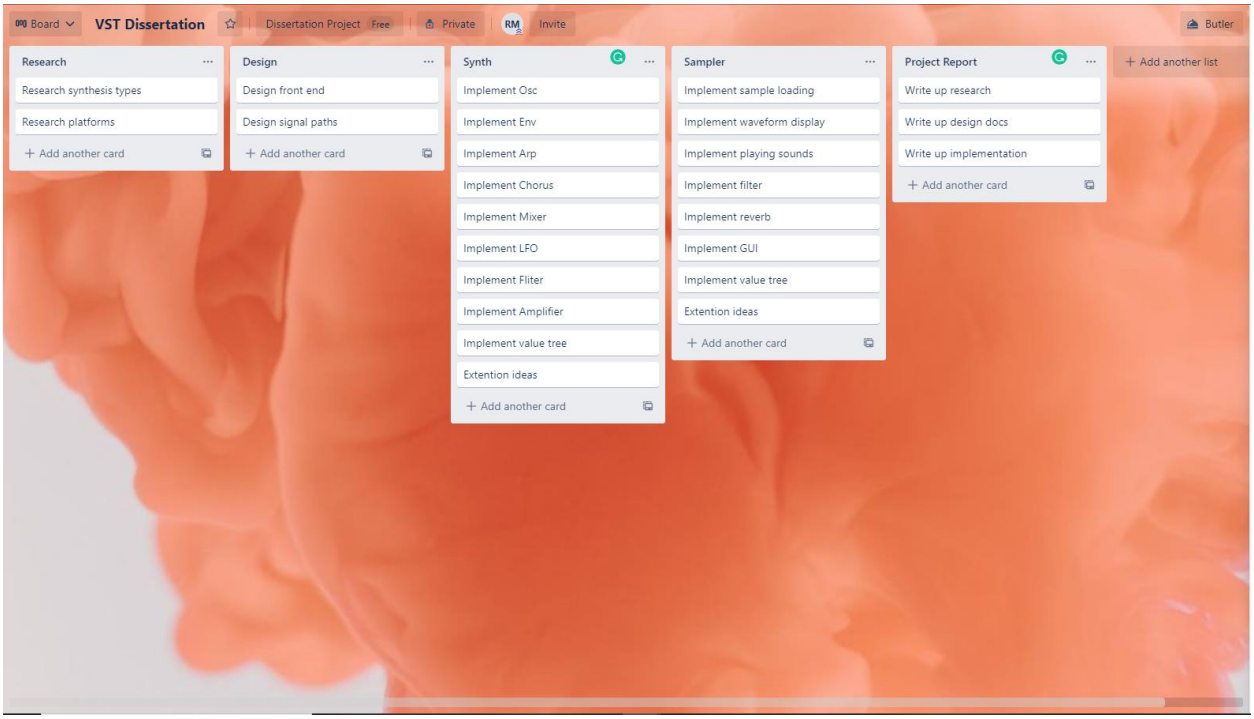

Fig 14. Image of the Trello board

2.7.2 GitHub

GitHub was the version control system used throughout the implementation.
By using this system, it was easy to keep track of previous implementation while prototyping the project.

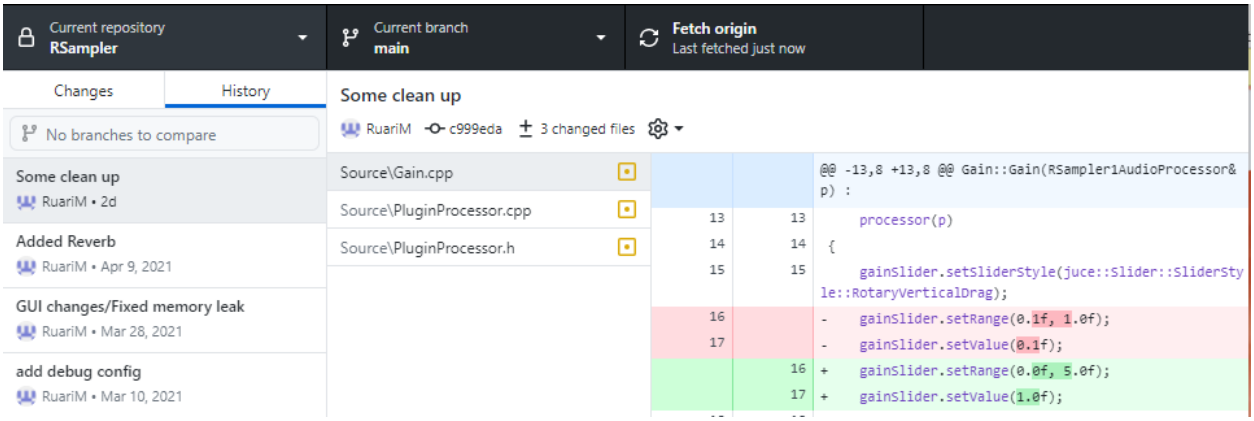

Fig 1.5 Image of sampler GitHub repository

3. Design

3.1 Implementation requirements

Both implementations must have

- 8 voices to play sounds with.
- A GUI to manipulate the sound that is produced.

Synthesizer

The synthesizer implementation must.

- Include components in the signal path that.
    - Generate the following waveforms with oscillators.
        - Square wave
        - Saw wave.
        - Square wave sub oscillator two octaves lower is pitch than the other two.
    - Control the level of each oscillator with a mixer.
    - Filter the audio signal with.
        - Low pass filter
        - High pass filter
    - Apply an envelope to the audio signal with the following ramps.
        - Attack
        - Decay
        - Sustain
        - Release
    - Apply a chorus effect to the audio signal.
    - Modulate the oscillators with a low frequency oscillator.
    - Control output volume.

Sampler

The sampler implementation must

- Accept files as valid sound sources. Valid file types include
    - .WAV
    - .MP3
    - .AIF
- Include components in the signal path that.
    - Apply an envelope to the sound source.
    - Filter the sample with.
        - Low pass filter.
        - High pass filter.
        - Band pass filter.
    - Apply a reverb effect to the sound source.
- Visualize the loaded sounds waveform.

Both implementations must be able to.
be loaded into a DAW host environment.

- Midi data from the host must be accepted by the plugin as input.
- Audio generated by the plugin must be accepted as output to the host.

3.2 GUI design direction

The design for the GUI is inspired by traditional hardware synthesizers. Since this is an existing design paradigm it made the most sense as users will already be accustomed to the expected functionality. Linear sliders have been used as the predominant value input method. The use of rotary sliders has been kept to a minimum as their functionality does not translate very well to on screen usage.

Fig 16./ Fig 17. Left, Arp Odyssey linear slider design. Right, Mini Moog rotary knob design.

Fig 18. Examples of JUCE Slider components used.

Mode selection is handled by either a button or a drop-down menu. If the number of selection options is two a button is used otherwise drop-down selection is used.

Fig 19. GUI component examples

3.3 Synthesizer design

3.3.1 GUI design

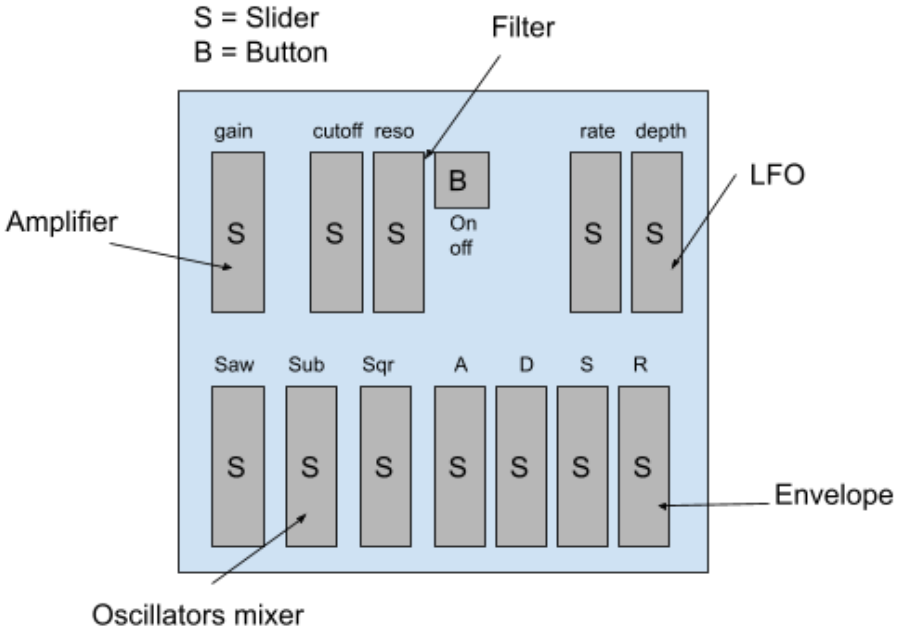

Fig 20. Synthesizer GUI Mock up

This is a GUI design for the synthesizer based off the general design ideas laid out above.

3.3.2 Signal path

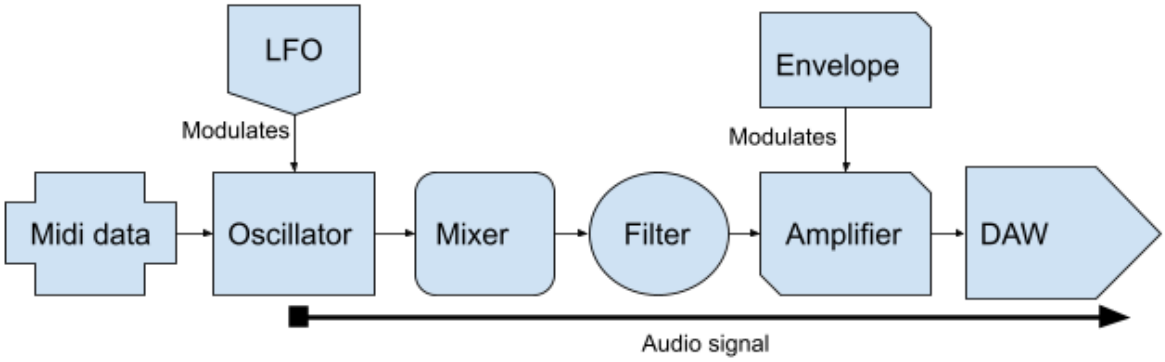

Fig 21. Synthesizer signal path used in RSynth

These are the must have elements of the synthesizer signal path.

3.4 Sampler design

3.4.1 Multi sound sampler design

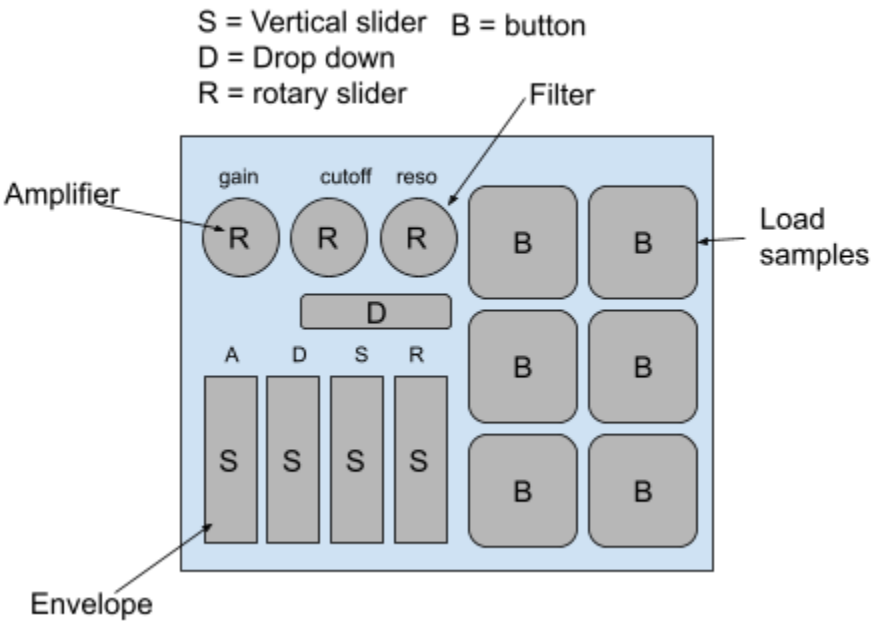

Fig 22. Multi sound sampler GUI mock up.

The initial design for the sampler was based around a drum pad sampler style in which each key plays a different sound. By clicking on a button, you would be prompted to select a file to load for that key. I decided this design was not the direction I wanted the sampler to go in as most of my research was based around single sound source samplers. Although, it was interesting to try this idea at first.

3.4.2 Single sound sampler design

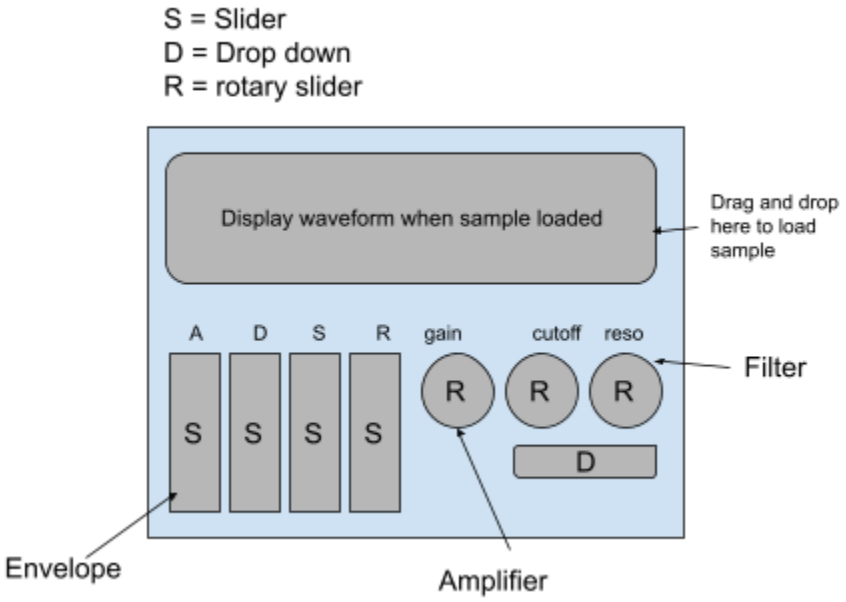

Fig 23. Drag and drop sampler GUI mock up.

A single sample drag and drop design was the final idea. This kept to GUI simple and uncluttered while performing the same functionality. Also, waveform display is add for improved look and feel.

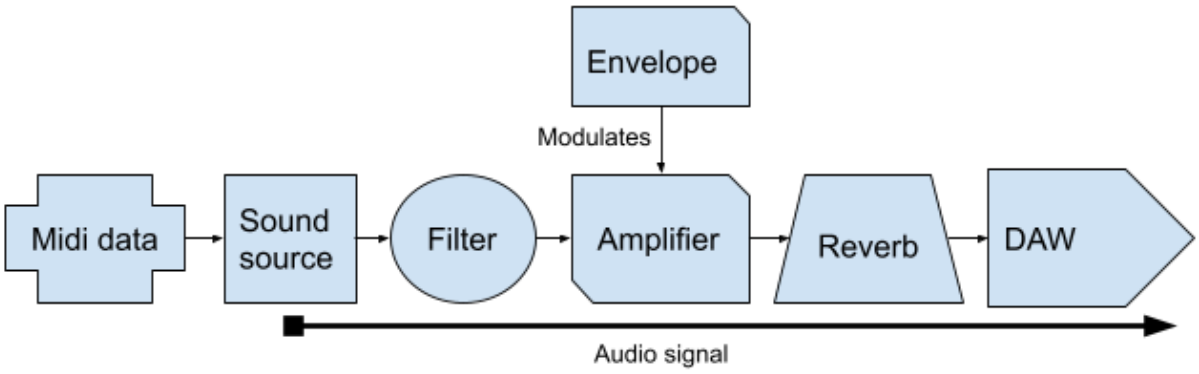

Fig 24. Architecture signal path in RSampler

These are the must have elements of the samplers' signal path

4. Implementation

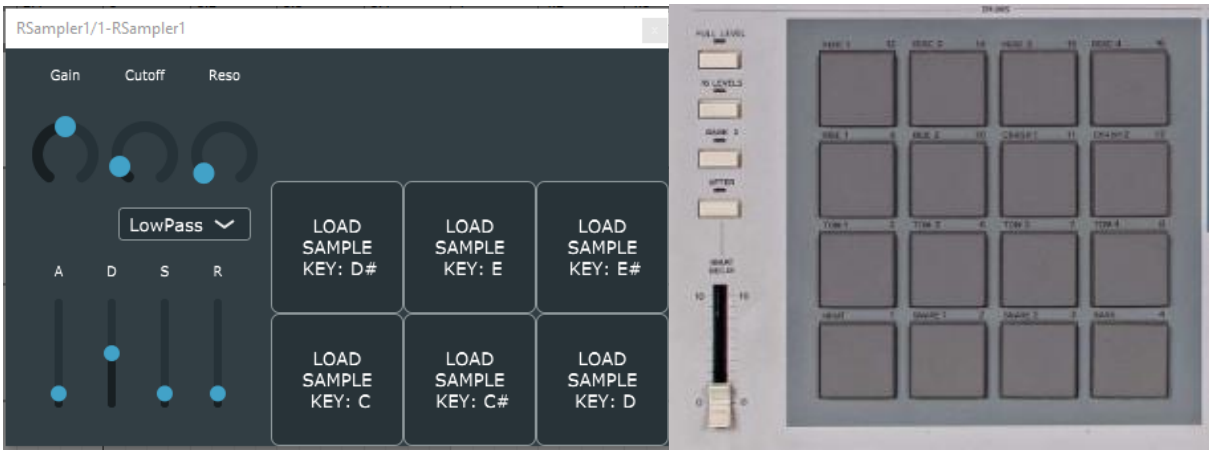

Fig 25. Left, first sampler prototype. Fig 26.Right, MPC60 drum pad GUI inspiration.

4.1 Overview of development

In this implementation section I will explain how JUCE should be used to create a synthesizer and a sampler plugin. The design of each of these implementations is based on the requirements, GUI mockups and signal paths set out in the design section of this report.

4.2 The development environment

4.2.1 Setting up the project using Projucer

First, in order to set up a plugin development environment using the JUCE library their Projucer application must be used. This application handles the creation and dependency management of the project. Since a plugin was being developed the Plug-in basic project template was selected. At this stage, the correct library modules were selected from the modules list. All essential modules are pre-selected and the only extra module I decided to use for both implementations was the DSP module. Visual Studio 2019 was selected as the exporter, so the plugin can be correctly compiled in my development environment. When the project is created, a .jucer file is generated to describe how the project should be compiled and the location of the dependencies. The classes that are created at this stage are the framework that the plugin must be developed inside. The PluginProcessor class that is created handles the interaction with the DAW, taking midi data as input, applying the DSP processing and then outputting audio data. This class is where the DSP objects are instantiated and the audio data they create is rendered. Also, the PluginEditor class is created, this handles the creation of the GUI and is where each of the GUI components are instantiated.

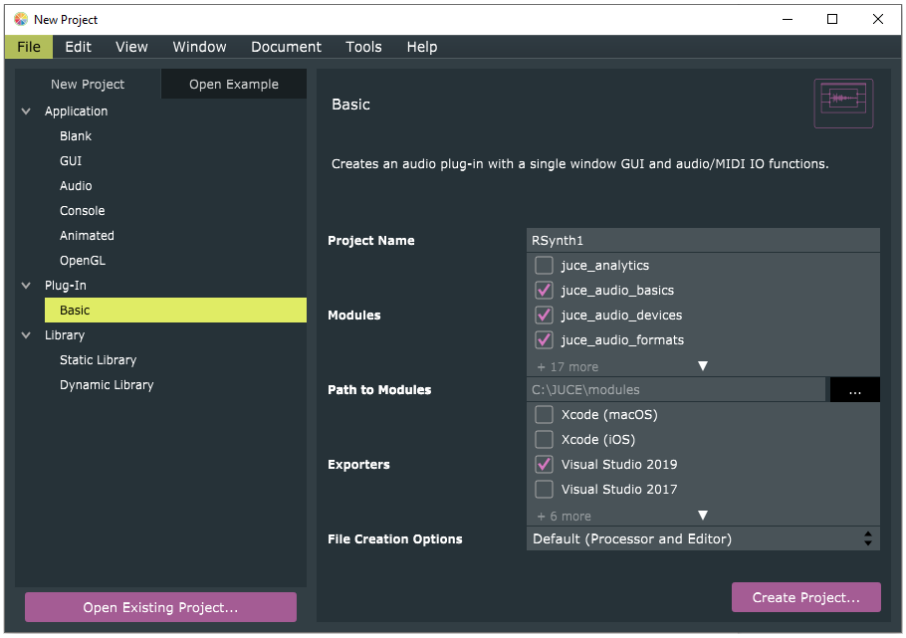

Fig 27. Projucer new project screen

4.2.2 Adding the Maximillian library to the synthesizer dependencies

In order to use the Maximilian library to implement the synthesizer components identified in the requirements the library was added to the projucer project dependencies. This is done by adding the location of the libraries to the header search path. I decided to use this library to implement synthesizer voice components because it was quick to use and relatively easy to understand.

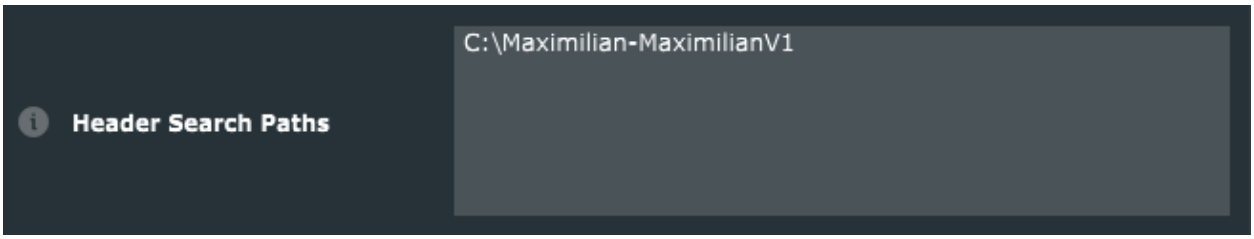

Fig 27. Adding the Maximillian file path

4.2.3 Setting up the debug environment

Since a plugin is required to have a host environment for it to function, an appropriate debug environment must be used to fulfill this role. The JUCE library provides the AudioPlugInHost application which can be configured to load when the project debugger is started. This is set up by opening and running the AudioPlugInHost Visual Studio project. First the built VST3 plugin must be scanned for. To do this enter the file location of the compiled vst plugin after selecting the ‘Options/edit list of plugin/scan for new or updated VST plugin’ menu. Now, when the scan button is selected the plugin will show up on the list of available plugins.

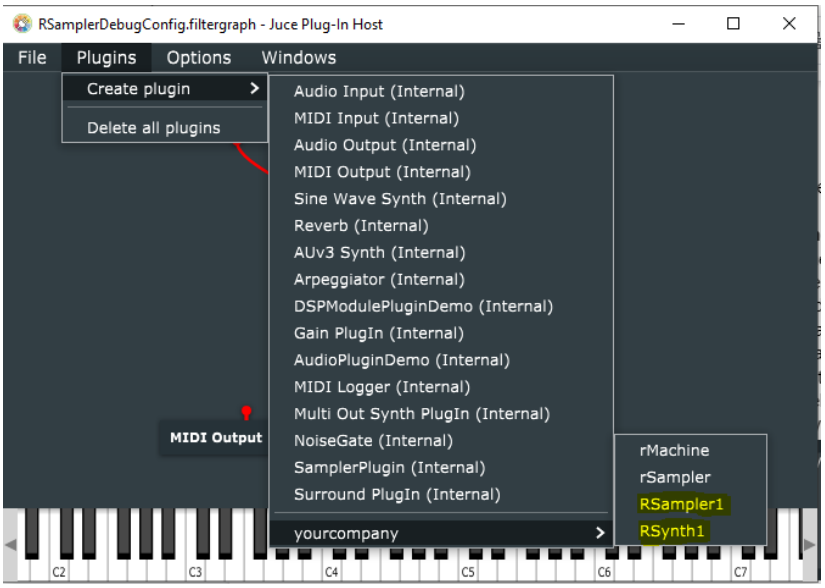

Fig 28. The create plugin drop down when the steps to scan for a plugin have been completed.

4.2.4 Creating the debug configuration

Now a debug configuration file can be created for each of the plugins. This is done by selecting the plugin from the “yourcompany“ option on the “Create Plugins” option in the “Plugins” drop down. This loads the plugin into the debug host environment. Since both plugins take midi data as input and produce audio data, the MIDI input widget is connected to the input of the plugin. Then both the left and right outputs from the loaded plugin are assigned to the “Audio Output” widget. The configuration must then be saved. This creates a “PluginName”DebugConfig.filtergraph file.

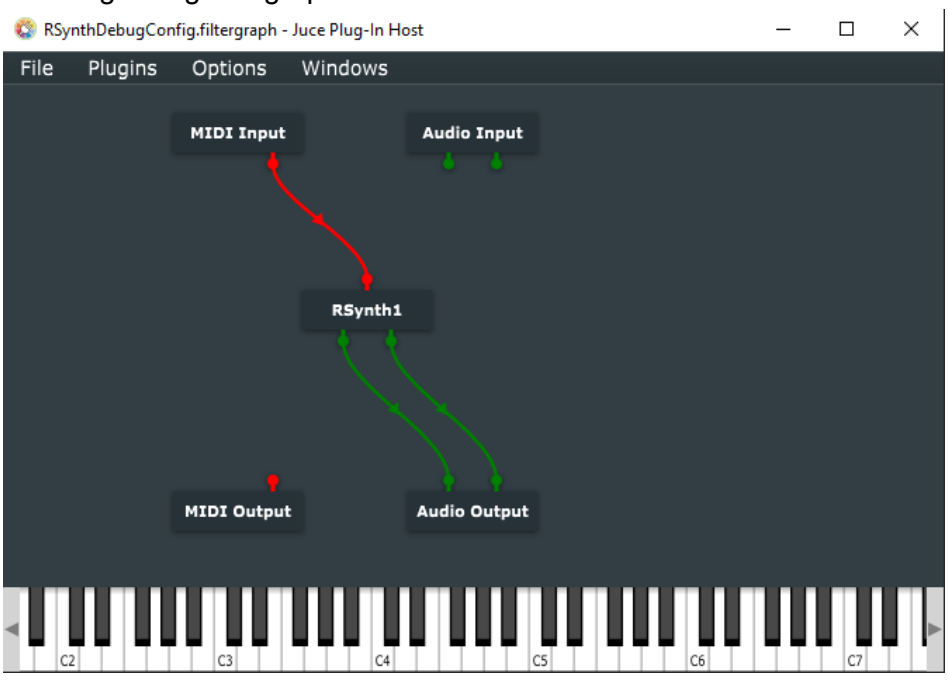

Fig 29. Showing how the widgets should be connected.

4.2.5 Connecting the Visual Studio debugger to the AudioPluginHost environment.

After the configurations have been created all that is left to do is to set up the debug environment in Visual Studio 2019. This is done by pointing the VST3 build target debugger at the AudioPluginHost executable. Now when the debugger is started the AudioPluginHost will open and the debug configuration that has been created can be loaded. The plugin GUI can be opened by double clicking on the plugin widget.

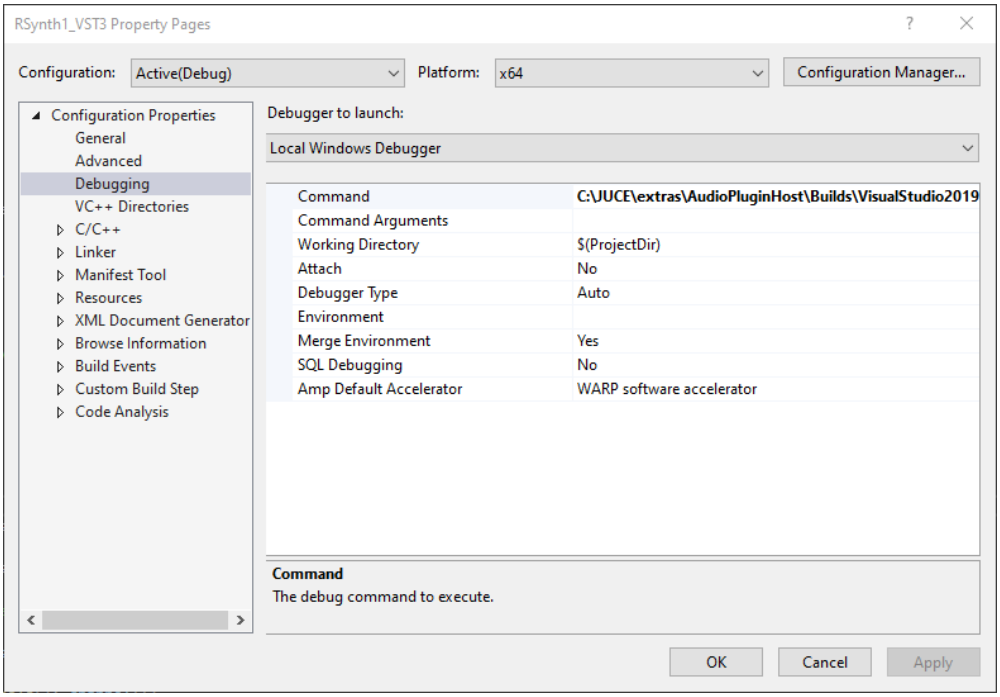

Fig 30. VST3 build target debugging properties page.

4.3 Synthesizer class

4.3.1 Setting up the synthesizer class and adding voices

The synthesizer class is at the heart of the DSP implementation of both plugins. It can be thought of as the shell of the audio data creation part of the plugin in which most of the DSP components are contained inside and controlled by. It is defined as a member variable of the PluginProcessor class; this is so the sound it makes can be rendered in the processBlock method of this class. The synthesiser class cannot make sound on its own, sound and voice objects must be added to define how it sounds. When the plugin processor is constructed, voices are added to a synthesizer class.

For each voice that is added, the addVoice method is called on the synthesiser object, passing a newly constructed SynthVoice as an argument for the synthesizer implementation and a newly constructed SamplerVoice for the sampler implementation. For the SynthVoice class to play a note the signal path needs to be implemented; this is explained in the signal path implementation section. On the other hand, the SampleVoice is a prebuilt class from JUCE and only needs a sound added to it to play a note. As per the requirement 8 voices are added to both implementations. Now the only thing left to-do to set up the synthesizer class is to add the sounds that the voices will play. This is implemented differently in the two plugins. Adding a sound to the synthesizer implementation is very simple. First a class called SynthSound is defined that inherits from the SynthesizerSound class. Then the addSound method is called on the synthesiser object and a newly constructed SynthSound object is passed as an argument. When it comes to adding sounds to the sampler it is a slightly more complicated process because the sounds that should be added to the sampler are audio files, so a method of handling file input must be implemented.

4.3.2 Handling audio files and adding them as sounds to the sampler's synthesizer object.

A way to load an audio file into the sampler is now implemented so it can play it as a sound. Initially, I used a button which allowed the user to select a file to load. This was implemented by using a button listener and a lambda function. When the button is clicked the loadFile method is called that opens the file explorer and adds the selected sample to the synthesizer object as a sound. This works well, but I replaced the button with drag and drop since this functionally is more common to most users of DAWs.

```
void RSampler1AudioProcessor::loadFile() //takes midi note to load sample for
{
    //clear old sounds to replace with new sound
    rSampler.clearSounds();

    juce::FileChooser chooser{ "PLEASE LOAD FILE." };

    if (chooser.browseForFileToOpen())
    {
        //might not check if audio file
        auto file = chooser.getResult();
        rFormatReader = rFormatManager.createReaderFor(file);


    }
    //72 - 80 midinotes for pad sampler style
    juce::BigInteger range;
    //0 - 128 for full pitch range
    range.setRange(0, 128, true);

    rSampler.addSound(new juce::SamplerSound("Sample", *rFormatReader, range, 60, 0.1, 0.1, 10));
    delete rFormatReader; //delete reader after sample read
}
```

Fig 31. Button file loading function

```
sample1Button.onClick = [&]() { audioProcessor.loadFile(); };
```

Fig 32. Listener with a lambda function that calls the file loader.

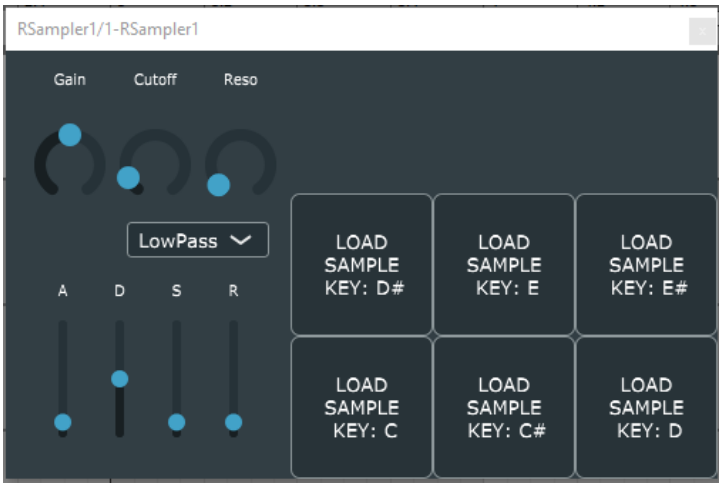

Fig 33. Button loading prototype

4.3.3 Drag and drop sample loading.

In order to load a file when a sample is dropped on the plugin GUI the pluginEditor must be set to accept this functionality. This can be done by inheriting from the FileDragAndDropTarget from the JUCE library. This makes the plugin GUI responsive to files being placed on it. As the sampler can only load audio files, the dropped file should be validated as one of the following file types; .AIF, .WAV and .MP3. To make sure that the plugin does not crash when excessively large files are dropped a 5mb limit has been implemented by checking the number of bytes in the file, this should be big enough for most uses of the sampler.

The file that has been dropped is then added to the sampler's synthesizer object as a sound that can be played across the full range of the keyboard. Now a sample can be added by calling the addSound method on the synthesizer object and passing a new sampler sound object with the arguments; the name of the sample, a reference to the file reader, the keyboard range that the sound should be played over, the normal pitch the sample should be played at which is normally 60 for middle c and the parameters to initialize the envelope. After completing these steps an audio file can be placed on the GUI and if it is valid, it will be added to the synthesizer object as a sound. The sampler can now play this sound back when notes are played.

```
bool RSampler1AudioProcessorEditor::isInterestedInFileDrag(const juce::StringArray& files)
{
    //check if dropped file is valid type
    for (auto file : files)
    {
        if (file.contains(".wav") || file.contains(".mp3") || file.contains(".aif"))
        {
            return true;
        }
    }
    return false;
}

void RSampler1AudioProcessorEditor::filesDropped(const juce::StringArray& files, int x, int y)
{

    for (auto file : files)
    {
        //if file valid, load in processor
        if (isInterestedInFileDrag(files))
        {
            //load sound
            audioProcessor.loadFileDragDrop(file);

        }
    }
    repaint();
}
```

Fig 34. File handling method

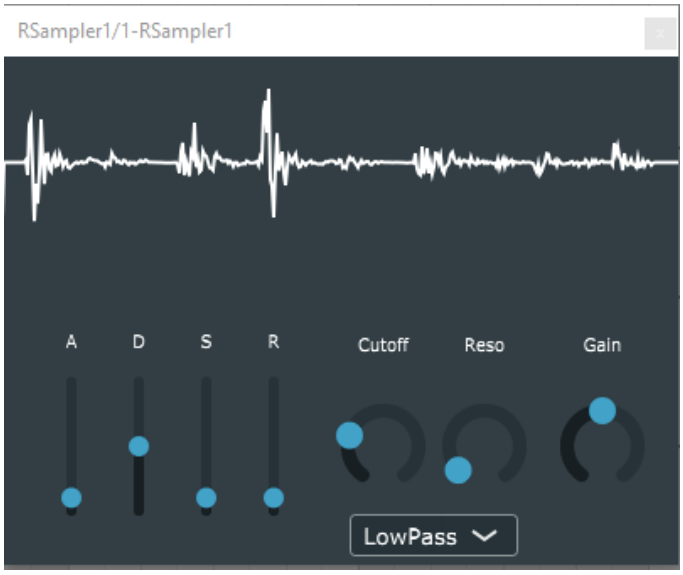

Fig 35. Drag and drop sampler prototype

4.4 Implementing the synthesizer voices signal path

In order to implement the signal path in the voice class a midi note must be passed to the voice and converted to a frequency. The converted frequency can then be used by an oscillator to generate the sample that the rest of the signal path will process. Notes that should be played by a voice start off as midi data and can be assigned to a voice by passing a midibuffer to the synthesizer classes' renderNextBlock method. This is a buffer structure that holds a sequence

of midi noteOn and noteOff events each tagged with a time value and sorted from earliest to latest. Any noteOn events that are in the buffer will be assigned to a voice by calling its startNote method and passing the midiNoteNumber of the note that should be converted to a frequency value. The frequency value that is converted must be assigned to a variable that can be accessed by the rest of the signal path.

```
void SynthVoice::startNote(int midiNoteNumber, float velocity, juce::SynthesiserSound* sound, int currentPitchWheelPosition)
{
    if (auto* checkSound = dynamic_cast<const juce::SynthesiserSound*> (sound))
    {
        frequency = juce::MidiMessage::getMidiNoteInHertz(midiNoteNumber);
```

Fig 36. Showing how the oscillator frequency is set.

4.4.1 Implementing the LFO

Now that the voice is initialized with a frequency, before it is used to generate an audio sample, a modulation value should be generated with the LFO and applied to the frequency. The sine wave method from the maxiOsc class is used to generate the modulation by passing the frequency of the LFO as an argument. The returned value can then be used to modulate the voice's frequency. Since the voice's frequency should be modulated before any samples are generated it must be the first component called when the signal path is created in the voice classes' renderNextBlock method.

```
double SynthVoice::getLFO()
{
    return lfo.sinewave(lfoRate) * lfoDepth;
}
```

Fig 37. Showing how the LFO is generated, and the depth multiplier is applied.

```
currentFrequency = frequency + (frequency * getLFO());
```

Fig 38. Showing how the voice frequency is modulated with the LFO.

4.4.2 Implementing an oscillator.

Once the voice frequency has been set and modulation has been applied the frequency can be used by the oscillators to generate an audio sample. The maxiOsc class from the maximilian library has been used to implement all three oscillator types. To generate a sample the required waveform method is called with the frequency of the sample passed as an argument. The result of calling this method is the audio sample source of the signal path which will then be manipulated by the other components of the synthesizer. The code snippet below shows how a saw wave is generated.

```
double SynthVoice::getSawOsc()
{
    return sawOsc.saw(currentFrequency);
}
```

Fig 39. Example of using the maxiOsc class.

4.4.3 Implementing a mixer.

After the samples have been generated by the oscillators their volume is controlled by the mixer. The three oscillators' samples are mixed together so they can be processed by the other components as one sample. This functionality is defined in the combineOsc method. Inside this method each of the signals are collected and multiplied by the volume level set in the GUI. After this all the signals are added together and returned. To generate the audio sample that will be processed by the other components the combineOsc method should be the next part of the signal chain that is called in the renderNextBlock method after the frequency is modulated.

```
double SynthVoice::combineOsc()
{
    return (getSqrOsc() * sqrOscLevel) + (getSawOsc() * sawOscLevel) + (getSubOsc() * subOscLevel);
}
```

Fig 40. Showing how the oscillators are combined.

```
double oscSound = combineOsc(); //get Oscillators sources
```

Fig 41. Showing how the code above is called in the signal path processBlock.

4.4.4 Applying a filter to the sample.

The next step in the signal path is to apply the filter to the generated audio sample. This is implemented with the maxiFilter class. Since the filter should be able to apply resonance at the cutoff, the lowres and hires methods from the filter class have been used. The filter that the signal should pass through is controlled by a button on the GUI. Both filter methods take the same arguments. The first argument passed is the sample that has been combined by the mixer, this is the sample that will be filtered. The other two arguments passed are the cutoff and resonance which will be set on the GUI.

```
//apply filter
if (fChoice == 0.0f)  //low pass filter signal
    outputSound = filter1.lores(oscSound, cutoff, resonance);
else                  //high pass filter signal
    outputSound = filter1.hires(oscSound, cutoff, resonance);
```

Fig 42. Showing how the maxiFilter object is used.

4.4.5 Creating the envelope ramps

Once the filter has been applied the next step in the signal path is to generate the envelope and use it to modulate the amplifier's gain level, so it changes over time. The adsr method from the maxiEnv class is used to implement the envelope since it must be able to generate attack, decay, and release ramps of a certain length and sustain at a selected amplitude. Each of these phases are controlled by the envelope component on the GUI. When generating an envelope, it is important that each of the phases are generated at the correct time. To implement this functionality a set of variables that control which phase the envelope should be in are assigned inside the StartNote and StopNote methods. The trigger variable is used to control when the envelope should start and continue generating the attack, decay or sustain phases. Since this

should only happen when a note is being played the value is set to one inside the startNote method and set to zero in the stopNote method. Everytime a new note is played the attack ramp is generated and the attackphase variable is set to one. Additionally, because a note may not be held long enough to finish the attack ramp, when a note is stopped the attackphase variable is set to zero. Finally, when a note is no longer being played the release ramp should be generated so the note fades to zero volume, therefore when a note is stopped the releasephase variable is set to one. Now that the control variables have been implemented all that is left to do is generate the envelope and apply it to the gain level. This is done by calling the adsr method in the renderNextBlock method and passing the trigger and gain level as arguments. The variable that is returned from this method can then be applied to a sample to modulate its gain level depending on the phase that the envelope is in.

4.4.6 Implementing the amplifier

Now that the gain level is set and modulated an amplifier has been implemented that will apply soft distortion as the gain level increases while stopping the output volume from getting too loud. In order to do this a waveshaping function is applied to the audio signal. The function I decided to use is particularly convenient as the value that is returned from the function is between the values of -1 and 1, the same range that audio samples should be in before they are output to the DAW. This removes the need to worry about scaling the output to avoid the audio from hard clipping which would cause artifacts. The input to this function is the audio sample multiplied by the modulated gain level. As the gain level applied to the input sample increases and the audio level moves closer to the maximum amplitude this function will slowly remove the louder parts of the sound while keeping the quieter parts. This creates a soft clipping effect which gives the sound more impact and loudness without causing the unwanted hard clipping mentioned earlier. The result of applying this function is the final processed sample that should be stored as one of the samples in the AudioBuffer that is passed to the renderNextBlock method.

```
processedSample = (2.0f / juce::float_Pi) * atan(processedSample * ampLevel);
```

Fig 43. Applying the waveshaper function

4.4.7 Storing the processed sample in the audio buffer

The audio sample generated by the oscillators has been fully processed by passing it to each of the signal path components inside the renderNextBlock method.
Now, the audio buffer is used to store the processed sample sequentially so it can be used by the rest of the application or output to the DAW for conversion to an analog signal. This is implemented by using the AudioBuffer class provided by the Juce library. In my implementation this class is used to store a sequence of channels pairs where each channel contains an audio sample. It is important to point out that the signal path that has been implemented is used to generate an identical pair of samples for every element in the buffer that is passed to the renderNextBlock method. In my implementation the buffer size is 512 so when the

renderNextBlock method is called the resulting audio buffer should be full, containing 512 pairs of identical samples. If any of the elements have not been assigned audio samples, glitches and pauses in the audio playback are likely to occur. When adding an audio sample to the buffer a nested loop structure must be used to first select the position in the buffer that the sample should be stored, inside this loop the sample is generated and processed, after which another loop is used to place identical copies of this sample in each channel. In order to place a sample in the buffer the addSample method must be called on a reference to the buffer. This method takes the selected position in the buffer, the selected channel number and the processed sample as arguments. After calling this method for both channels of every element in the buffer it is now full and ready for output or further processing.

4.5 Creating the GUI

The techniques involved in enabling the accept notes and processing them into audio data have now been discussed; there must now be a way for the user to input values to change the sound of the notes.

4.5.1 Creating a GUI component

The GUI layed out in the design section is implemented so the user can assign values to the DSP algorithms. Each GUI element is contained inside a class that inherits from the JUCE GUI component base class. Inside this class each of the components that make up a single GUI element are defined. When the component is constructed, the sliders are set to vertical as defined in the designs and a link is made between a slider and its label, so they are displayed together. Also, each component is made visible by calling the addAndMakeVisible with a reference to the component. In order to display a whole component on the GUI it should be defined as a member of the pluginEditor class and the addAndMakeVisible method is called in the same way as its internal components. The component is placed on the GUI with the setBounds method passing the size the component should be and its location as arguments. The component will now be displayed when the GUI is loaded, and the user can select values.

```
juce::Slider attackSlider;
juce::Slider decaySlider;
juce::Slider sustainSlider;
juce::Slider releaseSlider;

juce::Label attackLabel{ {}, "A" };
juce::Label decayLabel{ {}, "D" };
juce::Label sustainLabel{ {}, "S" };
juce::Label releaseLabel{ {}, "R" };
```

Fig 44. Declaring each slider and label of the envelope GUI

```
attackSlider.setSliderStyle(juce::Slider::SliderStyle::LinearVertical);
attackSlider.setRange(1.0f, 5000.0f);
attackSlider.setValue(1.0f);
attackSlider.setTextBoxStyle(juce::Slider::NoTextBox, true, 0, 0);
addAndMakeVisible(&attackSlider);
addAndMakeVisible(&attackLabel);
attackLabel.attachToComponent(&attackSlider, false);
attackLabel.setJustificationType(juce::Justification::centred);
attackLabel.setFont(juce::Font(12.0f, juce::Font::plain));
```

Fig 44. Creating attack portion of the envelope GUI

```
void RSynth1AudioProcessorEditor::resized()
{
    juce::Rectangle<int> area = getLocalBounds();

    envGUI.setBounds(145, 160, 400, 200);
```

Fig 46. Placing the envelope on the GUI

4.5.2 Connecting the GUI parameter to the DSP implementation

The values set on the GUI components must now be connected to the DSP algorithms so the user can manipulate the sound of the plugin. This connection has been made by using the AudioProccessorValueState class to store a pointer to each GUI component's value with an associated ID value. Since the value tree must be accessible by the DSP components it is declared in the PluginProcessor class. Before GUI components can be connected, each parameter must be created as an AudioParamter”type” object and added to the value tree, these parameters values will be set by the GUI components. When constructing this object, the first and most important parameter that is passed is the parameterID, this will be used to attach the GUI component and access the value that is stored. Also, the range of values that the parameter should store and a value to initialize it at are passed. The value tree is created in the Audioproccessor constructor and the parameters it should store are passed as a ParameterLayout object. Now that the value tree has been created, we must now make it so that values set on GUI components are reflected in the value tree. To create the connection, when a GUI component is constructed, a value tree Attachment object is instantiated for each input method. The first argument passed is the value tree the input method should be connected to, next the parameterID is passed to select which parameter the input method should be assigned to, last the input method from which the value should be taken is passed. Now that the connection is made between the GUI and the Audio Processor the stored values can be used in the DSP implementation.

4.5.3 Applying GUI values to the synth voice

Now that the value tree is set up it is simple to collect values from it and assign them to the synth voices. Since the parameters should be applied to the voices in real time the values are set in the proccessBlock method of the audioProccessor class. Identical values are set for each

voice by calling the setter methods for all the components inside a loop. The values that are passed to the setter method are collected from the value tree by calling the getRawParameterValue method and passing the appropriate parameterID. For example, if the attack value was being passed the method would be called with “ATTACK” as the argument. Inside the setter method values are dereferenced and assigned to the appropriate variable. Now this has been implemented when the user changes values on the GUI they are reflected in the sound that is generated by the plugin.

```
juce::AudioProcessorValueTreeState valueTree;
```

Fig 47. Value tree declared in the PluginProcessor class header file.

```
juce::AudioProcessorValueTreeState::ParameterLayout RSynth1AudioProcessor::createParams()
{
    std::vector<std::unique_ptr<juce::RangedAudioParameter>> params;

    params.push_back(std::make_unique<juce::AudioParameterFloat>("ATTACK", "Attack", juce::NormalisableRange<float>(1.0f, 5000.0f), 1.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("DECAY", "Decay", juce::NormalisableRange<float>(1.0f, 2000.0f), 500.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("SUSTAIN", "Sustain", juce::NormalisableRange<float>(0.0f, 1.0f), 0.1f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("RELEASE", "Release", juce::NormalisableRange<float>(1.0f, 5000.0f), 1.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("CUTOFF", "Cutoff", juce::NormalisableRange<float>(20.0f, 5000.0f), 2000.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("RESO", "Resonance", juce::NormalisableRange<float>(0.0f, 10.0f), 0.5f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("AMP", "Amp", juce::NormalisableRange<float>(0.0f, 5.0f), 1.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("SAWOSC", "SawOsc", juce::NormalisableRange<float>(0.0f, 0.33f), 0.15f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("SUBOSC", "SubOsc", juce::NormalisableRange<float>(0.0f, 0.33f), 0.15f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("SQROSC", "SqrOsc", juce::NormalisableRange<float>(0.0f, 0.33f), 0.15f));
    params.push_back(std::make_unique<juce::AudioParameterBool>("FILTERTYPE", "Hi/Lowpass", false));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("CHORUSMIX", "ChorusMix", juce::NormalisableRange<float>(0.0f, 1.00f), 0.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("CHORUSDEPTH", "ChorusDepth", juce::NormalisableRange<float>(0.0f, 1.00f), 0.0f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("ARPSPEED", "ArpSpeed", juce::NormalisableRange<float>(0.0f, 1.00f), 0.5f));
    params.push_back(std::make_unique<juce::AudioParameterBool>("ARPONOFF", "On/Off", false));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("LFORATE", "lfoRate", juce::NormalisableRange<float>(0.0f, 5.0f), 0.1f));
    params.push_back(std::make_unique<juce::AudioParameterFloat>("LFODEPTH", "lfodepth", juce::NormalisableRange<float>(0.0f, 1.0f), 0.0f));

    return{ params.begin(), params.end() }; //returns parameters list
}
```

Fig 48. Creating the parameter that the value tree will store.

```
RSynth1AudioProcessor::RSynth1AudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       ), valueTree(*this, nullptr, juce::Identifier("RSynthParameters"), createParams())
```

Fig 49. Adding the value tree to the plugin processor.

```
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> attackVal;
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> decayVal;
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> sustainVal;
std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> releaseVal;
```

Fig 50. Declaring Slider attachments in a GUI component class header file

```
attackVal = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment> (processor.valueTree, "ATTACK", attackSlider);
decayVal = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>(processor.valueTree, "DECAY", decaySlider);
sustainVal = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>(processor.valueTree, "SUSTAIN", sustainSlider);
releaseVal = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>(processor.valueTree, "RELEASE", releaseSlider);
```

Fig 51. Attaching sliders to their parameters when a component is constructed

```
rVoice->setADSRParams(valueTree.getRawParameterValue("ATTACK"),
                      valueTree.getRawParameterValue("DECAY"),
                      valueTree.getRawParameterValue("SUSTAIN"),
                      valueTree.getRawParameterValue("RELEASE"));
```

Accessing parameters from the value tree

```
void SynthVoice::setADSRParams(std::atomic<float>* attack, std::atomic<float>* decay, std::atomic<float>* sustain, std::atomic<float>* release)
{
    env.setAttack(*attack);
    env.setDecay(*decay);
    env.setSustain(*sustain);
    env.setRelease(*release);
}
```

Fig 52. Setting the envelope DSP component values

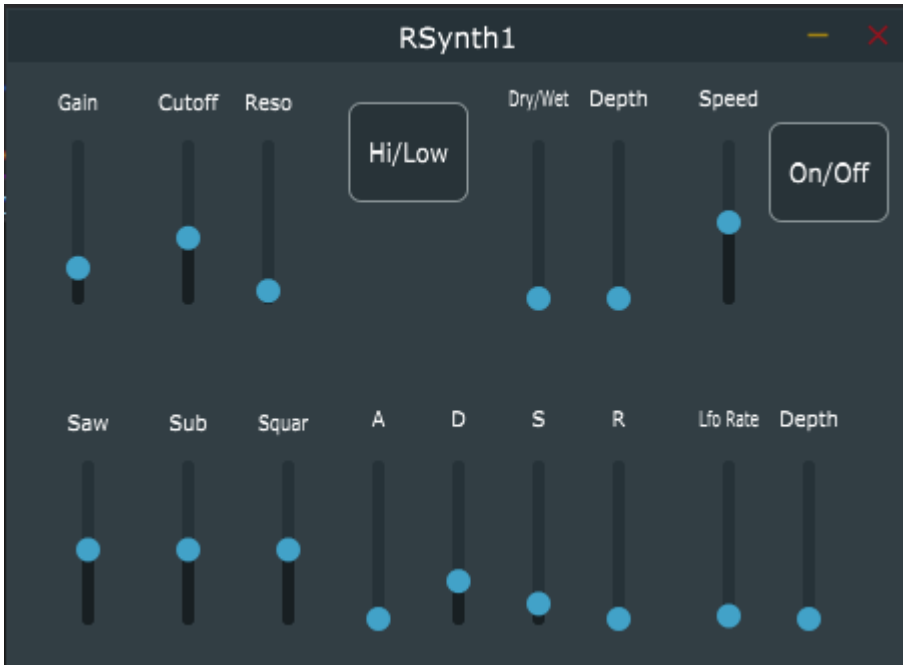

Fig 53. The final synthesizer implementation

Fig 54. The final sampler implementation

4.6 The testing approaches.

When testing my implementations an exploratory testing approach was used. This involved trying to break the plugins through various techniques. I decided not to use a unit testing approach as it would not test the integration of the plugin into a DAW environment very well. Some examples of the approaches I used to try and break the system include.

- Testing sample acceptance.
    - Trying to drop excessively large audio files onto the sampler.
    - Trying to place files of the wrong type onto the sampler.
- Testing the envelope stages.
    - Setting the attack to max and repeated pressing key to see if the attack phase was restarted every time.
    - Setting the sustain level to minimum and decay to the maximum to check if the sustain level follows the decay phase.
    - Setting the attack to minimum and the decay to maximum to check if the decay phase still occurs without the attack phase.
    - Setting all the values to the minimum to check that no sound is created. (only small clicks).

- Testing each slider can be moved to its maximum and minimum value without getting stuck.
- Moving a slider very quickly to see if the sound change keeps up.

All these tests were carried out in Ableton and the juce AudioPluginHost as host environment. No errors occurred.

Also, acceptance testing was carried out by using each plugin in an Ableton live project. Midi data was created and successfully used as input and the generated audio data was successfully output to the speakers and recorded to the host environment.

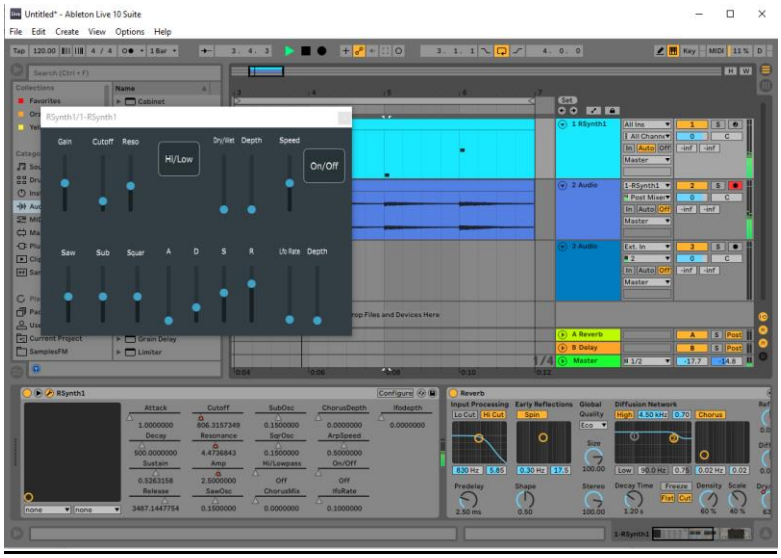

Fig 55. Synth plugin loaded into a DAW host environment during acceptance testing

5. Critical reflection

Overall, the project sticks to the aims and objects laid out in the specification and the implementation meets and exceeds the initial specification that was defined. Extra features that were not in specification have been added and they all function effectively in the DAW environment. An arpeggiator, for example, was not something I was intending to include but after understanding enough of the JUCE library I found I could implement this effectively. The decisions to use a prototyping approach also made extensions like this easy to include and alternative directions easy to evaluate. From this I think I can say the implementation has been very successful, especially considering that the concepts are not covered on my course.

On the other hand, in the research section of the project I did not direct my effort very effectively. This is mostly down to the fact that at the start of the project I was still working out exactly what form the implementation should take. At first, I was interested in the idea of implementing a plugin that was driven by machine learning, after which I looked at low level

DSP implementations before reaching a JUCE plugin with support from a DSP library as the final idea. This meant I spent a lot of time researching areas that were not very informative to the final direction of the project.

The most challenging part of the project was the write up and if I were to do it again much more time would be spent keeping a journal of the exact steps taken to reach the final outcome.

Future developments

The discovery of the JUCE library was one of the best parts of this project and looking into it and all the possibility has made me think of a variety of ways both these implementations could be extended. Some of these include.

- Adding a real time visualization of the frequency spectrum of the audio data using the Fourier transform implementation provided by JUCE. This is more of a look and feel extension, but it looks interesting to investigate.
- Using the velocity values and pitch wheel values. Throughout the project I ignored the velocity and pitch wheel values but if they were implemented, the breadth of tones the plugin could create would be increased dramatically and make the instrument more playable.
- Implementing more midi creation and handling by adding a chord memory function in which a user creates a chord that can then be played with one key and transposed across the keyboard.
- Data driven/machine learning chord progression plugin.
- Effect's plugin.

Personal reflection

Personally, this project has been a very rewarding endeavor albeit challenging at stages. As someone with a keen interest in music and the accompanying technology it has been interesting to learn how some of the applications that I have been using for years in my personal life may have been developed. It is rewarding to know that from the knowledge I have gained about the JUCE library it would now be possible for me to develop a complete audio plugin for use in my own music production. Additionally, my knowledge of general synthesis ideas and DSP has been greatly improved due to the research carried out in preparation for the project. Also, from a professional point of view I am now very interested in the idea of doing work in the audio app development field.

6. References

Chowdhury, J. (2020). *Why I use JUCE*. From https://jatinchowdhury18.medium.com/why-i-use-juce-fae2b1b7441e

Farcic, V. (2014). *Iterative and Incremental Development*. From Technology Conversations: https://technologyconversations.com/2014/01/21/software-development-models-iterative-and-incremental-development/

Steven W. Smith, P. (1999). *The Scientist and Engineer's Guide to*. From dspguide: http://www.dspguide.com/ch4/3.htm

7. Appendix A – Project Specification document

PROJECT SPECIFICATION - Project (Technical Computing) 2020/21

Student:	**Ruari Molyneux**
Date:	**30/10/2020**
Supervisor:	**Christopher Bates**
Degree Course:	**Computer Science Bsc**
Title of Project:	**Developing Virtual Studio Technology Plugins**

Elaboration

This project will explore digital signal processing and the creation of original sound with these ideas. The main direction of the project is to investigate the different methods of digital synthesis and sample playback, which will be outlined in my research. These ideas will then be brought together into a piece of software. The software will allow the user to experiment with different synthesis methods, sample processing and provide a way to sequence, edit and export sounds.
The project will expand my set of skills to cover Digital Signal Processing an area of interest that is not covered in my usual course syllabus.
A Trello will be set up a to track the progress of the research and development of the project. Also, through my project research a set of libraries will be identify for use in the software implementation.

Project Aims

Aims are **what** you want to achieve. Objectives are **how** you get to your aims.
This project aims to.

- Investigate digital synthesis methods
- Identify a set of libraries and a development platform for my digital signal processing needs
- Investigate methods to apply effects to sounds
- Investigate sample playback and sequencing methods

- Investigate sound envelopes
- Develop a piece of software to demonstrate the digital signal processing techniques I have researched.
- Demonstrate software development practices like testing and iterative development

Project deliverable(s)

A piece of software for digital synthesis, sample playback and sequencing that will be developed in python. A specific platform for development will be identified in my research.
The application will include.

- A method of digital sound synthesis
- Sample playback
- Effects (low pass and high pass filters)
- A way to envelope sounds
- A sequencer
- An ability to export sounds as wav files

An iterative software engineering approach will be used as this is a solo project with a relatively small codebase. A Git hub repository will be set up to handle version control throughout the project.
Testing will be done by using exploratory testing and acceptance testing.

Action plan

Action	Deadline Date
Finding a supervisor	**9th October**
Project specification and ethics form	**30th October**
Background Research -Investigate simple waveforms and research digital synthesis methods -Research Digital Signal Processing python libraries and GUI libraries. -Investigate sample playback and sequencing -Investigate envelopes and effects	**23rd November(Overall)** 30th October 2nd November 9th November 16th November
Information review	**4th December**
Develop the application Implement the back-end algorithms Connect back-end with GUI	**5th February** 18th January 1st February
Test the application	**2nd February**
Improve depended on test	**10th February**
Agree contents page with supervisor	**19th February**
Submit draft critical evaluation	**19th March**
Electronic Submission	**15th April**
Demo to supervisor	**29th April**

BCS Code of Cduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.
Signature:

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.
Signature:

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire or participant testing materials. This form is available on the Bb site for the module and as an appendix in the handbook.
Signature:

8. Appendix B – The Ethics form

UREC 1 RESEARCH ETHICS REVIEW FOR STUDENT RESEARCH WITH NO HUMAN PARTICIPANTS OR DIRECT COLLECTION OF HUMAN TISSUES, OR BODILY FLUIDS.

All University research is required to undergo ethical scrutiny to comply with UK law. The SHU Research Ethics Policy should be consulted before completing the form. Answering the questions below will confirm that the study fits this category and that any necessary approvals or safety risk assessments are in place. The supervisor will approve the study, but it may also be reviewed by the College Teaching Programme Research Ethics Committee (CTPREC) as part of the quality assurance process.

The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research.

Note that students and staff are responsible for making suitable arrangements to ensure compliance with the General Data Protection Regulations (GDPR), for keeping data secure and if relevant, for keeping the identity of participants anonymous. They are also responsible for following SHU guidelines about data encryption and research data management. Information on the ethics website

The form also enables the University and College to keep a record confirming that research conducted has been subjected to ethical scrutiny.

The form may be completed by the student and the supervisor and/or module leader (as applicable). In all cases, it should be counter-signed by the supervisor and/or module leader, and kept as a record showing that ethical scrutiny has occurred. Students should retain a copy for inclusion in the appendices of their research projects, and a copy should be

uploaded to the module Blackboard site for checking.

Please note if it may be necessary to conduct a health and safety risk assessment for the proposed research. Further information can be obtained from the Safety Co-ordinator.

1. General Details

Name of student	Ruari Molyneux
SHU email address	B8010709@my.shu.ac.uk
Course or qualification (student)	Computer Science Bsc
Name of supervisor	Christopher Bates
email address	cmscb@exchange.shu.ac.uk
Title of proposed research	**Developing software for audio synthesis and sample playback.**
Proposed start date	31/10/2020
Proposed end date	
Brief outline of research to include, rationale & aims (250-500 words).	This project will explore digital signal processing and the creation of original sound with these ideas. The main direction of the project is to investigate the different methods of digital synthesis and sample playback, which will be outlined in my research. These ideas will then be brought together into a piece of software. The software will allow the user to experiment with different synthesis methods, sample processing and provide a way to sequence, edit and export sounds. This project aims to. • Investigate digital synthesis methods • Identify a set of libraries and a development platform for my digital signal processing needs • Investigate methods to apply effects sounds • Investigate sample playback and sequencing methods • Investigate sound envelopes • Develop a piece of software to demonstrate the digital signal processing techniques I have researched. • Demonstrate software development practices like testing and iterative development The project will expand my set of skills to cover Digital Signal Processing an area of interest that is not covered in my usual course syllabus

I confirm that this study does not involve collecting data from human participants _/_ (please tick)

2. Research in Organisations

Question	Yes/No
1. Will the research involve working with/within an organisation (e.g. school, business, charity, museum, government department, international agency, etc.)?	no
2. If you answered YES to question 1, do you have granted access to conduct the research? *If YES, students please show evidence to your supervisor. PI should retain safely.*	n/a
3. If you answered NO to question 2, is it because: A. you have not yet asked B. you have asked and not yet received an answer C. you have asked and been refused access. *Note: You will only be able to start the research when you have been granted access.*	n/a

3. Research with Products and Artefacts

Question	Yes/No
1. Will the research involve working with copyrighted documents, films, broadcasts, photographs, artworks, designs, products, programmes, databases, networks, processes, existing datasets or secure data?	no
2. If you answered YES to question 1, are the materials you intend to use in the public domain? *Notes: 'In the public domain' does not mean the same thing as 'publicly accessible'.* • *Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission.* • *Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc.* *If you answered YES to question 1, be aware that you may need to consider other ethics codes. For example, when conducting Internet research, consult the code of the Association of Internet Researchers; for educational research, consult the Code of Ethics of the British Educational Research Association.*	n/a
3. If you answered NO to question 2, do you have explicit permission to use these materials as data? *If YES, please show evidence to your supervisor.*	n/a

4. If you answered NO to question 3, is it because: A. you have not yet asked permission B. you have asked and not yet received and answer C. you have asked and been refused access. *Note You will only be able to start the research when you have been granted permission to use the specified material.*	**A/B/C**

Adherence to SHU policy and procedures

Personal statement	
I can confirm that: • I have read the Sheffield Hallam University Research Ethics Policy and Procedures • I agree to abide by its principles.	
Student	
Name: Ruari Molyneux	Date: 30/10/2020
Signature:	
Supervisor or other person giving ethical sign-off	
I can confirm that completion of this form has confirmed that this research does not involve human participants. The research will not commence until any approvals required under Sections 3 & 4 have been received and any health and safety measures are in place.	
Name:	Date:
Signature:	
Additional Signature if required:	
Name:	Date:
Signature:	

Please ensure the following are included with this form if applicable, tick box to indicate:

	Yes	No	N/A
Research proposal if prepared previously	☐	☐	☐
Any associated materials (e.g. posters, letters, etc.)	☐	☐	☐
Health and Safety Project Safety Plan for Procedures	☐	☐	☐