

ECE 4040  
Group M2 Final Design  
Digital Music Synthesizer

University of New Brunswick

April 11, 2013

## Summary

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Problem Statement . . . . .	8
1.2	Document Layout . . . . .	9
<b>2</b>	<b>Product Requirements</b>	<b>11</b>
2.1	Customer Requirements . . . . .	11
2.2	Engineering Requirements . . . . .	13
<b>3</b>	<b>Research</b>	<b>15</b>
3.1	Synthesis . . . . .	15
3.2	Linux . . . . .	25
3.2.1	Development Environment . . . . .	28
3.2.2	IO Management . . . . .	28
3.2.3	Concurrency Handling . . . . .	29
3.3	Sound in Linux & ALSA . . . . .	29
3.3.1	ALSA API . . . . .	33
3.4	Android Application . . . . .	36
3.5	Communications . . . . .	38
3.5.1	Communication Protocols . . . . .	38
3.5.2	MIDI Hardware . . . . .	43
3.5.3	MIDI Software . . . . .	44
3.5.4	MIDI USB Interface . . . . .	45
<b>4</b>	<b>Final Design</b>	<b>47</b>
4.1	Synthesis . . . . .	47
4.1.1	The Digital Waveguide . . . . .	49
4.1.2	Piano Model . . . . .	65
4.1.3	Harpsichord . . . . .	78
4.1.4	Clarinet . . . . .	81
4.1.5	Flute . . . . .	93
4.1.6	Guitar . . . . .	101
4.1.7	Tuning and Initial Testing . . . . .	109
4.2	Software . . . . .	112
4.2.1	MIDI as a System Wide Communication Scheme . . . . .	114
4.2.2	Synthesis Core . . . . .	115
4.3	Top Level Program . . . . .	129
4.4	Android Application . . . . .	133

4.4.1	Application Environment . . . . .	133
4.4.2	Application Functionality . . . . .	134
4.4.3	Keyboard . . . . .	136
4.4.4	Instrument Select . . . . .	138
4.4.5	Volume Slider . . . . .	139
4.4.6	Bluetooth Connection . . . . .	140
4.4.7	Sending Data . . . . .	142
4.4.8	User Warnings and Messages . . . . .	142
<b>5</b>	<b>Specifications &amp; Dependancies</b>	<b>143</b>
5.1	Hardware Dependancies . . . . .	144
5.1.1	CPU . . . . .	144
5.1.2	Soundcard . . . . .	145
5.1.3	USB Port . . . . .	145
5.1.4	Bluetooth . . . . .	146
5.2	Software Dependancies . . . . .	147
5.2.1	Linux Operating System . . . . .	147
5.2.2	ALSA Development Package . . . . .	148
5.2.3	Java Runtime Environment . . . . .	149
5.2.4	Bluetooth . . . . .	150
<b>6</b>	<b>Final Product Prototype</b>	<b>152</b>
<b>7</b>	<b>Conclusion and Future Work</b>	<b>158</b>
<b>A</b>	<b>Testing and Verification</b>	<b>161</b>
A.1	Submodule Testing and Development . . . . .	162
A.1.1	Instrument Models and Individual notes . . . . .	162
A.1.2	Synthesis Software . . . . .	163
A.2	Overall System Testing . . . . .	164
<b>B</b>	<b>Android Table</b>	<b>167</b>
<b>C</b>	<b>User Manual</b>	<b>167</b>
C.1	Starting the GUI and using the software . . . . .	167
C.2	Connecting a controller . . . . .	170
C.2.1	Connecting a MIDI controller . . . . .	170
C.2.2	Connecting via Bluetooth . . . . .	171
C.2.3	Connecting a Virtual MIDI device . . . . .	173
C.3	Troubleshooting . . . . .	173
C.3.1	Bluetooth troubleshooting . . . . .	174

C.3.2	Nexus 7 Software troubleshooting . . . . .	175
C.3.3	GUI Troubleshooting . . . . .	176
C.4	Using the Nexus 7 tablet to control other MIDI devices . . . . .	177
<b>D</b>	<b>References</b>	<b>179</b>

**Figures**  
**Equations**  
**Tables**

# **1 Introduction**

## **1.1 Problem Statement**

Musical instruments have time varying output waveforms that are exceedingly complicated and thus extremely difficult to synthesize. Stringed instruments in particular present a number of difficulties with respect to synthesis because each instrument has its own unique characteristics in terms of sound transduction that need to be taken into account in order to accurately reproduce each note. The objective of this design project was to create a synthesizer that can accurately reproduce the sound created by a stringed instruments as well as woodwind instruments and to implement this synthesizer in a software platform that will provide the user with control of which note is being played as well as basic control of certain aspects of the instrument itself.

## **1.2 Document Layout**

This document is broken down into 6 main sections: Introduction, Product Requirements, Initial Research, Final Design, Specifications, and Conclusion and Future Work. The product requirements section outlines the customer and engineering requirements. The initial research section outlines the research that was done in order to determine how each of the functional blocks required to make the synthesizer would be accomplished. The final design section talks about how each of the functional blocks was implemented in order to create the final product. The specifications section discusses what the customer needs in order to use the final prototype of the product. The document concludes with a section that discusses what has been accomplished along with what could be done to improve the synthesizer in the future.

At the end of the document, there is an appendix which includes three sections: Testing and Verification, User Guide, and a Bibliography. The testing and verification section outlines the tests that were carried out on the final prototype along with the results from each test. The user guide is an easy to read instruction manual which indicates how to use the synthesizer along with how to trouble shoot it should the need arise.

Lastly, an electronic appendix is included that holds all of the code that was written for this project. This includes Matlab code for all of the digital waveguide models that were created in Matlab along with all of the structures created to describe each note for each model. All of the C code files that were written to implement these models in real time are also attached in this appendix. Lastly, the code for the GUI on the computer as well as the code for the mobile android app is included.

## **2 Product Requirements**

### **2.1 Customer Requirements**

The customer requirements were determined through planning sessions with our customer, Professor Veach. The requirements were broken down into sections that outline the basic desires of the customer.

The first section covers the basic functionality of the music synthesizer. The customer desires a music synthesizer that produces sound using some form of digital synthesis. The platform for the synthesizer may be software based and the sounds produced by the synthesizer should emulate the sound of actual instruments. The instruments desired by the customer include



a piano, a harpsichord, at least one woodwind and a third stringed instrument, such as a guitar or violin. Any additional instruments are also welcome.

The device must also have an easy to use playing interface. This includes a one or two octave keyboard for note selection; a master volume control; instrument selection control; and finally a display to show the user what options have been selected. The customer desires this playing interface to be implemented as a tablet application and a software GUI will also be necessary to operate the system using a standard MIDI controller. The tablet should be able to communicate with the synthesizer by either direct connection or wireless communication.

The customer desires the system to be capable of standard audio output. To do this the system must be able to connect to external powered speakers which can be achieved through a standard computer audio output jack.

Finally the budget for this project as laid out by the customer is \$200. A high level diagram of the conceptual system is shown below in figure 2.1.

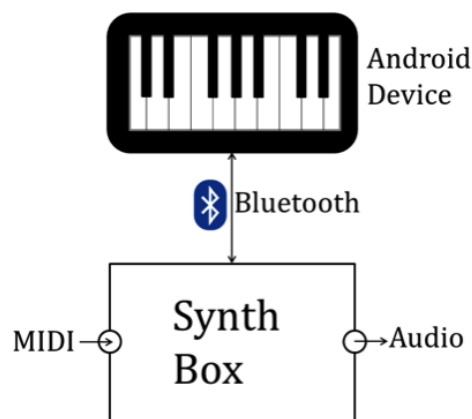


Figure 2.1: Conceptual design generated from customer requirements

## 2.2 Engineering Requirements

The functionality requirements are:

### **Realistic Representation of Various Instruments :**

Method of synthesis which produces sound indistinguishable from the true instrument

### **Real Time Sound :**

Latency between key press and sound output less than 125ms

### **Good output sound quality :**

sound output at 44.1 kHz with 16 bit resolution

### **Multiple notes playable at a time :**

Capable of synthesizing 10 notes simultaneously

### **Governing Platform :**

A software based platform to handle input, output, and synthesis

The interface requirements are:

### **Aesthetically pleasing comprehensive interface :**

Develop an Android application for Nexus 7 and a GUI interface of external MIDI control

### **Piano style playing interface :**

One octave display capable of octave selection

### **Communication between governing platform and tablet interface :**

Adhere to MIDI software specification embedded within Bluetooth Communication

### **Communication between governing platform and external MIDI controller :**

Capable of communicating with a MIDI port registered on the governing platform.

## 3 Research

### 3.1 Synthesis

Sound synthesis is one of the most important aspects of this project. Creating simple sounds digitally is not that difficult of a task when using programs such as Matlab. However, creating sounds that imitate the notes created by real instruments is extremely difficult and requires an intimate knowledge of either the sound produced by the instrument or the physics of the instrument itself. Therefore, the first step in this project was to research different methods of sound synthesis to determine how it could be done. Once these methods had been researched, the second step was to pick a synthesis method to be used for this project.

There are a number of methods currently used for sound synthesis. These methods can generally be broken down into four categories: abstract algorithms, sampling synthesis, spectral modeling, and physical modeling.

Abstract algorithms are a group of methods that use complex algorithms in order to try and recreate the sound produced by the instrument. One of the most popular examples of this is FM synthesis. In the simplest case, FM synthesis involves a modulator and carrier signal, as shown below in figure 3.1.

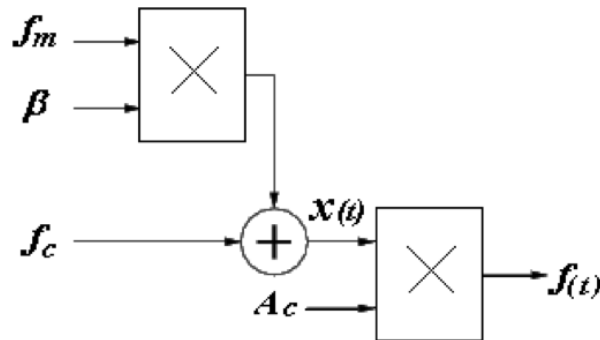


Figure 3.1: Basic Frequency Modulation

The carrier signal creates the main tone, or frequency, while the modulator signal helps to create the harmonics of the signal, or the overtones of the note. The sound created can be varied by altering the modulator frequency, the carrier frequency, the amplitude of the carrier, and the amplitude of the modulating signal. Varying these parameters can change the frequency of the note, the position of the harmonics (overtones), as well as the inharmonicity of the overtones. More complicated FM synthesis methods are possible that cascade multiple modulators, however the simple method presented above gives a good overview of how the sound is created. FM synthesizers were prevalent in the 1980s and 1990s and are ubiquitous in digital music today. However, creating sounds that replicate a real instrument using FM synthesis is difficult and the process is rather abstract, thusly why it is called an abstract algorithm.

Sampling synthesis is rather self-explanatory. Sounds from the instrument are sampled, and then upon request this sound can be played back. The obvious drawback from this method is that once the sound is recorded, only a few variables can be changed in order to vary the output sound. The amplitude can be varied to increase or decrease the volume, the sampling frequency can be changed in order to increase or decrease the apparent frequency,

and pitch shifting can be done in order to vary the delay applied to certain frequencies, distorting the output.

This means that a sampled sound must be stored for every possible parameter variation. For example, a piano can be hit with a number of different hammer velocities. If different hammer velocities were to be implemented using sampling synthesis, the note played would have to be recorded for each hammer velocity, and then would have to be done for each note. So, if there are a number of parameters that the user wishes to vary, it is easy to see that sampling synthesis would require a lot of memory. Therefore, the second main drawback of sampling synthesis is simply the amount of memory required in order to implement an instrument with a number of variable parameters.

Spectral modelling synthesis involves acquiring the output frequency spectrum of one note of the instrument as it changes in time, and trying to recreate this frequency domain progression. One of the commonly used methods for spectral modelling is additive synthesis. Additive synthesis involves analyzing the spectrogram of a note, or the progression in the frequency domain of the note with respect to time, and picking important frequencies in the note. The decay rates, or envelopes, of the important frequencies are then analyzed and stored. The note can then be recreated digitally by implementing digital oscillators for each specified important frequency and varying the amplitude of each frequency with respect to time using the previously identified envelopes. The amplitude of each signal at a given moment can then be summed in order to produce an output. A very basic block diagram for additive synthesis is shown in figure 3.2 below. In this diagram,  $f$  is the desired frequency of the oscillator, and  $r$  is the envelope that is being applied to the frequency.

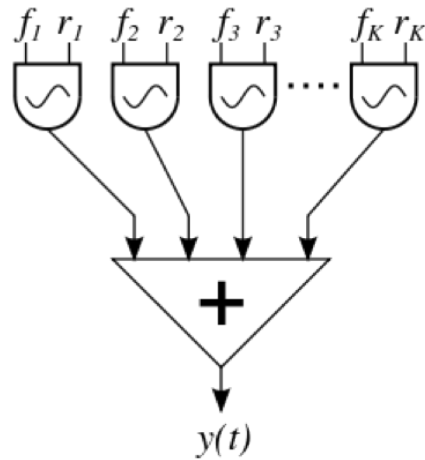


Figure 3.2: Block diagram for an additive clock

Again, the problem with additive synthesis is that it relies on recreating the output note rather than the mechanism that created the note itself. Therefore, if different parameters of the note are changed, then for each combination of parameters the spectrogram of the note needs to be reanalyzed and the important frequencies along with their envelopes need to be determined and stored. Storing these frequencies and the envelope functions would not take up nearly as much space as the stored note in sampling synthesis, however it still means that the output of the real instrument must be analyzed first. This would be very time consuming for an instrument such as a piano.

The last method of sound synthesis that was researched for this project is digital waveguide synthesis, which is a form of physical modeling. Physical modeling differs from the methods mentioned above because rather than just trying to recreate the output sound, it attempts to recreate the mechanism by which the sound was created. This provides some advantages as well as some disadvantages. The main advantage is that by modeling how the sound was produced, different parameters can be changed and the output sound should vary accordingly. This means that only the physics behind the instrument itself needs to be understood. One of the main disadvantages of physical modeling is that a number of physical parameters

of the instrument need to be quantified, which is an extremely difficult task to say the least. The second downside is that physical modeling can be computationally expensive depending on the method that is used. The use of physical modeling in synthesizers has been hindered by the computational requirements. However, with the development of inexpensive and power full DSP microchips, physical modeling is becoming more widely used in industry in high-end synthesizers.

Four very different methods for producing the sound created by instruments are described above. These are the four main synthesis methods that were primarily researched in order to gain a better understanding of how sound can be created. Once this research was completed, a decision needed to be made to determine which synthesis method would be used for this project.

For this project it was decided that physical modeling would be used on order to synthesis the sounds created by the instrument. This decision was made with the help of the decision table shown below. Each method was rated out of 5 for each category and then the final value for each was calculated.

Table 1: Comparison of synthesis techniques.

	<b>Spectral Modeling (Additive)</b>	<b>Sampling Synthesis</b>	<b>Abstract Algorithm Synthesis (FM)</b>	<b>Physical Modeling (Digital Waveguide)</b>
Computational Complexity (0-high complexity)	3	5	3	1
Memory Requirements (0-indicates high memory usage)	3	0	4	4
Intuitiveness	3	5	0	5
Parameter Variation	2	0	1	5
Previous Knowledge of Instrument (output sound and instrument parameters)	3	0	3	3
<b>Total</b>	<b>14</b>	<b>10</b>	<b>11</b>	<b>18</b>

This table clearly shows that physical modeling is the best and most intuitive method to use. The instrument is recreated in a digital form and when parameters are varied it has an immediate effect on the output sound without having to have a bunch of different predetermined values stored in memory.

Physical modeling is still a fairly general synthesis technique and it can be done a number of ways. One of the more popular forms of physical modeling is called digital waveguide synthesis. Digital waveguide synthesis (DWS) implements the string, or hollow tube, of an instrument as a digital delay line and once the string is digitized the specific excitation provided by the instrument as well as any loss mechanisms or sound radiators can be added to the model to accurately recreate the sound of an instrument.



Once the decision was made to use digital waveguide synthesis, there was still a lot of research that had to be done in order to fully understand the digital signal processing involved with DWS and also to determine how each specific instrument could be modeled. The basics behind DWS and the specifics for how it was used to model each instrument created in this project will be explained in the final design section.

## 3.2 Linux

Once viewed as a contrivance to automate the execution of multiple programs in large mainframes, the computer operating system has become an integral part of everyday life. Most will concede that, from an operating system perspective, the personal computer space is dominated by Microsoft's *Windows* series, followed by Apple's *Mac OS*, however savvy readers may acknowledge the presence of a third, albeit less popular, operating system called *Linux*.

Linux is an amalgamation of a kernel developed by Linus Torvalds and the *GNU* toolchain and libraries commonly credited to developer and free software advocate Richard Stallman. For many intents and purposes, Linux can be viewed as a clone of the popular UNIX operating system while maintaining the advantages that it is free and in many instances controllable from both a graphical user interface and command line <sup>1</sup>. Fig. 3.3 below shows the high-level hardware/software stack that makes up the Linux operating system.

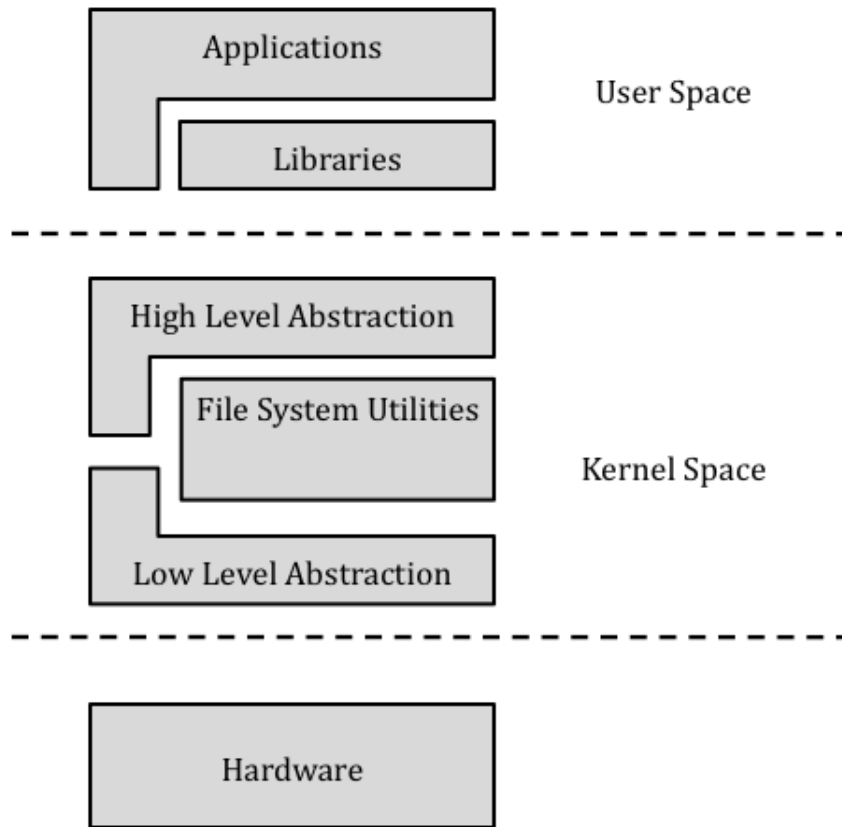


Figure 3.3: Hardware/Software stack in Linux  $\_X X$

At the lowest level in the stack is the hardware. Linux can run on most commonly available CPU architectures including x86, ARM and Microblaze. Interested readers are encouraged to examine the kernel source code for a comprehensive list of supported architects. Drivers have been written to provide functionality for many peripheral devices.

Next is the kernel. The job of the kernel is twofold: to provide a common interface by which user software can interact with hardware (low level abstraction), and to manage multiple programs executing concurrently (high level abstraction). File System Utilities include modules, which are kernel level code that reside in the filesystem and are loaded dynamically when required, and virtualized device files. Generally device drivers fall under the module category.

At the highest level are the applications. These are programs that can be executed by users to control their computer. Most Linux distributions come with a set of programs including, but not limited to, a text editor, c compiler and sometimes a window manager. This is also the level that contains the libraries used by the aforementioned applications.<sup>2</sup>

With so many available options, the choice of which operating system to use may seem daunting. From the earliest stages of this project, it was noted that Linux provided three functional pillars upon which the software could be created. These are the development environment, IO management and concurrency handling. These will now be described in detail.

### **3.2.1 Development Environment**

Linux provides a fully functional C development environment through its GNU C compiler, *gcc*, and the GNU debugger, *gdb*. *gcc* is a very widely utilized compiler that comes preinstalled with most Linux distributions. *gdb* is the well documented debugging counterpart to *gcc*. These tools, coupled with the text editor, *vim*, allowed the design team to begin developing software easily and immediately for use with the Linux operating system.

### **3.2.2 IO Management**

Linux, like its influential predecessor UNIX, virtualizes most IO devices as files. This would allow the development team to interface the various peripherals of the system with relative ease.

### 3.2.3 Concurrency Handling

As was stated earlier, the original purpose of the operating system was to manage the execution of multiple programs running on the same system. Linux is not an exception in this regard. The Linux kernel has the ability to manage the concurrent execution of thousands of threads. Furthermore, the GNU C Library contains a POSIX compatible set of functions to control threads and processes within the OS. The advantage here is that the system's IO can be handled separately from the sound synthesis processes.

## 3.3 Sound in Linux & ALSA

In the early days of Linux, sound was supported from within the kernel using a stack, analogous to the OSI stack, called OSS (open sound system). When PC sound cards began to become increasingly complex, the variation in architectures led to increased difficulty in interactively developing supportive stable kernels. At some point prior to the 2.4 kernel release, it was decided that sound was better supported in Linux as a device driver rather than through the kernel itself.

This led to the creation of the Advanced Linux Sound Architecture (ALSA). While the name may be deceiving as it seems to describe an *architecture* rather than a device driver, the term *ALSA* is colloquially used to describe the various modules used to support sound on a Linux system coupled with a software API used to interface the driver and, through that driver, the soundcard.

It should be noted that sound in Linux is, in reality, much more complex than users would expect. The functionality of ALSA goes far beyond what is expected of a normal device

driver in the sense that it has an API that can be used to control a vast array of soundcard features, transfer blocks of data in many ways, and even control features within the software modules themselves. ALSA also has what some may consider a virtual machine type feature, in that it can emulate a virtual soundcard that other software, including the legacy OSS, can operate on. Figure 3.4 below serves to illustrate the complexity that has become the software stack modern Linux distributions use to control sound.

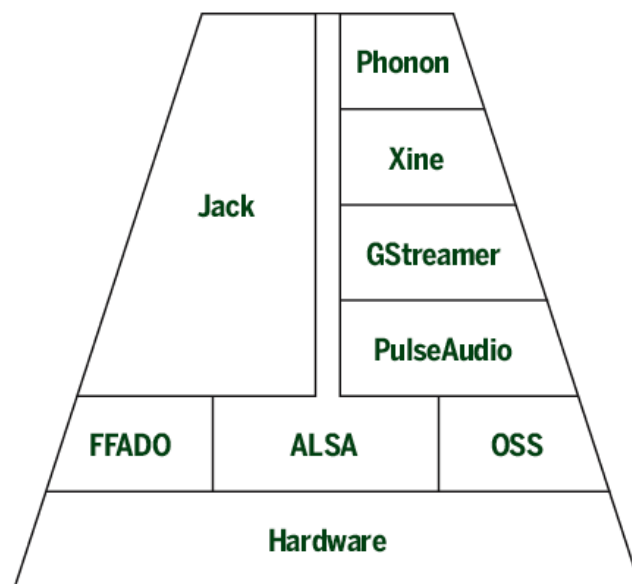


Figure 3.4: Various software packages used to control audio in Linux

As can be seen in the figure above, various applications make use of the ALSA API. OSS, described above, supports legacy applications by interfacing ALSA's virtual soundcard. *PulseAudio* is used to mix audio streams from various sources and sort them between ALSA and OSS. *FFADO* has the same function as *PulseAudio*, except it only accepts audio streams from firewire interfaces. *Jack* is used for scheduling between various applications with the goal of mitigating latency. *Gstreamer* and *Xine* provide support for proprietary codecs. Finally *Phonon* provides cross compatibility between applications built for the two

major families of Linux distributions, KDE and Debian. <sup>3</sup>

With so many programs handling sound in Linux, it may seem like a daunting task to choose the best way to handle sound output. For this project, the team chose to use ALSA. The reason for this choice is threefold.

- The ALSA API has a set of C libraries that can be used to control the soundcard and C was the chosen development language for this project's synthesis core.
- It is desirable in this application to have as little latency as possible from the time an input is given to the system and when the corresponding sound is heard. Having less layers of software abstraction between the synthesis software and the soundcard will mitigate this latency.
- Keeping embedded system implementation in mind, some distributions of Linux that are developed to run on embedded systems have very limited software installed by default. Its possible that users may not have all seven of the software packages described above available on their system, but the one that is most likely to be installed is ALSA, as it is the basis for the others. This helps to keep the software available to as many users as possible.

### **3.3.1 ALSA API**

As was mentioned above, ALSA comes standard with a software API in the form of a C Library that can be used to interface and control the soundcard. This API has many functionalities and makes ALSA highly configurable from the perspective of the application that makes use of it. The basic functionalities made use of within the scope of this project will be described herein.

Samples are transferred to the soundcard, using ALSA, in discrete length sets called *periods*. The API can be used to configure the length of these periods in samples or time depending on the requirements of the application. When configuring the period time, obviously the sampling rate of the card must be factored into the calculation. Thankfully the sampling rate can also be configured using ALSA.

ALSA can transfer periods to the soundcard in one of two ways. The first method is to have the application manually specify a location in memory corresponding to the start of the period. When this happens ALSA transfers the sample to the *end* of a hardware *buffer* from which the samples are read and played using the soundcard. The length of this hardware buffer is configurable and must be carefully tailored by applications desiring the lowest possible sound latency. The second transfer method is using a memory mapped region. In this method, ALSA automatically ensures congruency between the memory and the hardware buffer. The first transfer method was used in this project as this method requires less memory to be managed manually by the developer.

The way the period is written to memory must also be carefully maintained by the application. The most common way to manage this period is to have the samples stored analogously to how CDs store their audio data. That is, to have the right and left audio channels interleaved, each containing signed 16 bit samples and stored in little endian format. In ALSA documentation, one pair of left and right channel samples is generally called a *frame*, however this can be extended to mean multiple pairs or even sets of samples beyond just two channels. Once again, all these features are configurable through the ALSA API. For simplicity, the CD audio configuration described above is used in this project. Figure 3.5, below, is included to give clarity to the discussion above.

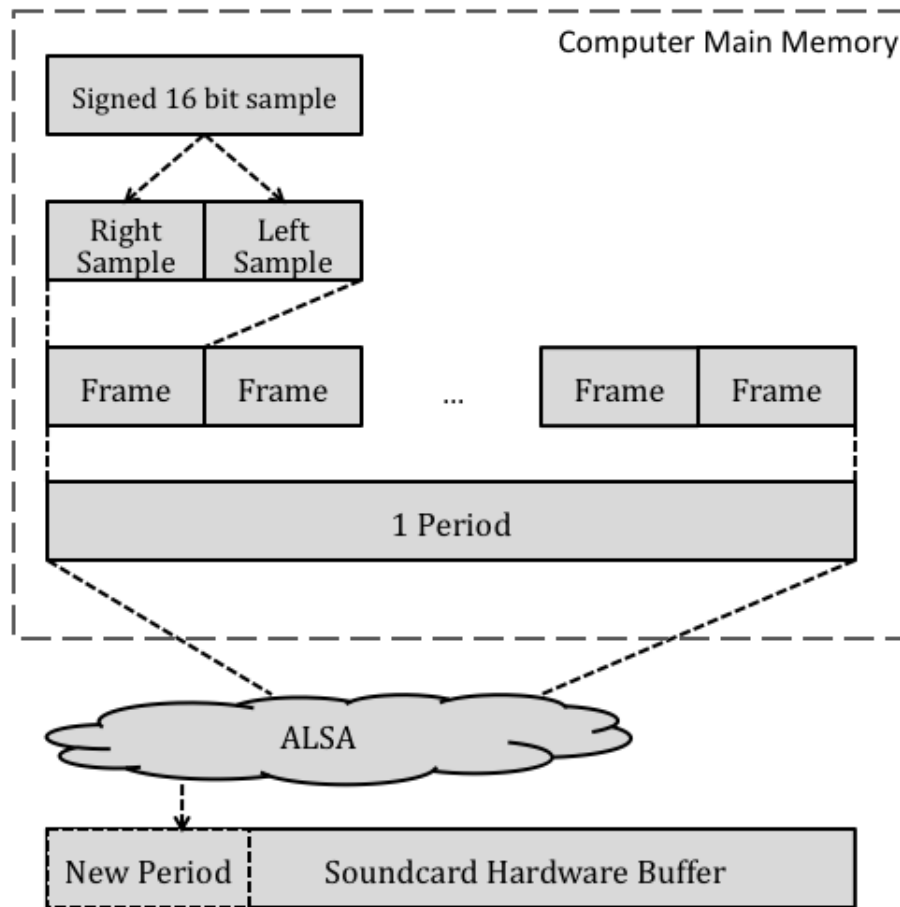


Figure 3.5: Memory Organization of samples to be passed to ALSA

### 3.4 Android Application

As previously mentioned one of the interfaces to control the synthesizer will be an Android application. To create an outline of the application design research was done on existing tablet and Smartphone piano applications. Two existing applications that were examined were the Perfect Piano app for Android devices and the My Piano App for Android devices. The following table outlines characteristics of each application.



Table 3.2: Comparison of existing sound synthesis applications

	<b>Perfect Piano</b>	<b>My Piano</b>
Number of Visible Octaves	6 - 20 White Keys	1 or 2 Octaves
Instrument Capabilities	Piano,Organ,MusicBox,Rhodes,Synth	11 Instruments
Capable of MIDI Control	No	Yes
Capable of Variable Note Strength	No	Yes

After reviewing the two apps some fundamental characteristics were identified. The number of visible octaves should be variable and range from one to two octaves at a time. Another key feature is multi touch which allows the user to play multiple notes at the same time. Both devices had the capability to select additional instruments which will be a feature also available on our device. The My Piano app is also a MIDI capable controller and capable of sensing note velocity. These are two characteristics which will also be implemented on our device. Both of the devices had external menus for adjusting the settings of the app. Our design will differ from these and have selectable options on the same interface screen as the keyboard. After reviewing both the Perfect Piano and My Piano applications the following conclusion can be made; our device will attempt to have the same or better sound quality as the Perfect Piano while having the functionality of the My Piano application.

## 3.5 Communications

### 3.5.1 Communication Protocols

When considering communications involved in the SynthBox, it was realized that there were a multitude of distinct and different communication paths. Wireless communication to an external controller, communication of notes and commands by the user, and the interpretation of received commands by the synthesis unit were all identified as potential connections. Based on this, research was performed in order to develop a comprehensive communication strategy that would attempt to standardize and coordinate effective flow control between the various units and modules of the SynthBox in a way that would ensure reliability, compatibility, and minimize coding requirements as would be needed if several different protocols were used.

Preliminary research began with the analysis of several current digital music synthesizers to ascertain whether a well developed or accepted standard already existed. At this point, it was discovered that there was already a well-defined and ubiquitous communication standard for musical instruments known as the Musical Instrument Digital Interface standard, or MIDI.

MIDI was developed in the late 1970's based on interest of several musical instrument manufacturers in developing a universal standard to allow musical devices to communicate with each other, and the first full MIDI specification (version 1) was published in 1983. Since then, MIDI has evolved and developed, but is still supported by virtually every digital music device manufactured today.

This wide acceptance lead the design group to seriously consider MIDI as the basis for our communication protocol. In addition to it being well supported, MIDI is a complete hard-

ware and software specification, each of each are easily implemented with modern technology. Additional details about the exact specification can be found in Development section of the document.

However, although MIDI standardizes wired communication between devices, it was lacking in two areas for the SynthBox: wireless communication is not considered, and MIDI also does nothing to actually synthesize or generate musical notes; it merely specifies which note should be played at what time, as well as volume, pan, and other common sound effects. This leads to additional consideration of a communications scheme for the SynthBox that could incorporate wireless communication.

In light of the fact that wireless communication with a common smart phone was desired, research into what forms of communication are possible with such devices was performed.

Investigation into the wireless capabilities of common devices such as an Android Smart-Phone (LG Optimus One), the Apple iPad, and an Android-based tablet (Nexus 7) revealed that two forms of communication were well suited to the SynthBox: Wireless communication in frequencies of 2.4 Ghz and above (commonly referred to as Wi-Fi or standard wireless internet), or Bluetooth, a protocol for creating personal area networks. Another form of wireless communication is becoming popular with smartphones and tablets known as Near-Field Communication or NFC, but as it has severely limited range (in the order of a few centimetres), it would not be well suited to the SynthBox.[1]

In researching the capabilities of both wireless and Bluetooth communication, it was discovered that Bluetooth has a well developed profile called Serial Port Profile which allows two Bluetooth transceivers to emulate a common serial COM port in a wireless fashion, with little software overhead to the user. Essentially, by configuring Bluetooth modules in this

fashion, a user can serially transmit data to the Bluetooth module, and it will arrive at the other end without any need for custom modulation, data encapsulation.

Wireless communication in the typical internet band, however, is more complicated and would necessitate additional software to properly transmit data. As MIDI communication is built upon standard asynchronous serial communication, the use of bluetooth with serial port profile was deemed to be an appropriate fit with the rest of the system.

### **3.5.2 MIDI Hardware**

As seen, the electrical specifications are simple and easily implemented. The important components to note are the double buffer on the MIDI out and MIDI THRU ports, as well as the opto-isolation on the MIDI IN port. These components are in place in order to prevent ground loops and protect the microcontroller that is interfacing with the MIDI hardware if a user were to plug a MIDI connector into the incorrect port.



MIDI messages are to be transmitted using an asynchronous 8-N-1 transmission format at a baud rate of 31250. It is noted that while the software implementation of this is simple, 31250 is a non-standard baud rate when considering a computer COM port, and so some conversion may be necessary when interfacing the MIDI hardware to the system.

### **3.5.4 MIDI USB Interface**

Although MIDI has a well defined hardware interface, the relation of this hardware to the somewhat outdated Serial communication means that very few modern computing systems can support a MIDI connector natively. As the project was eventually abstracted from the hardware and intended to run a personal laptop, it became prudent to consider a way to communicate with a MIDI device on a physical level.

After some research, it was discovered the most common way to perform this task is with a USB MIDI interface such as the M-Audio USB MIDI Uno[3]. This type of interface is required to connect virtually any MIDI compliant device that still uses a 5-Pin MIDI connector to a personal computer( some newer MIDI controllers offer a USB connection built in).

## 4 Final Design

### 4.1 Synthesis

As was mentioned in the initial research section, it was decided that digital waveguide synthesis would be used in order to synthesize the instruments desired by our customer. Digital waveguide synthesis, or DWS as it will be referred to in the rest of this section, is a synthesis technique that was developed at Stanford by a man named Julius O. Smith III. Smith maintains a website that describes many aspects of DWS. Much of the theory in the sections that follow can be found on this website and so the website is referenced in the bibliography [XX]. If a reference is not made to a specific source, it can be assumed that reference was made to this website.

DWS is a method for physically modeling an instrument using digital signal processing (DSP) techniques. At the heart of this method is the digital waveguide, which is an efficient model for physical media that can support the propagation of waves at acoustic frequencies. Therefore, the digital waveguide itself is what supports the time varying wave that is responsible for the output sound that is heard.

In order to provide the reader with enough detail about DWS and how the instruments included in the synthesis core for this project were created, this section is broken down into seven further sub sections. The first section will deal with the digital waveguide itself. This section will provide a brief derivation of the ideal digital waveguide, followed by an introduction of the DSP techniques used to implement the non-idealities that make synthesized instruments using DWS sound more realistic. The next 5 sections will deal with each of the 5 instruments that were included in the synthesis core. Each instrument had a unique model and these subsections will highlight how each model differed from the others. This

will generally include a description of the excitation as well as any other sound radiation that made the instrument sound unique. The goal of these sections will be to provide the reader with enough information about each model so that the structure of the Matlab scripts that implement each model can be thoroughly understood. Therefore, these sections will not include specific values used in the final models, but rather explain the general structure of each model.

The last sub section will briefly discuss how each model was tested and explain the tuning procedure used for each instrument.

#### **4.1.1 The Digital Waveguide**

As was mentioned above, the digital waveguide is at the heart of any DWS model. It allows for efficient modeling of a media that supports a traveling wave from an initial excitation. In this project, two mediums that support waves at acoustic frequencies were used. The piano, harpsichord and guitar all used strings in order to support acoustic waves. While the flute and the clarinet, the woodwinds, used hollow tubes, which will also be referred to as bores in the sections that follow. Although a string allows for mechanical waves to propagate along it and a bore allows for acoustic waves to propagate along its length, the digital waveguide itself does not change. The only thing that needs to change is how the excitation is applied and how the sound is actually transduced into something that can be heard. This is because while the sound from a woodwind comes directly from the bore of the instrument, which is already supporting an acoustic wave, a string supports a mechanical wave. So this mechanical wave first needs to be transduced into an acoustic wave that can be heard by a listener.



The digital waveguide itself is based on the simple traveling wave equation. The traveling wave equation, which is shown in equation XX, is used to describe the motion of a 1-dimensional wave in both time and space. In this equation,  $c$  represents the speed of propagation of the wave in the medium,  $x$  represents the distance traveled by the wave in the  $x$  direction, and  $y$  represents the transverse displacement of the wave. A diagram that graphically shows this simple wave is included in figure 4.1.

$$\frac{\delta^2 y}{\delta x^2} = \frac{\delta^2 y}{c^2 \delta t^2} \quad (1)$$

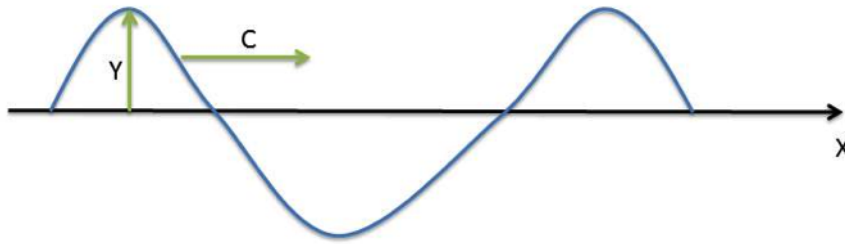


Figure 4.1: Traveling wave

The variable of interest in this equation is the transverse displacement of the wave,  $y$ . The solution to this differential equation yields the formula shown in equation XX below.

$$y(x, t) = y_f(x + ct) + y_r(x - ct) \quad (2)$$

This solution is interesting because it shows that at any one point, the transverse amplitude of the wave at a specified distance and time is equivalent to the superposition of a reverse and forward traveling wave. This indicates that there is not only one wave traveling in the medium, but two. In the case of a musical instrument, this reverse traveling wave is caused by reflections at the terminations of either the string or the bore. These reflections are due to the fact that not all of the energy in the waveguide is transferred out of the instrument.

Any musical instrument has a waveguide that is of finite length, so the solution above could be used to describe the amplitude of the wave at any point on the waveguide as well as at any time provided an excitation has been applied. Also, assuming a perfect reflection and no loss, the reverse traveling wave can just be expressed in terms of an inverted and time delayed version of the initial wave. However, this solution is in continuous time and space, which cannot be implemented digitally without using some sort of numerical method, which is time consuming. So, the next step is to discretize the solution to the traveling wave equation. The discretized version of the solution is shown in equation XX below.

$$y[nX, mT] = y_f[n + mT]y_r[n - mT] \quad (3)$$

The discrete version of the solution again shows that the amplitude of the wave is the sum of the reverse and forward traveling waves. However, now both the position and the time samples are discrete values. The time step in this equation can be related quite easily to the sampling frequency at which a model would run,  $\frac{1}{T_s}$ . However, the discrete position is slightly more difficult to quantify. So, a clever way to define the distance between each discrete spatial sample is to choose the distance between these samples to be the distance that the wave would travel in one sampling interval. By defining the discrete position using the method above, the discrete solution can simply be implemented using two digital delay lines for the forward and reverse traveling waves. After each sampling instance, the values in the delay lines can simply be shifted one delay forward; simulating the wave moving the distance it would cover in  $\frac{1}{T_s}$  seconds.

In this implementation, the upper delay line propagates the forward traveling wave and the lower delay line propagates the reverse traveling wave. In the ideal case, the upper and lower delay lines can be linked using perfect reflections. After each sampling instance, the delay lines are simply updated, or shifted. This update simulates the wave traveling down the

string or bore. This implementation is shown in figure 4.2 below.

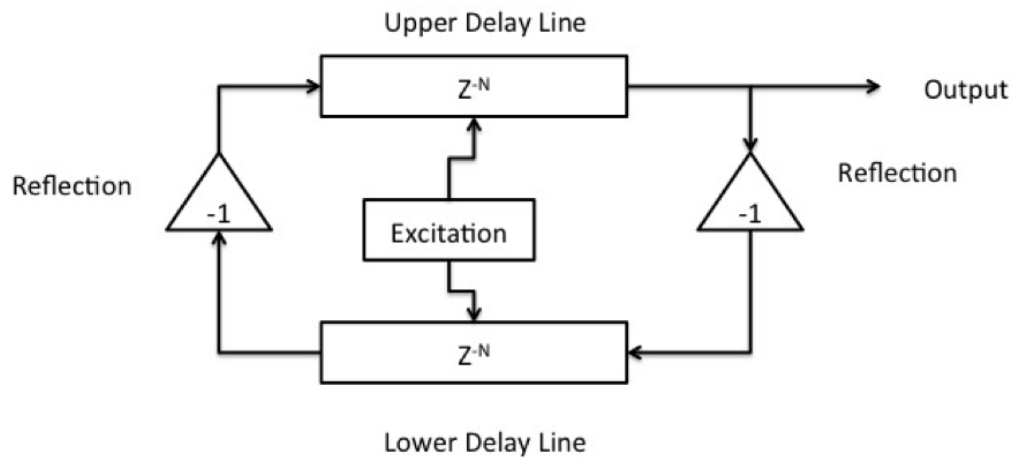


Figure 4.2: Delay Line model

In this diagram, the terminations of the waveguide are modeled as perfect reflections, or as if they had infinite impedance. The excitation is also being input into the upper and lower delay lines at a position that is between 0 and  $N$ . Where 0 represents the start of the waveguide and  $N$  is the end. This model provides a very simple example of how a digital waveguide works. However, the main problem with this model is that it is based off of the ideal wave equation. The ideal wave equation does not account for losses in the medium or for dispersion of the wave. These non-idealities are what give each instrument a unique sound, and that is why the digital waveguide shown above would not produce a very real sound.

In order to implement these non-idealities, simple digital filters can be used. This is the case because the model in figure XX above is a linear system. Most waveguides in instruments show a frequency dependent loss and this loss generally resembles a low pass filter. This allows lower frequencies to propagate through the waveguide for longer periods of time while the higher frequency components are quickly attenuated. This loss could be implemented

between each delay element, but a better way to do it is to clump these losses into one single filter and place it at the end of the waveguide. Most literature on the subject also suggests that a simple first order low pass filter can be used to simulate the loss in the waveguide. This first order filter is shown in equation XX below.

$$H_{LP}(z) = g \frac{1 + a_1}{1 + a_1 z^{-1}} \quad (4)$$

This filter is fairly simple, but accurately represents the frequency dependent loss in most instruments. The bode plot for this low pass filter with a gain,  $g$  value, of 0.75 and a pole,  $a_1$ , at -0.24 is shown in figure 4.3 below.

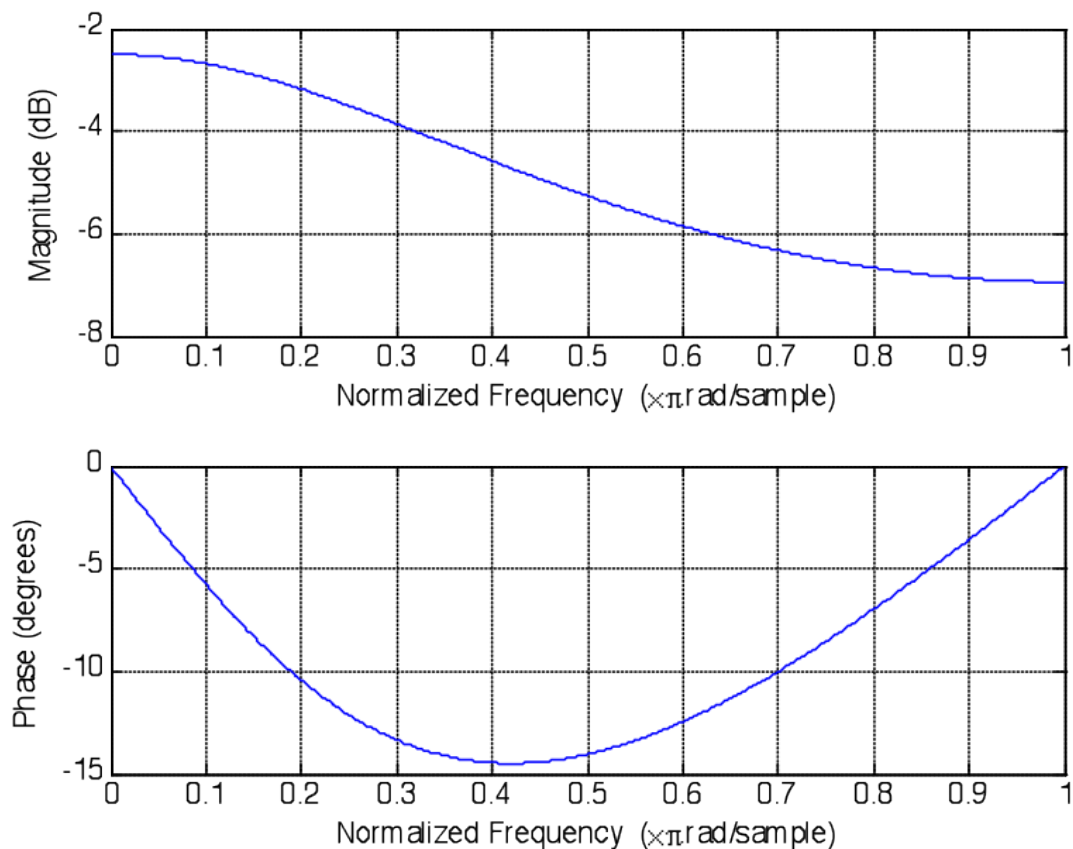


Figure 4.3: Frequency response of low pass filter

The filter specified above is the low pass filter that was used to implement the loss in the strings for the lower notes of the piano. These plots use a normalized frequency and show that there is not a drastic difference between the gain in the pass band and the cut off region. In fact the difference is only about 5 dB. However, what should be noted is that the output does not only pass through this filter once. Once the excitation has been initialized, the wave in the digital waveguide constantly passes through the upper delay, is filtered and reflected, passes through the lower delay and then is again reflected but this time back into the upper delay. Each time a round trip is completed, the wave will have been filtered once. With respect to the sampling frequency that the model is running at, the length of the delay is relatively short. So even if the model is run for only one second, the signal in the waveguide could pass through this loss filter 80 or more times. So, the effect of this filter is cumulative and the difference in gain between higher and lower frequencies becomes much more evident.

One other thing that needs to be noted is that this filter is used to implement loss, but due to its nature it also implements a phase delay. This phase delay can be seen in the second plot of the phase with respect to frequency in figure XX above, and shows that in the middle range of the normalized frequencies there is a noticeable delay. This causes problems because in order to get an accurate output frequency from the instrument the delay needs to be just right. The solution to this is to add variable length delay filters that can compensate for this and effectively tune the note to the proper frequency. This will be discussed more in the piano section as this was the only instrument in which an IIR tuning filter was implemented.

The last non-ideality that needs to be accounted for in the digital waveguide is dispersion. Dispersion is the phenomenon that causes the speed of propagation of a wave to be frequency dependent. To bring this back to the digital waveguide, this means that the length of the upper and lower delay line need to vary for different frequency components in the signal. To do this, an all pass filter can be used. An all pass filter does not affect the magnitude of

the signal, but only affects the delay presented to the signal. The general form for a second order all pass filter is shown in the equation below.

$$H_{AP}(z) = \frac{a_2 + a_1 z^{-1} + z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \quad (5)$$

The difficult part about this filter is that the variables  $a_1$  and  $a_2$  are harder to quantify. In order to be able to choose these values, first the inharmonicity constant,  $B$ , needs to be defined for the waveguide. The formula below shows how the inharmonicity constant can be calculated for a string.

$$B = \frac{\pi^3 E r_{core}^3}{4L^2 T} \quad (6)$$

In this equation,  $E$  is Young's modulus for the string,  $r_{core}$  is the radius of the string,  $L$  is the length of the string and  $T$  is the tension of the string. Once  $B$  is defined for the waveguide, a method laid out by Rauhala and Valimäki [XX] can be used to determine the values for  $a_1$  and  $a_2$ . This method is interesting because it relates the digital delay required to the physical characteristics of the string, which helps to make the design of this filter slightly more intuitive. The design steps used to determine the parameters for the filter will not be discussed fully in this report. However, the basic steps will be discussed briefly.

This method uses the inharmonicity constant,  $B$ , of the note as well as the note number itself to determine the frequency dependent delay that needs to be introduced,  $D$ . Once this  $D$  value is found the filter coefficients are calculated using the equation below.

$$a_k = (-1)^k \frac{N}{k} \prod_{n=0}^N \frac{D - N + n}{D - N + k + n} \text{ for } k = 1, 2, \dots, N \quad (7)$$

In this equation  $D$  is the delay that was calculated and  $N$  is the order of the filter, which for this filter design is two, as it is a second order filter. Once these coefficients are calculated the

all pass filter can be implemented using equation 7 above. The Bode plot for the dispersion filter designed using the method above for the A4 string of the piano is shown in the figure below.

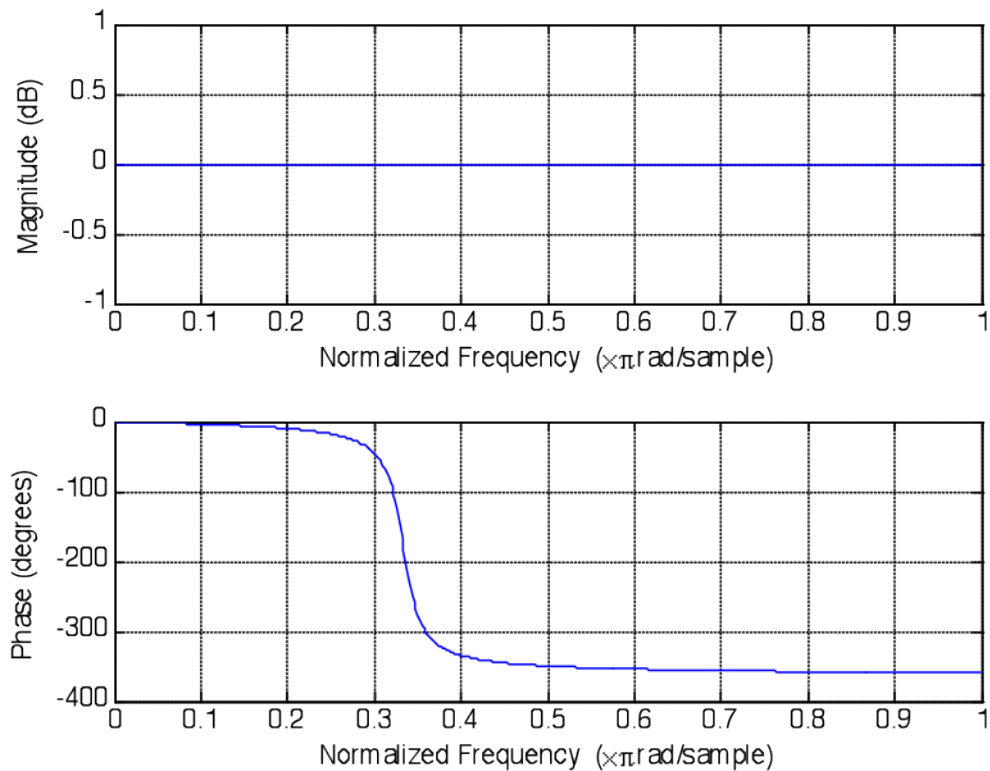


Figure 4.4: Filter Frequency response

The figure above shows that the magnitude of the response is always 0 dB, or a gain of 1. It also shows that the phase delay at the lower frequencies is not very big. This means that for the most part for the lower frequencies the speed of the wave remains fairly constant.

Although most literature includes a dispersion filter and a loss filter, from auditory testing it was found that the dispersion filter does not greatly affect the output sound quality for most

instruments. For the woodwind instruments it was extremely difficult to hear a difference between the models that had and that did not have dispersion filters. So, the dispersion filter was only included in the piano model where it was determined that there was an audible difference for most of the notes.

When the dispersion filters were included in the digital waveguide for the piano, there were two different forms. For the lower notes, MIDI 1 to 44, 4 cascaded all pass filter were used. For the high notes, MIDI 45 to 88, only one all pass filter was used. In the paper written by Rauhala and Valimaki that outlines the method for designing a piano dispersion filter the use of two configurations is justified by the fact that one design could not be used for the whole range.

Figure 4.5 below shows the final diagram for the digital waveguide with the non-idealities included.

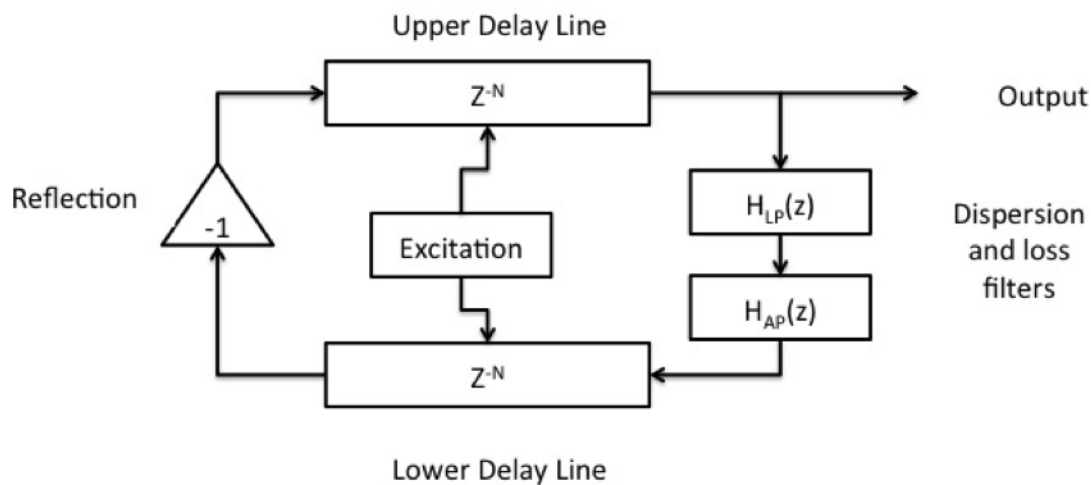


Figure 4.5: Physical model with non-idealities

The digital waveguide is at the core of every instrument that was created for the synthesis



core. However, there are two more very important aspects of each instrument that need be identified and modeled before an accurate synthesized sound can be attained. Those two things are the excitation and the sound radiation from each instrument. A general, simplified structure that encompasses each DWS model is shown below.

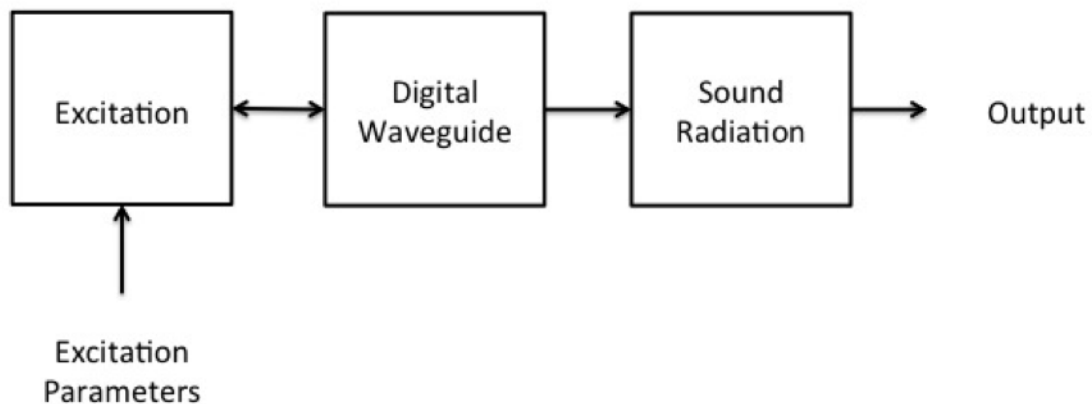


Figure 4.6: Cascaded stages of model

In the sections below, the excitation used to create the signal that enters the digital waveguide for each instrument model will be explored. The method of sound radiation will also be explained for each instrument. As mentioned at the beginning of this section, the stringed instruments require some way of transducing a mechanical traveling wave to a sound wave. Therefore, the body of each instrument becomes extremely important in the output sound that is heard. For woodwinds, this sound radiation is not quite as important, but there are still some aspects of how the sound is radiated that need to be explored.

### 4.1.2 Piano Model

The piano was the most complex model that was created for this project. It involved a different model for each note, and some notes required more than one string. This meant that for a single note, sometimes up to 3 digital waveguides were used. Another aspect of the piano that made the model more complex was modeling the body and soundboard of the instrument. The body of the instrument contributes greatly to the quality of the output sound, so a good body model was required.

The excitation of a piano is one of the simpler instrument excitations to model because there is only one variable that needs to be varied, the initial velocity of the hammer. A piano is excited through the use of a key. This key is connected to a wooden hammer covered in felt and when the key is pressed the hammer is given a velocity that causes it to hit the string. The force that the hammer applies to the string at any moment in time is calculated using the formula in equation 8 below.

$$F_H = Kx^P \tag{8}$$

The image below in Figure 4.7 shows this compression visually.

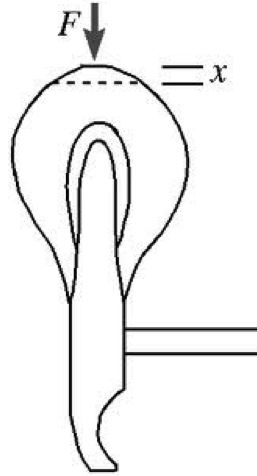


Figure 4.7: Piano Compression

In this equation  $K$  is the felt stiffness constant, which dictates how much force is applied for a certain displacement of the felt. The variable  $x$  represents the displacement of the felt and  $p$  is the stiffness exponent. This equation is used only when the amount of felt deflection is positive, if the deflection is zero or negative then  $F_H$  is set to zero.

In order to determine the displacement of the felt, the string and hammer velocities need to be related. The current hammer velocity is known and the current string velocity is simply taken from the digital waveguide. So to relate the two, Newtons laws are used and the following equation can be derived. The variables used in this equation can be visualized using figure 4.7 above and figure 4.8 below.

$$\frac{dx}{dt} = v_{hammer} - v_{string} \quad (9)$$

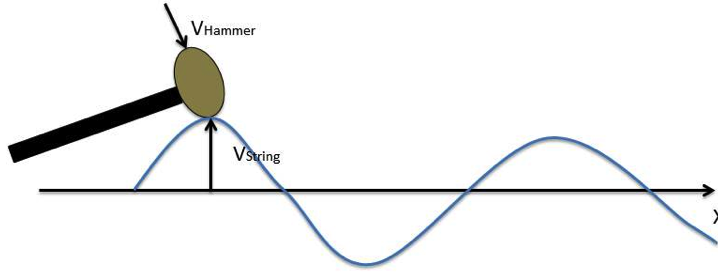


Figure 4.8: Hammer interaction with piano string

This equation provides the derivative for the amount of compression of the felt. In order to get the actual value for the compression, numerical methods were used to find the integral of the derivative of the compression. Once the felt compression was found, the hammer force applied to the string at that sampling instance could be calculated using equation 10.

The last problem became determining the new hammer velocity based off of the current hammer force. For the first sampling interval, the hammer velocity was simply the velocity of the hammer caused by the keystroke. However, for every sampling interval after, the hammer velocity varied. So, to recalculate the hammer velocity Newtons second law was used which is shown in equation 11 below.

$$\frac{dv_{\text{hammer}}}{dt} = \frac{F_{\text{hammer}}}{m_{\text{hammer}}} \quad (10)$$

Again, this formula provides the acceleration of the hammer and the variable of interest is the velocity of the hammer. So, to find the velocity simple numerical integration was used. For both of the numerical integrals that were taken in the excitation calculation, the trapezoidal rule was used. This is summarized in equation XX below.

$$\int_a^b f(x)dx \approx (b-a)\frac{f(b)+f(a)}{2} = \frac{T_s}{2}(f(b)+f(a)) \quad (11)$$

In order for these numerical integrals to remain stable, the excitation calculations had to be run at twice the sampling frequency of the rest of the model. Generally, if the change in the function 11 is too great between sampling intervals, then the numerical integral output will be large, causing the approximation to go unstable. Therefore, running the hammer calculations at twice the sampling frequency allowed the output of the numerical integrals to always remain stable.

The last step with the excitation was to implement the hammer force into the digital waveguide. To do this, scattering junction theory was applied. The first step in using scattering junction theory is to sum all of the incoming forces at the point of contact. To do this equation 12 below was used.

$$F_T = F_H + \sum_{NumStrings} 2Zv_f + 2Zv_r \quad (12)$$

In this equation  $Z$  represents the characteristic impedance of the string and  $v_r$  and  $v_f$ , represent the velocity of the string at the point of contact for the wave in the reverse and forward direction, respectively. In this implementation of the waveguide, the displacement of the string was not the variable that was propagated in the digital waveguide; instead the velocity of the wave was the variable that was propagated. It can be shown that the solution to the ideal wave equation also holds for the transverse velocity on the string, so the digital waveguide can be used without any modifications.

Once the total force was calculated, the last step was to reintegrate this force back into the waveguide. To do this the equations below were used.

$$v_f = \frac{L}{Z_{total}} - v_r \text{ and } v_r = \frac{L}{Z_{total}} - v_f \quad (13)$$

It should be noted that  $v_r$  and  $v_f$  in the equations above represent the forward and reverse wave velocities sampled at the point of contact and  $Z_{total}$  is the sum of the characteristic impedances for each string in contact with the hammer. These new velocities are then implemented into the forward and reverse waveguide at the point of contact, replacing the values used to calculate the hammer force.

The hammer-sting interaction causes the creation of a time varying waveform, however this waveform does not have a specified frequency. To specify the frequency, the length of the delay line is varied. A simple formula that uses the desired frequency and the sampling frequency to determine the required delay is shown below.

$$M = \frac{f_s}{2f_{desired}} \quad (14)$$

This equation calculates the required length for the upper delay line. However, there are two problems with this calculation. The first is that  $M$  is not always an integer value, and only integer delays can be implemented using delay lines. The second is that the filters used to replicate the non-idealities of the string also add delay. So, in order to have an accurate output frequency all of these delays need to be taken into account. To do this a second all pass filter is added that implements a non-integer delay.

The upper delay line is set to the rounded value of  $M$  and all of the filters are added to the lower delay line. So, now the total delay from the filters needs to be calculated and then the length required for the lower delay line,  $N$ , can be found by subtracting the filter delay from  $M$ . The last step is to round  $N$  down to the nearest integer value and design an all pass filter so at the frequency of interest the delay corresponds to required decimal delay.

To determine the delay of the discrete time filters, the filters are evaluated at  $z = e^{j\omega}$ . In this equation,  $e$  is the natural logarithm and  $\omega$  is the frequency of interest. The delay needs to be exact at the frequency of interest, so the filters are evaluated at  $\omega = 2\pi f_{desired}$ .

$$D = \text{angle} \frac{f_s}{\omega} \quad (15)$$

With this delay calculated for each filter the length of the lower delay line can be found by subtracting these delays from  $M$ . Once  $N$  is determined, the integer part of  $N$  is implemented using a delay line and the non-integer part is implemented using a second order all pass filter. This filter is the same as the dispersion filter described in the digital waveguide section and is therefore designed using the same method. The only difference is that now the delay,  $D$ , is defined as the decimal portion of  $N$ .

With the tuning filter and excitation implemented, the digital waveguide model itself is complete. The output of the string can simply be taken from the end of the upper delay line. The final model for a two-string note is shown below in figure 4.8.

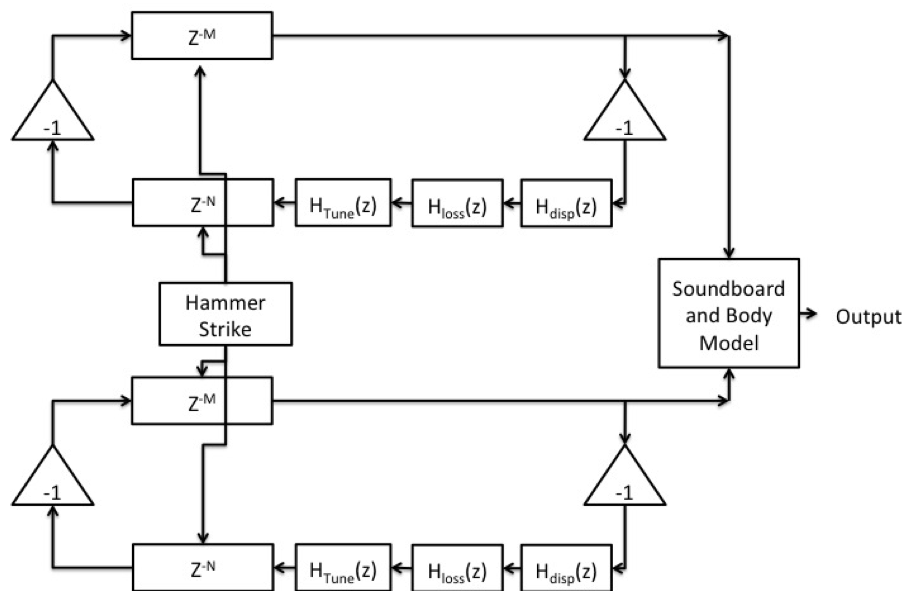


Figure 4.8: Block diagram of piano model

From the diagram above it can be seen that the output of each string is fed into the soundboard and body model before the sound is output. The reason for this is that the soundboard and body contribute greatly to the sound of the output of the string. The soundboard is responsible for actually transducing the mechanical waves on the strings into acoustic waves; while the body is responsible for sustaining the notes and adding resonance at certain frequencies.

For this project, the soundboard and body model was implemented as a feedback delay network. This method was developed by Balazs Bank and presented in his master thesis [XX]. The feedback delay network presented in this paper consisted of 8 delay lines of varying length. In order to simulate the frequency response of the soundboard low pass filters with varying pass bands are placed at the end of each delay line. The output of the delay lines are summed and this is the sound that is output to the speakers. The outputs of the delay lines are also feedback to the inputs of the delay lines using a feedback matrix A. Therefore,



the input to the delay lines is actually the sum of the outputs of all the string and a scaled version of the output of each delay line. This method is summarized in the diagram below.

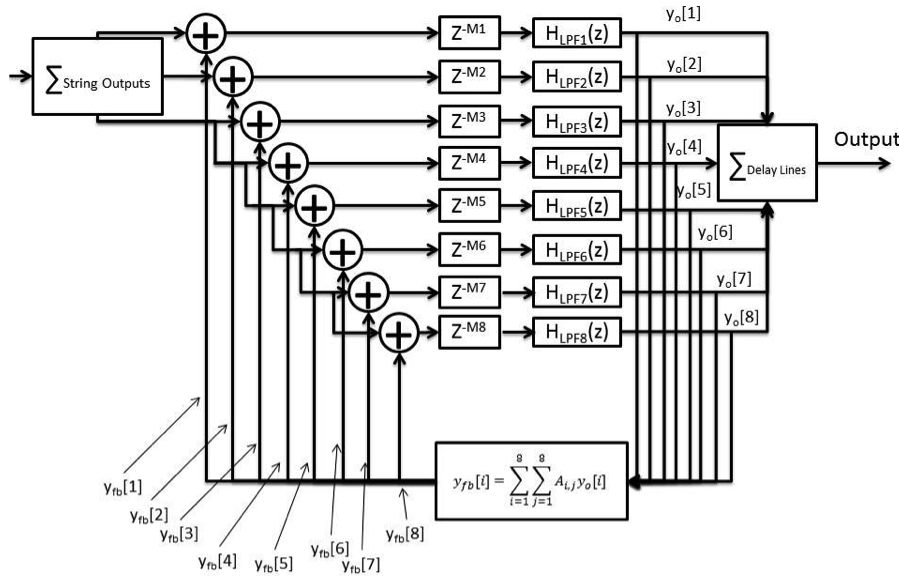


Figure 4.9: Block diagram of piano model including many delay lines

This model was fairly complex and the values used to implement it were taken directly from Banks master thesis. Although it was complicated, the soundboard and body model greatly improved the sound quality and gave it a much richer sound.

With the soundboard and body model complete, the piano model was also complete. As was mentioned at the beginning of this section not all of the notes had the same structure. Notes below MIDI 31 used only one string and ran at a sampling frequency of 22050 Hz. Notes below MIDI 41 used two strings and ran at 44.1 kHz. Notes under MIDI 71 used three string and ran at a sampling frequency of 44.1 kHz. Finally, notes under MIDI 108 used three strings and ran at 88.2 kHz.

The sampling frequency needed to be varied in order to ensure that the models were stable. With the higher notes, at lower sampling frequencies the length of the delay lines is very small. This causes rapid feedback to the hammer model, which quickly causes instability. The lower delay line lengths also cause the waves to get filtered many times, so the output is attenuated extremely quickly. So even if the model is stable, the output is short and quiet.

The final implementation of the piano had models for notes ranging from A0 to G6, MIDI 21 to 91. These models could take a number of different input velocities in order to vary the initial force with which the hammer hit the string. Finally, a damper was added that stopped the note from playing after the user removed their finger from the key.

As mentioned above, the piano model was the most complex, but it also gave the most lifelike output sound when compared to a real piano, so the complexity of the model was justified by the end result. The Matlab script for each note range and the accompanying structure for each note can be found in the electronic appendix under Piano.

### **4.1.3 Harpsichord**

The harpsichord is physically similar to a piano. Each note is initiated through the depression of a key and the excitation is connected to a string. There is a separate string for each note and the sound is radiated through the use of a soundboard with some resonance being added by the body. Where the two instruments differ is in their excitation. When a key of a harpsichord is pressed it initiates the movement of a plectrum. The plectrum is a device that simply takes the movement of the key and plucks the string. So the only real difference between the piano and the harpsichord is that one uses a hammer and the other uses a pluck to excite the string.

Because these two instruments are so similar, the harpsichord model used the exact same structure as the piano model shown in the sub section above in figure XX. The only difference is that the excitation was changed from a hammer to a pluck. To do this, a very simple method was used. Instead of using feedback from the string and initial key press velocities, the excitation was made to be a random input. This may seem slightly odd, but a number of older synthesis methods used random noise to imitate plucks. The random noise was simply initiated when a key was pressed and was scaled so as to excite the waveguide with enough force. The scaled random noise was directly input into the waveguide. In order to avoid instability, the random noise sequence was set to be as long as the sum of the upper and lower delay lines. After this length was reached, the input was simply set to zero.

The resulting harpsichord model had a distinct pluck sound and the body and soundboard model made the sound more rich. However, the output did not quite have the same sound as a real harpsichord. The pluck was slightly dull and the soundboard and body model were a little bit off. This is to be expected though as the model was not varied at all. With variation to the original model the harpsichord could be made to sound much more like a real harpsichord. Unfortunately there is not huge body of research for the harpsichord, so this is quite a difficult task.

The final harpsichord model used the same string models as the piano and so it also had the same range from A0 to G6. There was no velocity sensing because a real harpsichord does not vary the pluck strength based on key velocity. Again, a damper was added to dampen the note back to zero when the player removed their finger from the string. There are no harpsichord specific Matlab scripts as the harpsichord uses the piano string models, and so nothing needed to be tuned using Matlab.

#### 4.1.4 Clarinet

The clarinet was the first woodwind instrument that was modeled. The woodwind models differ from the stringed instruments above in the fact that only one model was used, and parameters of the model were simply varied to create different notes. Unlike the piano and harpsichord which had a different model for every string. To model a woodwind instruments, tone holes were added to the digital waveguide structure. These tone holes could be placed in two positions, either opened or closed.

In the open position, a tone hole presents a large impedance to the pressure wave propagating in the bore of the musical instrument. So, some of the traveling wave is transmitted while most of the wave is reflected. This can also be visualized as a pressure node where the pressure must be low, causing the wave to reflect. Therefore, the first open tone hole from the excitation generally dictates the period of the standing wave in the bore, or the frequency of the note being played. When the hole is closed, or covered, then the impedance presented by the tone hole is greatly reduced. This allows for most of the wave to be transmitted while only some of the wave is reflected.

So the question becomes how can the tone hole be accurately modeled. There is some literature on the subject, however a comprehensive report on modeling woodwinds instruments is a PHD thesis by Gary Scavone [5]. Most of the figures presented below come from this document. In the thesis, two methods are proposed for modeling the tone hole in the bore of a woodwind instrument. The first is to use a two-port scattering junction, as shown in figure 4.10 below.

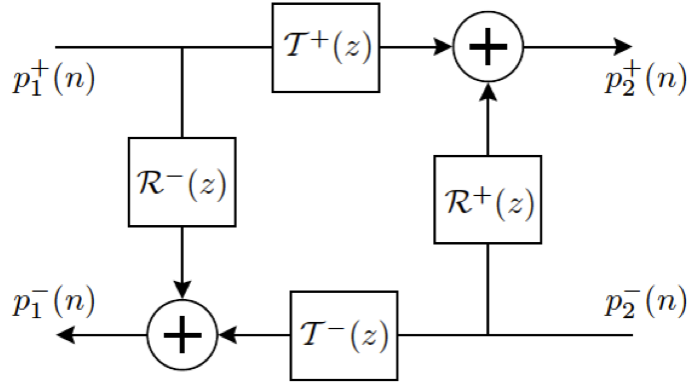


Figure 4.10: Two port scattering junction used in clarinet model

In this implementation there is transmittance and reflectance of both the forward and reverse traveling pressure waves. The amount of transmittance and reflectance is modeled through the use of second order filters that are derived from the physical parameters of the tone hole itself. In order to change the tone hole from the open to closed position, the reflectance and transmittance filters for the open position are replaced with filters for the closed position.

The second method is to use a three-port scattering junction. In the two-port implementation the tone hole is represented as an impedance. In the three-port implementation, the tone hole is actually treated as its own separate waveguide with transmission and reflectance characteristics at the end of the tone hole, modeling whether it is opened or closed. A visual representation of this model is shown below in figure 4.11.

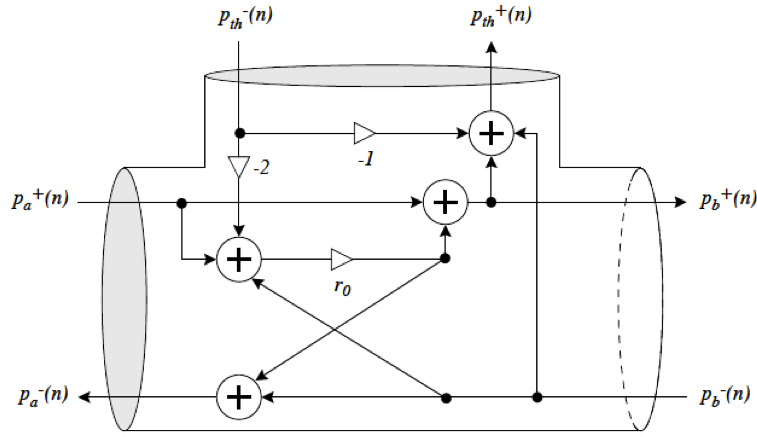


Figure 4.11: Visual representation of Clarinet model

The flow diagram, shown in the figure above, implements the scattering properties of a 3-port network. The unknown in this model is how to relate the forward and reverse traveling wave in the digital waveguide representing the tone hole. To do this a reflectance filter is implemented. The reflectance filter is implemented as an all pass filter that uses the bilinear transform to relate continuous time implementation to the discrete time implementation. Equation 16 below implements the first order all pass filter (the reflectance filter), and equation 17 shows how the variable  $a_1$  is calculated for the all pass filter.

$$R_{th}(z) = \frac{a_1 - z^{-1}}{1 - a_1 z^{-1}} \quad (16)$$

$$a_1 = \frac{t\alpha - c}{t\alpha + c} \quad (17)$$

In equation 17,  $t$  represents the height of the tone hole,  $c$  represents the speed of sound and  $\alpha = 2f_s$ . When the tone hole is closed, the value of  $a_1$  is calculated based off of the physical parameters of the tone hole. When the tone hole is open  $a_1$  is set to zero, this causes no reflectance to occur which means that all of the pressure that enters the tone hole is lost.

For the woodwind instruments in this project the tone holes were modeled using the 3-port method. This was done mainly because some implementations of the 2-port tone hole caused the model to go unstable, while the 3-port model always remained stable.

Implementing the tone hole model into the digital waveguide involved two main steps. The first was to specify the diameter and height of the tone hole as well as the diameter of the bore. The height of the tone hole was used to calculate the value of  $a_1$  for the reflectance filter specified in equation 16 above. This value was calculated using a script provided by Scavone in his thesis. The tone hole and bore diameters were used to calculate the value of  $r_0$ , which is the reflection coefficient for the bore itself and can be seen in the diagram in figure 4.10. This reflection coefficient is calculated using the equation below.

$$r_0 = \frac{r_{ToneHole}^2}{r_{ToneHole}^2 + r_{Bore}^2} \quad (18)$$

With the parameters for the tone hole specified, the last step was to select a position in the waveguide to implement each tone hole. This was done through trial and error because there was not a simple formula that produced the position in the digital waveguide from a desired frequency.

For the clarinet, there were 12 tone holes which were implemented. Each tone hole represented a different note and each one was given the same dimensions for simplicity sake. Through trial and error notes were made which spanned from C3 to B3. Because each tone hole had the same physical dimensions, this meant that they all also had the same filter coefficients for the reflection filter as well as the same value for the reflection coefficient,  $r_0$ . So, to implement the clarinet an excitation was applied and the script ran through each tone hole. Each tone hole had a position on the waveguide from which to take samples, of

the forward and reverse traveling wave, and then place values back in. The reflection filter coefficients that were used depended on the state of the tone hole, either opened or closed.

The length of the digital waveguide itself was chosen to be the length required to reach the lowest desired frequency, in this implementation, C3. The reason for this is that the lowest note corresponds to each tone hole being closed and the open end of the bore causing the reflection of the wave, which creates a frequency all in itself.

A diagram of the final clarinet model is shown below in figure 4.12. Not all tone holes are shown in this diagram, but as mentioned above, 12 tone holes were implemented for the clarinet.

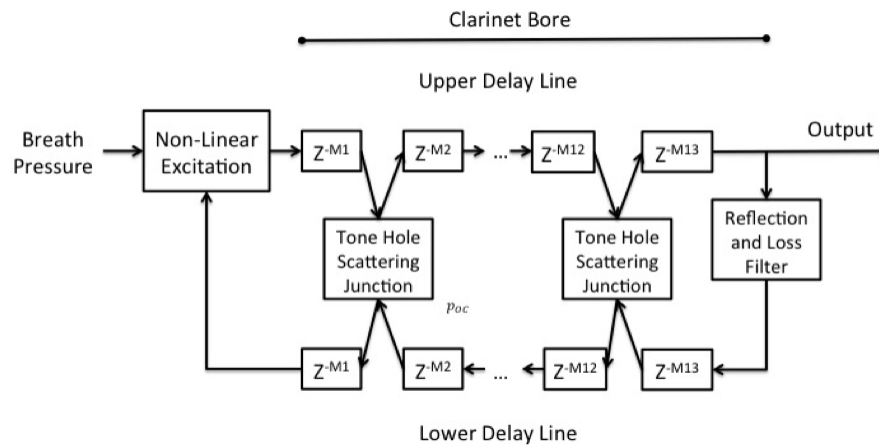


Figure 4.12: Final Clarinet model with tone holes

The total delay of the waveguide above is the sum of the delays M1 through M13, this corresponds to the delay of the lowest note. The delays between each tone hole represent the change in frequency from one tone hole to the next. Finally, the total delay from the excitation to any one of the tone holes represents the frequency of the tone hole.



In the model above, there are two elements that have not yet been explained. The first is the excitation mechanism and the second is the reflection filter. As was mentioned above, when all of the tone holes are closed, the frequency of the note is determined by the open end of the bore of the instrument. Therefore, the reflectance at the open end of the bore needs to be specified. Unlike the piano, the reflection in the clarinet model is frequency dependent and also varies based on the diameter of the bore itself. However, it turns out that the reflectance at the open end of the filter is modeled quite well by a low pass filter with a phase delay of 180 degrees. So the reflectance and low pass filtering is accomplished using the same filter as described in the digital waveguide section, simply multiplied by -1.

The non-linear excitation used in the clarinet model is also not overly complicated. A real clarinet is excited using a single-reed that acts as a pressure-controlled valve. Pressure is developed in the oral cavity and the difference between the pressure in the oral cavity and the pressure in the bore of the instrument causes air to flow from the oral cavity into the instrument. Conversely, when the pressure in the bore is higher than the pressure in the oral cavity air actually flows from the bore into the mouth. Therefore, the reed is modeled as a simple non-linear reflection coefficient that looks at the pressure differential between the bore and the oral cavity. The pseudo code for this calculation is displayed below.

1. Sample pressure of reverse traveling wave at mouthpiece ( $p_{mp}^-$ )
2. Calculate pressure differential between the oral cavity pressure and the reverse wave pressure.

$$\Delta p = \frac{p_{OC}}{2} - p_{mp}^-$$

3. Calculate the reflection coefficient ( $\tau_{mp}$ )

$$\tau_{mp} = 0.7 + 0.4\Delta p$$

4. Update forward traveling pressure wave at mouthpiece using reflection coefficient

$$p_{mp}^+ = \frac{P_{oc}}{2} - \Delta p \tau_{mp}$$

The excitation in this model uses a  $p_{oc}$  value of 1. This fairly simple algorithm accurately models the single reed mouthpiece and produces sounds that are very similar to that of a single reed instrument. The only drawback with this excitation mechanism is that sometimes the output sounds too clean, which makes the instrument sound slightly electronic.

With a proper non-linear excitation in place, the model is now complete. In order to choose a note the desired fingering is simply sent to the model in an array. So for example, if all of the tone holes are closed then an array of [1 1 1 1 1 1 1 1 1 1] is sent to the model, where 1 indicates the tone hole is closed and 0 indicates it is open. When the model sees that a tone hole is closed, it uses the proper filter coefficients for the tone hole reflection filter and each tone hole is looped through. This loop updates the pressure value in the position on the digital waveguide that the tone hole is connected to for each sampling interval. Once all of the tone hole calculations have been completed, the reflection filtering at the end of the waveguide is done. Finally, the excitation is calculated, the first position in the upper delay line is updated and the digital waveguides are shifted. The next iteration can begin after these steps have been completed.

The output sound of the instrument comes from the pressure in the bore of the instrument itself, so no body model is required. In a real clarinet the sound is produced at the end of the bore is connected to the embouchure. So, for this model the output sound was also taken from where the bore met the excitation. This corresponded to sampling the waveguides at the start of the upper delay line and the end of the lower delay. The sum of these two values is used to produce the output sound.

One thing that should be noted about this instrument is that the bore does not physically implement the bore of a real clarinet. The reason for this is that the exact measurements for a clarinet were not available. Also, the clarinet does not simply have tone holes, but rather a complex physical structure in which some buttons close multiple tone holes. So the physical structure of the clarinet would have been difficult to implement. With more time however, it would be possible to accurately model a clarinet.

#### 4.1.5 Flute

The flute was the second woodwind instrument that was modeled in this project. The bore of the flute had a structure that was identical to bore of the clarinet described in the section above. The main difference between the flute and the clarinet was the excitation. While the clarinet had a fairly basic non-linear excitation, the flute had a fairly complex excitation.

A real flute is excited using air that is blown over the embouchure of the instrument, which is shown below in figure 4.13.



Figure 4.13: Embouchure of the flute

Blowing over the embouchure of the instrument causes a pressure difference between the air above the embouchure and the air in the bore of the flute. This pressure difference causes a constant flow of air within the bore of the flute, however this does not create a standing wave. The bore of the instrument has a resonance that is dictated by which tone holes are opened and closed. The energy provided by the initial blowing of air over the embouchure brings the instrument into a resonating state. Once it reaches this resonating state, the player blowing air over the embouchure keeps it in a resonating state by constantly supplying energy. Therefore, the flute requires a constant excitation.

The model used to mimic this excitation comes from a paper written by Vesa Vlimki et al. [6]. In this paper the blowing of air from the mouth of the player is reproduced by random noise; the delay between when the air leaves the players mouth and when it reaches the embouchure is simulated by a separate delay line, called the jet delay; and the interaction of the jet with the embouchure is modeled using a sigmoid non-linearity, which is simply a hyperbolic tangent function. The block diagram for the final excitation model is shown in figure 4.14 below.

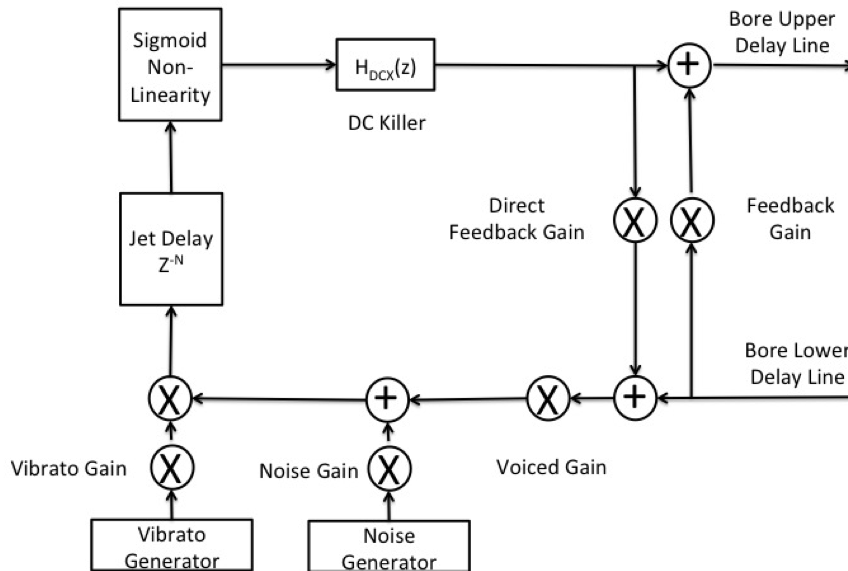


Figure 4.14: Final excitation model of the flute.

This figure only shows the excitation portion of the flute model, only the input to the upper delay line of the bore and the output of the lower delay line of the bore are visible. The diagram above shows a few extra elements in the excitation that were not mentioned above. The first is the direct feedback, this models the reflection of the wave already in the bore when it meets the embouchure. The direct feedback gain models the air from the air jet that is reflected from the embouchure back towards the players mouth. The voiced gain indicates how much of the air within the bore escapes and again travels towards the players mouth. The noise gain is used to increase the pressure of the blowing force. This gain has a threshold value, under which the model will not resonate.

The vibrato is used to modulate the envelope of the output noise, or air that has been blown over the embouchure. This allows the user to slightly vary the amplitude of the flute output, which causes a listener to hear low frequency, periodic variations in the amplitude of the signal. Once the noise has passed through the  $N$  sample jet delay, the output is fed into the Sigmoid non-linearity, which is shown below in equation 19.

$$sig_{out} = k_{sig} \tanh(JetDelay_{out}) \quad (19)$$

In this equation,  $K_{sig}$  represents the Sigmoid gain. The output of the Sigmoid function is fed into a high pass filter which has a pass band that starts around 10 Hz. The reason for this filter is to eliminate the DC component of the jet, which builds up over sampling intervals. Finally, the output of the DC killer filter is summed with the direct feedback and feed into the upper delay line of the flute bore model.

The tone holes in the bore of the flute are modeled using the same 3-port technique as used for the clarinet, the only difference is the position and the number of tone holes. For this project, a simple 6-hole flute was implemented capable of playing notes from G4 to F6, excluding sharps.

The interesting thing about this flute implementation is that with 6 holes, only 7 frequencies should be able to be played, G4 to F5. In fact, the position of the tone holes and length of the bore were chosen so that these would be the notes that were played for the different fingering patterns. However, there is a technique that flute players use called over blowing that increases the tone of the standing wave in the bore by an octave, or doubles the frequency. This increase in blowing velocity is mimicked using in the excitation model by decreasing the jet delay, this decreases the number of sampling intervals required for the blown air to reach the embouchure. Thus, increasing the blowing velocity. Surprisingly, this simple technique works quite well. The length of the jet delay can be reduced by about half, and the frequency will increase by 2.

As mentioned above, the bore of this instrument was the same as the bore of the clarinet; it just had different tone hole positions and numbers. The reflection and loss filter was also

kept the same. Tuning this model was similar to tuning the clarinet. The position of the holes were chosen so that when a certain tone hole was the first open tone hole from the excitation, the output matched the desired frequency. The flute however became more difficult to tune because there are a number of parameters in the excitation model that need to be altered in order to achieve the proper output sound. For the most part, once all of the different gains were set they did not need to be varied from note to note. However, the jet gain did need to be varied for each note. A general rule that was followed was that the jet gain should be about the same length as twice the delay from the excitation to the tone hole. For an overblown note, this jet delay was halved.

So for each desired note the fingering of the instrument had to be provided along with the jet delay. Once these values were found, the model could be played from G4 to F6. Again, one thing that should be noted is that a concert flute was not implemented in this project, but rather a simple 6-hole flute. Similar to a real clarinet, a real flute does not simply have tone holes but also a complex button structure that would have to be implemented.

Lastly, one difference from the clarinet is where the output sound was sampled from. For the flute, the output sound was taken from the end of the flute bore. This corresponded to sampling the end of the upper delay line and the start of the lower delay line.

Overall, the flute model produced very convincing sound. The main problem was that there was a background noise that sounded like air being blown through a straw. While this is characteristic in some blown instruments, generally a flute provides a very clean sound.

### 4.1.6 Guitar

The last instrument that was implemented in this synthesis core was a guitar. This guitar did use digital waveguide synthesis, however a more abstract version was used called Karplus-Strong synthesis. Karplus-Strong synthesis differs from DWS in two main ways. First, the upper and lower delays are grouped into one delay. Second, the loss and dispersion filters along with the body model are grouped into one filter called the loop filter. For this implementation, the loop filter was a simple single zero, single pole filter.

The block diagram for this system is shown in figure 4.15 below. It can be seen from the diagram that in general it is much simpler than the models that were described above. This is due to the fact that it was one of the earliest forms of DWS.

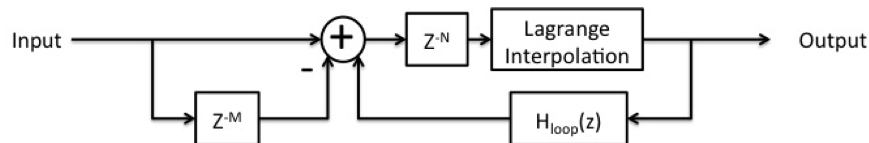


Figure 4.15: Karplus Strong model of Guitar

There are a couple of important features of this model. The first is the loop filter. As mentioned above, the loop filter itself is fairly simple. However, the tricky part is defining the coefficients for a filter that meets all of the requirements for loss, dispersion and body effects. To do this, a recording is made of the guitar and the `invfreqz()` function in Matlab is used. The `invfreqz()` takes the frequency response of a filter and provides the coefficients for the best fit filter. So, if the measured magnitude and phase bode plots for the instrument are fed into this Matlab function, the outputs are the values for the numerator and denominator of the loop filter.



For this project, there was not enough time to measure the frequency response of a guitar when an excitation was applied. So instead the filter values were taken from a project done by Steven Sanders and Ron Weiss at Columbia University [4]. Using the same technique described above, the loop filter that was found to model their acoustic guitar was:

$$H_{Loop}(z) = \frac{0.8995 + 0.1087z^{-1}}{1 + 0.0136z^{-1}} \quad (20)$$

The frequency response of this filter is shown below in figure 4.16.

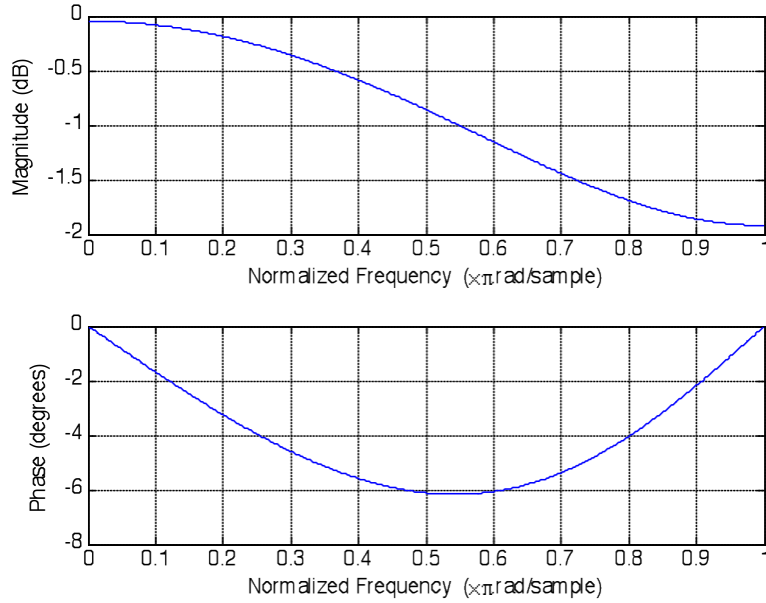


Figure 4.16: Frequency response of guitar model

From this filter it can be seen that the filter acts as a low pass filter and provides minimal dispersion, or phase delay. Again, the difference in magnitude between the pass band and cut of region is only around 2 dB. However, due to the fact that the signal passes through this filter multiple times, the higher frequencies are attenuated much more quickly.

With the loop filter designed, the next step was to define the excitation. For this project the excitation was chosen to be a plucked stringed. However, because the pluck could only be modeled from the output sound, the recorded note first had to be put through an inverse filter of the form.

$$H_{inv}(z) = 1 - H_{loop}(z)z^{-N} \quad (21)$$

Again, due to time constraints the excitation that was used came from the project at done by the students at Columbia. However, again, they used a very simple technique that easily could have been done with the appropriate amount of time. Once the excitation had been inverse filtered, the output was much shorter and had a length that was roughly half a second. A plot of the inverse filtered pluck is shown below in figure 4.17.

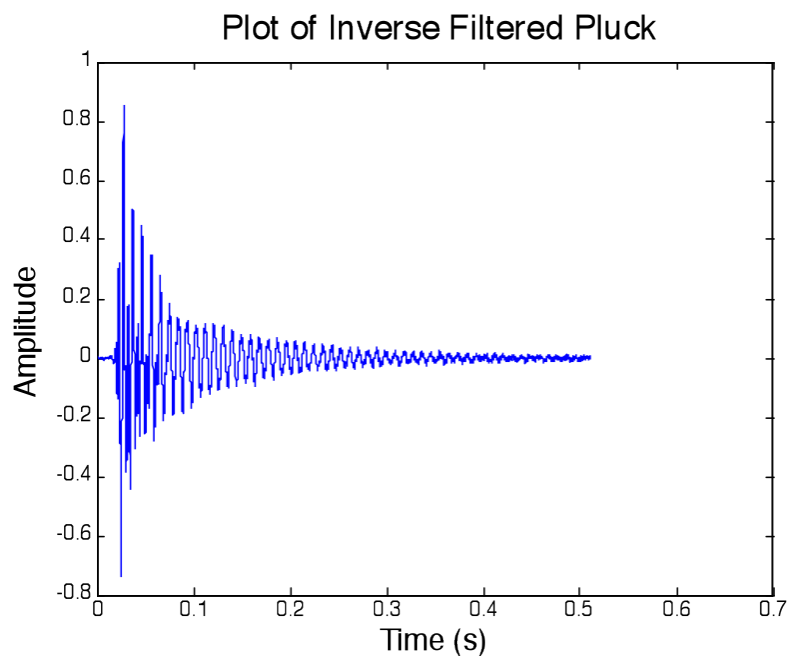


Figure 4.17: Filter Pluck

With the pluck defined, the second last step was to determine the length of the delay line for

each note. The delay can be defined using the formula shown below in equation 22. However, again the problem becomes that fractional delays are needed and only integer delays can be implemented using a digital waveguide. With the piano, this problem was overcome using an all pass IIR tuning filter. However, in this model the fractional delay was implemented using an FIR filter.

$$D = \frac{f_s}{f_{desired}} \quad (22)$$

A third order Lagrange interpolation was used to implement this fractional delay. The steps for implementing this delay are summarized below:

1. Determine the required delay,  $D$ , using equation XX
2. Take the modulus of  $D$  with respect to 1,  $d$  is the remainder
3. Subtract  $d$  from  $D$ , this will be  $X$
4. Set the delay length,  $N$ , to be equal to  $X$  plus 3 (the extra 3 delay elements are for the interpolation)
5. Normalize  $d$  between  $N/12$  and  $N+12$ , or between 1 and 2 for this third
6. Determine the interpolation coefficients for the  $N$  to  $N-3$  delay elements using the following formulas

$$c_{N-3} = \left(-\frac{1}{6}\right)(d-1)(d-2)(d-3)$$

$$c_{N-2} = \left(\frac{1}{2}\right)(d-2)(d-3)$$

$$c_{N-1} = \left(-\frac{1}{2}\right)(d-1)(d-3)$$

$$c_N = \left(\frac{1}{6}\right)(d-1)(d-2)$$

7. Now the fractional delay can be implemented as shown in the diagram below

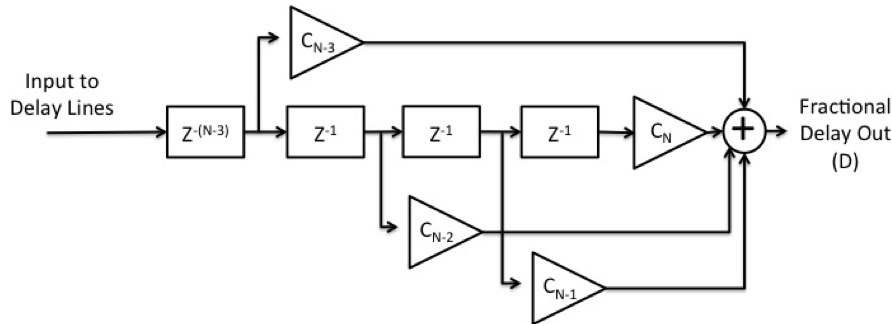


Figure 4.18: Fractional delay implementaion

The Lagrange interpolation worked extremely well, the output frequency was always within 1 Hz of the desired output frequency. Once the Lagrange interpolation had been implemented, the last step was to define where the input pluck occurred. This is a normalized value between 0 and 1, where 0 represents the Nut and 1 represents the bridge. For this project, this normalized delay was always set to 0.9. The coefficient that determined the pluck input was the delay length,  $M$ . To get this value, the normalized delay was multiplied by the delay of the digital waveguide,  $N$ . Once this value is found, the model is completely defined and can now be used to play any note.

For this project notes from E2 to E4, including sharps, were implemented. Tuning each note involved determining the lengths,  $M$  and  $N$ , of the delay lines as well as finding the Lagrange interpolation coefficients for the fractional delay.

In terms of complexity, this was the simplest model. However, the resulting output sound was extremely accurate and very similar to that of an acoustic guitar. The main problem with this model is that for high notes the model begins to lose accuracy in terms of convincing sound. This is due to the fact that the loop filter was designed using the low E string and so the dynamics of the low strings are captured, but not the high strings. To solve this,

a different loop filter could be found for each string and the loop filter used for a note would depend upon which string the note was played on. It is not an extremely difficult task and just involves an accurate measurement set up.

#### **4.1.7 Tuning and Initial Testing**

Although it was not be explicitly detailed in the sections above, all of the initial models were tested and tuned using Matlab. The scripts for each instrument can be found in the electronic appendix of this report. For the original tests of the models, Matlab functions were used to accomplish tasks such as filtering. However, a final implementation for each model was created in Matlab that did not make use of any Matlab specific functions and that used structures to change the note that was being played. This allowed the code to be easily ported to C, which was used for the real time synthesis.

For each model, there is a tune model, or Matlab script, along with a general, simplified script, which was ported to C. The tuning script was used to create structures for each note. These structures contain all of the information that makes each note, for each model unique. Once the structure is created for a note, then the simplified model can simply call upon the structure for a note instead of having to do a number of recalculations to get the information required to synthesize the note. This drastically reduces the computation time for each model. The structure for each note and for any of the models can also be found in the electronic appendix grouped with the tuning and simplified scripts.

In general, tuning each model involved two main steps. The first was to tune the model, or calibrate the gains in the model so that it remained stable and so that the output sound was similar to the physical instrument. This step was quite tedious as it required playing with a

lot of gains and also trying to figure out why certain models would go unstable. The second step was to tune the notes for each model. This involved changing variables of the model to get the proper output frequencies and sound dynamics. For example, for the piano, the frequency had to be right, but the hammer also had to be adjusted to get a proper hammer-string interaction with the string.

This tuning procedure was done for each model described in the sections above. For each note during the second step two main tests were completed. First, the frequency was checked using an FFT of the output sound. This was done to ensure the note was close to the desired pitch. The second test that was done in verifying each note was an auditory test. The sound function in Matlab was used to play the note. Sometimes, even if the pitch was correct, the note sounded off. So, the model would have to be adjusted and the steps mentioned above repeated. This was done for every note for every model.

## **4.2 Software**

The software written as part of this project has many functionalities and as such many programs were written to accommodate the variety of required tasks. This section begins with a high level diagram showing the relationship between these programs and subsequently a brief description of the various sections. A detailed discussion of the programs is given in later sections of this document.

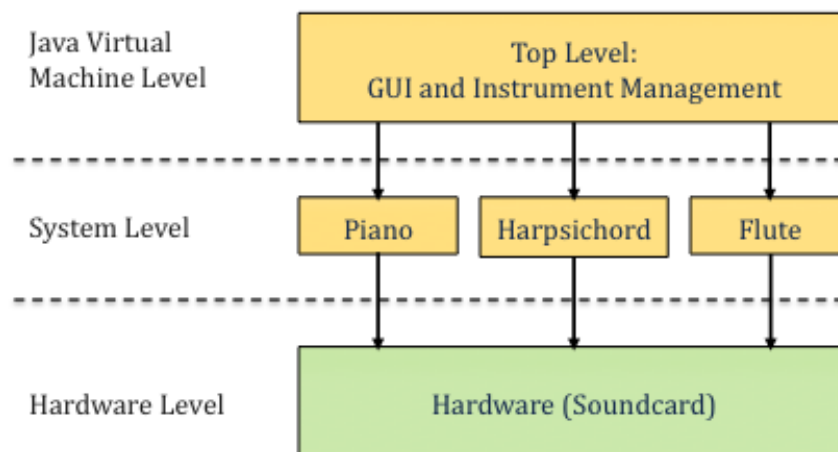


Figure 4.19: Software architecture

At the top level is the user interface. This is written in Java and allows the user to select which instrument they would like to synthesize, as well as the input method used to generate the notes. It also lets the user control the volume of the system.

The programs to synthesize the instruments are written in C, however after they are compiled they exist on the filesystem as binary executables. The top level java program will create a process within the operating system to execute the binary executable corresponding to the selected instrument. From here, the executable handles the configuration of the soundcard, opening a connection to the selected input port and synthesis of notes as dictated by the user. What this means is that each of the binary executables can, in theory, act as a standalone program to synthesize one particular voice and the top-level java program just serves to choose which program is running.

A detailed discussion of the various programs will be given herein.

### 4.2.1 MIDI as a System Wide Communication Scheme

Though it directly related to the synthesis of the software, this section begins by a discussion of how and why the MIDI protocol was used as a system wide communication scheme. This gives perspective to how the various processes described herein communicate. Through the research performed on communication protocols, an overall strategy for the SynthBox arose naturally. If the synthesis core was developed in such a way that it could respond to standard MIDI commands, and Bluetooth was enabled in a serial port configuration, the entire SynthBox system could have a standardized and efficient communication scheme that is supported by virtually all other digital music devices.

An additional benefit of this selection was that the SynthBox would now be MIDI compliant, rendering it controllable by any external MIDI device. Furthermore, if control of the SynthBox was based off of the MIDI protocol, then the SynthBox itself could be used to control other MIDI compliant devices, even with its associated Android application. Considering this benefit of full integration with a wide variety of other digital music devices, and the simplicity of MIDI from a hardware and software standpoint, made MIDI the best choice for a system wide strategy.

Once this decision was made, a secondary benefit of incorporating MIDI surfaced in that by designing a synthesis core that is MIDI compliant, several MIDI controllable parameters such as volume, pan, and reverb were then considered as requirements for effects and parameter.

### 4.2.2 Synthesis Core

In the scope of this project, the term *Synthesis Core* refers to the set of binary executables that synthesize the voices of each instrument. Five instruments were instantiated as part of



this project: piano, harpsichord, flute, clarinet and guitar.

## Algorithm

The complex algorithms that have been discussed in the Synthesis section that are used to synthesize the instrument voices on a sample by sample basis. Implementing these algorithms in C so that multiple notes can have their samples created concurrently, mixed and sent to the soundcard is a non-trivial task. This section discusses the way this challenge was addressed at a high level so that the details can be readily understood.

The algorithm to produce the various samples relies on two assumptions: the time taken to synthesize 1 period of samples for all active notes is

1. less than the time it takes to play those notes at the sampling frequency
2. less than the time that a user would notice a delay between when a key is pressed and when they hear the note

where once again a *period* is a discrete length set of samples that are transferred as a block to the soundcard using ALSA.

Each executable has two threads running. One thread watches the input source for incoming MIDI data that represents the note being played. The second thread, herein denoted as the main thread, performs all synthesis, mixing and soundcard output.

When the user inputs a note to be played, the thread watching the input source first parses the incoming MIDI data to determine the users intentions. The MIDI data does not always represent a note to be played as some MIDI devices will send a recurring keep alive pulse. MIDI commands are also used to change instruments and adjust the system's volume. If the

incoming data represents the user pressing, or releasing, a note, the thread will reformat the data and pass it to the main thread.

The main thread will do some initialization of the hardware and various data structures. It then begins a loop in which synthesis is performed. When the main thread sees some data has arrived, it must parse the data. If the new data represents a key press, the main thread initializes a structure called a *sample creator* and sets a flag denoting it as active. Conceptually, these sample creators can be thought of as object, but since C really does not have objects, they are truly just large data structures that contain all variables and data required to synthesize samples for a particular note. If the incoming data represents a key release, the thread locates the sample creator corresponding with that note and resets the active flag.

Following the potential parsing of the input data, the next step in the algorithm is to generate sampled values. For each active sample creator, the main thread will call a function used to create one period of samples. Following this, the samples are mixed. If the instrument uses a body model, the samples are run through the body model equations to perform the mixing. This is the case for the piano and the harpsichord. Otherwise, the samples are simply added. Finally, one period of mixed samples are sent to the soundcard using ALSA.

Figure XX, below, shows the flow chart of the algorithm that has been discussed. Time  $t$  is the time mentioned in the assumptions above. The following subsections will give detailed discussion of various parts of this algorithm.

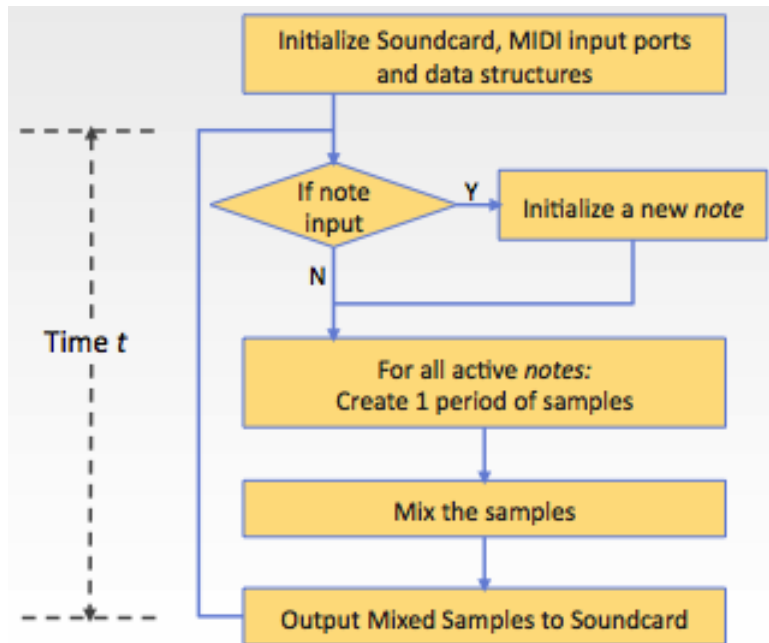


Figure 4.20: Synthesis core algorithm

## Initialization

When an executable for one of these instruments is started, the first thing to happen is that various data structures and system interfaces are created and initialized. This section discusses that process.

The first thing to be initialized is the ALSA interface to the soundcard. A data structure that acts as a handle for interfacing the soundcard is first created. It is then passed to two functions, one which sets the hardware parameters and one that sets the software parameters. Hardware parameters setup the buffer so that it is dual channel, signed 16 bit interleaved samples. The period size is set to the smallest possible size and so is the hardware buffer size. Both these mitigate the delay between when a key is pressed and when the corresponding note is heard, but they also cannot be too small as the overhead of looping between periods can take up so much time that the samples are not written to the soundcard fast enough. We

have set the period size to be 24 frames and the hardware buffer size to be approximately 2300 frames, where once again a frame is 2 samples of interleaved left and right channel sound data. In the source code, the functions to initialize the soundcard in this way are stored in the header file `synth.h`.

Following the initialization of the soundcard, the sample creator structures are configured. This process can be different for each instrument but generally it involves allocating memory for the period of samples each sample creator will make at a time, as well as setting up certain static parameters. This process also involves creating an array of data structures called *notes* that contain data parameters unique to each particular note that can be synthesized. Note structures can be complex or very simple depending on the instrument and the complexity of the corresponding model. Memory is then allocated for the periods that will be mixed. Following that a thread is initialized to watch a particular input port. The parameters passed to this thread are the location in memory that the main thread will watch for incoming data, as well as a flag to indicate whether the input method is USB MIDI, or MIDI over Bluetooth.

### **Input Monitoring Thread**

A separate thread from the main program is used to monitor the input. This allows for a non-blocking input in that the main thread is able to check if data has been received and, if not, continue with a normal course of execution. The thread first opens the MIDI port based on the flag sent from the main thread. It will open the port to receive MIDI commands over USB or Bluetooth. Once the port is open, the thread uses the following decision flow chart to decide how to process the data.

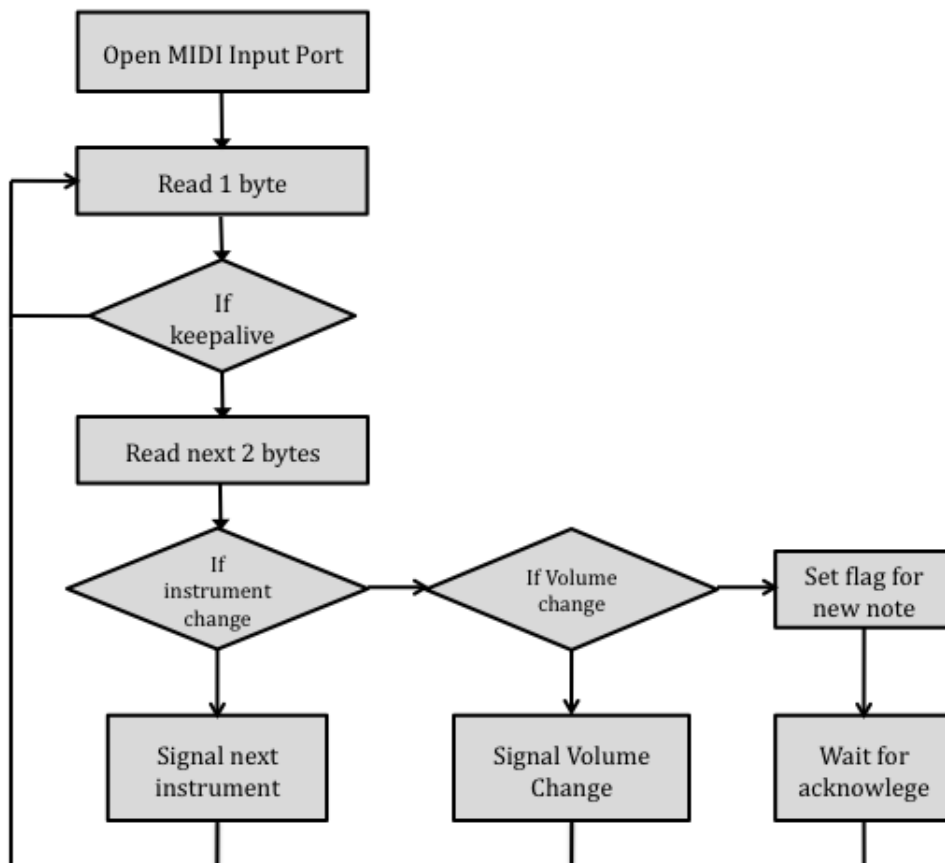


Figure 4.21: Decision tree of handling incoming MIDI commands

The thread will first read 1 byte. If this received byte is 0xFE, then it is a MIDI keepalive command called *active sense*. This byte is ignored by the application.

If the received byte is not a keepalive command, it is one of three command types. It can be a command to change the instrument that is currently being synthesized, it can be a command to change the master volume, or it can be a note input.

A midi command whose first byte is between 127 and 133 is a signal to change instrument, and the thread must write the data to a file called *proc\_msg* so that the top level GUI knows to change processes.

Similarly, data received with second byte equal to 7 means a volume change, and the new volume level, which is the third byte, is written to a file called *volume*. This file is read by the function that sends samples to the soundcard when the samples are converted from type double to type unsigned 16 bit integer.

Finally, if none of the cases discussed above apply, the received data must be a note that was input by the user. The thread reformats the data and places the data in the address passed to the thread by the main program. It must then wait for the main thread to reset the flag as an acknowledge that the data has been received.

## **Synthesis Loop**

One could easily imagine that some sort of looping algorithm might be used to iteratively generate synthesized samples. This section is dedicated to discussing how this happens. This provides only a high level view of this as this process because there are subtle changes in the details between different instruments. For very specific details of this entire process, readers are encouraged to view the source code.

As was mentioned above, the first thing that the synthesis loop does is checks if MIDI data has been received from the other thread. If the received data indicates that a key has been pressed, the thread must initialize a new sample creator structure corresponding to the note that has been pressed. If all available sample creators are currently busy synthesizing notes, the one that is oldest is stopped and replaced by the new note.

The way these new sample creators are initialized is different for each instrument but generally the process involves allocating memory for the notes delay lines, zeroing all arrays or setting them to their initial values, resetting the sample creators cycle count and setting the

active flag to true.

If the MIDI input data represents a key that has been released, the program searches for which sample creator corresponds to the released note and then sets the active flag to false.

The program then loops through all available sample creators. If the active flag is true, that sample creator is passed to a function used to compute one period of samples using the equations from the physical model of the instrument.

After the samples are computed, the samples are mixed. In the case of the piano or the harpsichord, the samples are sent through the equations that represent the instruments body model. Otherwise, the samples are simply added.

It should be noted that up until this point, the samples are all stored as double precision floating point. It is here that they are converted into integer and scaled by the volume level that the user has set. The samples are then arranged properly into memory using the format described above and are then passed to ALSA for sound output.

## **Top Level Interfacing**

Communication between the top level process/UI management java program and the synthesis core binary executables is handled using files. Specifically, there are two files called *proc\_msg* and *volume* that are used to pass messages between the two programs.

The volume file contains a single number between 0 and 100 that is read by the binary executable and used to scale the output volume of the system. This file is written by the top level program when a user adjust the volume on the GUI, and it can also be written by

the synthesis core program if a volume change MIDI command is received.

The `proc_msg` program is used exclusively to pass messages from the synthesis core program back to the top level. There are two instances where messages are passed.

#### **Instrument Change :**

If a MIDI command is received by the synthesis core and that command indicates a change of instrument, the synthesis core will write `i <#>` to the file where the `i` represents an instrument change and the number is the number of the instrument. The synthesis core then closes all its file descriptors and goes to sleep, waiting to be killed by the top level program. The top level program will then kill the process, flush the input of any remaining MIDI commands and start the process of synthesizing the next instrument.

#### **Volume Change :**

If the MIDI command received by the synthesis core is for a volume change it will write `v <#>` to the file where `v` indicates a change in volume and the number is the new volume level between 0 and 100. The top level program uses this value to correct the volume control slider of the user interface.

## **4.3 Top Level Program**

The top level program has two tasks. The first is to create a Graphical User Interface (GUI) from which the user can control the system. The second task it must accomplish is managing the underlying processes that are used to synthesize the various sounds. This section describes how these two tasks are accomplished.

### **User Interface Management**



The user can interface the software using the GUI shown below in figure XX. The GUI has a slider to change the volume of the system as well as drop-down menus for selecting the instrument and input method. There is also start and stop buttons, which are useful as the user can turn off the synthesizer and still use their MIDI input devices without having to disconnect them. The Logo is also displayed.

The `proc_msg` file used in the previous section is also used to manage the GUI. When the synthesis core receives a MIDI command indicating a change in one of the values on the GUI, that change is written to a file. The GUI will read the change from the file and adjust the values of the various fields accordingly. This means that congruency between the Android application and GUI is constantly maintained.

The user interface was created in Java, using the NetBeans IDE and making heavy use of the Java *Swing* packages.

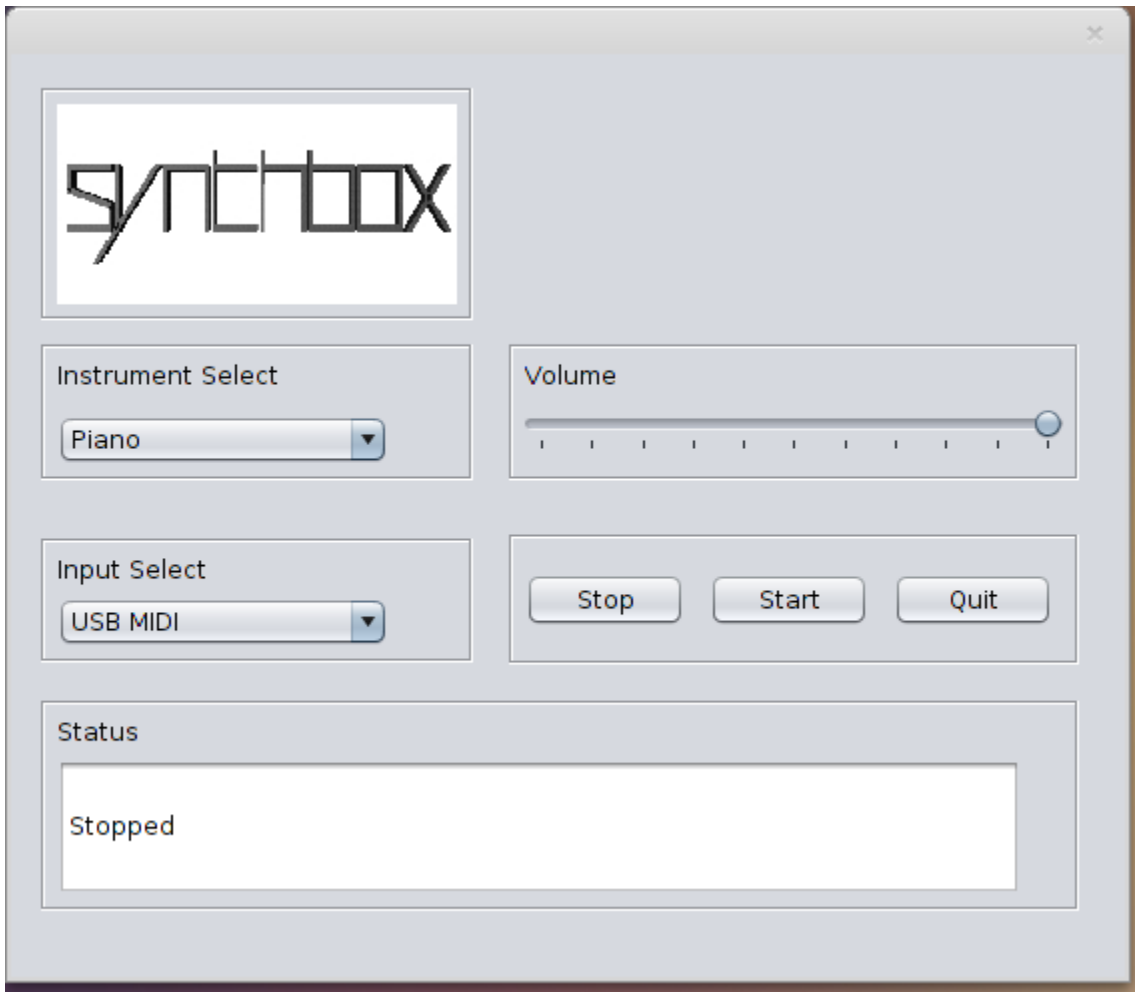


Figure 4.22: Graphical User Interface

## Process Management

Much of the GUI functionality lies behind the scenes in its management of the various processes required to synthesize the various instrument voices. From the inception of Java version 2, the standard library has included a package called *ProcessBuilder* which, as the name implies, is used to build processes and run them on the system alongside the virtual machine.

When the system starts, the first process that is created is the invocation of a shell to run a

script called *init.code.sh*. This script first compiles the c code used to create the instruments and places the resulting executables in the root folder of the synthesis software so that they can easily be invoked by the GUI when selected by the user. The script then writes default values to the configuration files such as *proc.msg* and *volume*. Finally, the script opens a serial port Bluetooth connection and listens on the channel for incoming connections. This allow the Android application to connect to the software.

As was mentioned earlier, each of the instruments can function as a standalone binary executable after it has been compiled from the C code. The GUI leverages this feature and so reduces its complexity by staying outside the synthesis and solely managing which instrument is currently being synthesized.

The GUI will switch processes in response to three inputs. The first is if the instrument select slider changes values, the GUI will stop the current process, if it is running, and replace it with anew process corresponding to the synthesis of the correct instrument. A similar thing will happen if the input select drop down is changed. It should be noted that these processes are switched *on the fly* in that there is no need to stop the synthesizer, change the drop down, and restart. On can simply change the drop down and let the GUI manage the start and stop behind the scenes.

The third time the process switches is when the synthesis core receives an instrument select change. This is signalled as a message in the *proc.msg* file which is checked periodically by a separate thread running within the GUI. When a new instrument select command is issued, the thread will first kill the current synthesis process and then quickly run a process to dump all data in the input midi ports as otherwise the first notes read by the new process would be garbage data. Following this, the thread will reinstate a new process with the new instrument and will adjust the value in the instrument select drop down. This is usually only

the case when the Android App signals an instrument change and so the drop down menu value must be adjusted to automatically maintain congruency between the app and the GUI.

While some effort and much success has been made to conveniently integrate the GUI and the Android app, there is still one unimplemented feature that users may desire. As was stated above, the GUI opens a single port at its inception to listen for incoming Bluetooth connections. If the Android App connects but then disconnects, the user will have to quit and restart the synthesis software to reconnect the application. This is not a huge inconvenience as the synthesis software usually takes less than a second to start, but it may catch users off guard if they did not know this.

It should also be noted that because the serial port for Bluetooth connection is created as a virtual file in the /dev directory. Because this folder generally requires root permissions to write, the process to connect the port and hence create the virtual file must be run as root. Therefore, the owner of the virtual file is root and so therefore, in order to read the file, processes must be owned by root as well. This means that the entire synthesis software must be run as root.

## **4.4 Android Application**

The customer requirements stated the desire for a wireless controller with an easy to use customer interface. Desires for a touch screen interface were also communicated. Thusly the decision was made to develop a tablet application by which the user would be able to control the synthesizer settings and play the synthesizer.

### **4.4.1 Application Environment**

The application was developed for Android 4.1 Jellybean operating system and designed to be used on a nexus 7 tablet. Android OS was selected over iOS because there is an easier developing environment and is more accessible to work with. The application was develop in eclipse Android software development kit. The layout of the application was created using XML and the app functionality was developed in java.

### **4.4.2 Application Functionality**

The basic purpose of the app is to serve as a midi controller which can be used to communicate and control the synthesis core. The application displays one full octave of keys which are used to play the selected instruments. To ensure maximum playing space the android application is locek in landscape view by altering the application Manifest. Buttons in the button left and right corners allow the user to select the current playing octave and this is displayed in the central display below the keys. A button to select which instrument is played can be found above the keyboard. As the app will need capability to communicate via Bluetooth, Bluetooth settings were established in the application manifest. Two buttons to select and communicate with a Bluetooth device are also found above the keyboard. Also above the keyboard is a switch which serves a sustain pedal when the piano is selected. A slide bar can be found above the buttons which can be used to control the master volume of the synthesis system. Finally, at the top of the application, is a display which shows the current note or notes being played. These features can all be seen in the screen capture of the application below.

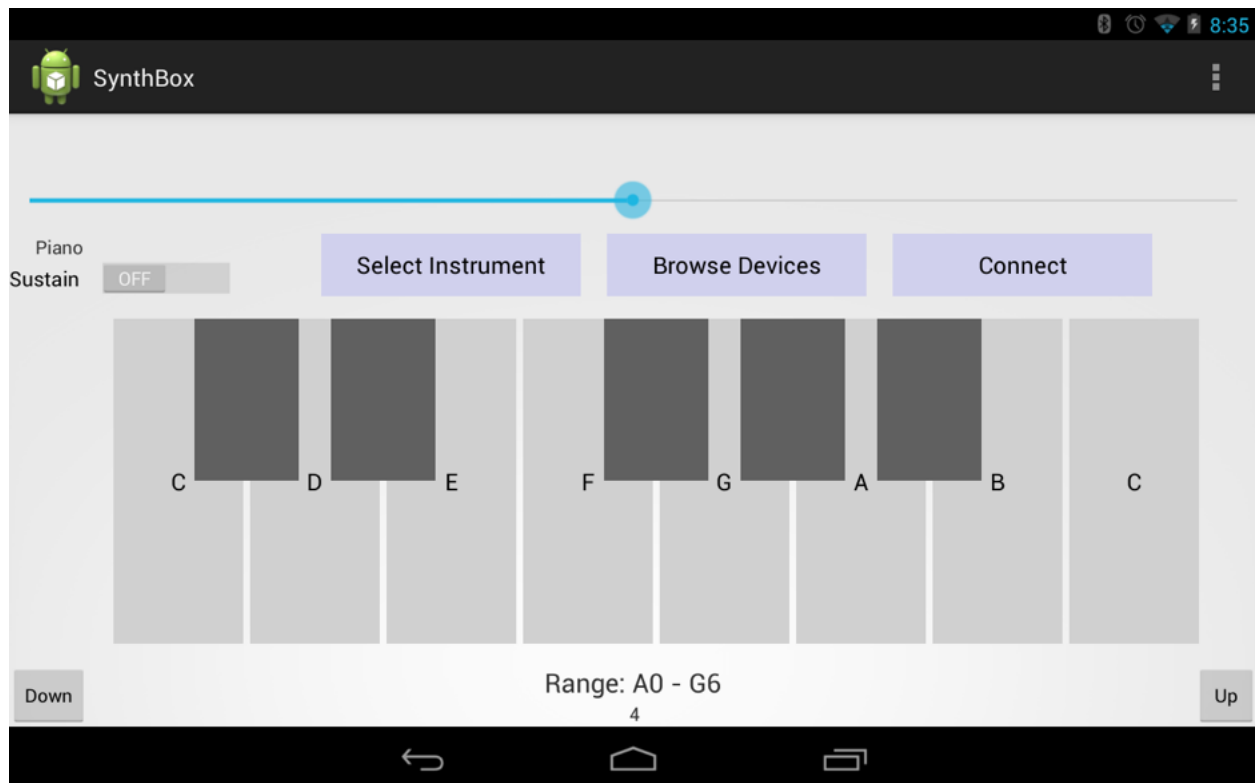


Figure 4.23: Android Application Interface

The control of the synthesis core is done using MIDI commands. This compliance with MIDI standards allows the tablet not only to control the synthesis core, but also to control any MIDI compliant device. A table of the MIDI commands used has been included in APPENDIX B

### 4.4.3 Keyboard

The keys of the keyboard are all buttons which implement an on touch listener. This means that when a button is pressed or released it triggers a function call to the corresponding button. When a given button is pressed there is a specific set of instructions which are performed. First the event action is checked to see the button was pressed or realised. If

pressed, the note value is calculated. The note is calculated using the following equation:

$$note = octave + 1.12 + position \quad (23)$$

The position value is dependent on the degree in the C major scale. For example C corresponds to 0 and F corresponds to 5.

The next step is to determine which channel or instrument is selected. This is done by checking the instrument state variable and adding 144 to comply with standard midi values. Finally the note velocity is calculated. The note velocity presented an interesting design challenge when working with a touch screen interface. The desirable effect would be that the strength of the touch corresponds to the note velocity. However to obtain this on the tablet the note velocity is a relationship to where the key is pressed. The touch position is captured using a function call. This cursor position is then normalized value between 0 and 127 which correspond to MIDI compliant note velocities.

The Channel selected, note value, and note velocity are then packaged as a byte array which can be sent via Bluetooth to the synthesizer core.

Additional aesthetic features also occur when a note is pressed. First the background color of the note is changed to show the user which note is currently being played. Also the note name and octave are displayed in a central display above the keyboard to show the user the name of the notes being played.

When the event action of a key is released a different set of commands occur. The first check is whether or not the sustain effect is enabled. If sustain is enabled the only changes upon release are the keys being reset to the default background color and the note display is

cleared. However if sustain is not enabled an OFF command must be sent to the synthesizer. Similarly to the key press the channel is polled from the selected instrument and added to 144 to agree with MIDI compliance. The note value also stays the same as when a key is pressed. The difference between key press and release is that the note velocity is set to zero upon release. These values are again packaged into a byte array and sent via Bluetooth to the synthesizer core.

#### **4.4.4 Instrument Select**

The instrument select button is used by the user to select which instrument sound will be used. This is done using an alert dialog. When the button is pressed an additional pop up window covers the keyboard with a list of instrument options. From here the user must simply touch the list containing the desired instrument and the alert dialog will update the current instrument and close the pop up window. When the instrument is changed two other events also occur. The first is a display update which shows the range of playable notes for the selected instrument. Secondly a MIDI command of channel plus 128 is sent to the synthesizer core. This tells the synthesizer that the application has requested an instrument change and that the playing file should be switched. Again this is packaged as a byte array and sent via Bluetooth.

#### **4.4.5 Volume Slider**

The volume slider is implemented using a seek bar and an on progress change listener. The seek bar has positions 0-10 as volume points. The user can simply slide the status icon using their finger and the master volume of the synthesizer will be updated. This update is triggered by a progress change. The update is implemented once again by packaging a MIDI



command to be sent over Bluetooth. To comply with MIDI standards the volume change array is composed of a channel select, the number 7, and then the state of the volume slider multiplied by 10. Additionally to sliding the volume by touch the volume slider can be moved using the hardware buttons on the Nexus 7. When a hardware button is pressed the keycodes are checked to distinguish between volume up or down and the corresponding change to the volume slider is made.

#### **4.4.6 Bluetooth Connection**

The Bluetooth connection is the key to the functionality of the tablet interface. Using the Bluetooth is a two-step process; selecting a device and connecting to the selected device.

The first step, selecting a device, is done by pressing the browse button. When the browse button is pressed a check is first done to ensure Bluetooth is enabled on the Nexus 7. If Bluetooth is not enabled an activity window will be launched asking the user to enable Bluetooth. Once Bluetooth is enabled the Bluetooth adapter of the Nexus 7 is checked and all paired devices are put into a set array containing the device name and address. Similarly to the instrument select button, an alert dialog is then launched containing the list of devices and their addresses. When a device is selected it is split apart from its address and the address is saved to be used in the Bluetooth connection process.

Once a device has been selected the connect button can be pressed to complete the connection to the synthesizer. This is done using the Bluetooth Socket. The Bluetooth device is first set to the adapter at the selected address. A Bluetooth socket is opened by creating a RF communication socket on the device adapter towards the defined UUID. The default UUID for Linux is 00001101-0000-1000-8000-00805F9B34FB. The key to a successful connec-

tion is having an open RF communication port of the synthesizer side of the connection. If this port is present the created Bluetooth socket will be able to connect with the synthesizer. Once a connection has been established the input and output streams from the socket are obtained and cleared.

#### **4.4.7 Sending Data**

Once a Bluetooth connection has been established data can then be sent back and forth from with the synthesizer. The data is packaged into byte arrays and fed into the socket output stream. These values can then be read by the synthesizer and interpreted appropriately.

#### **4.4.8 User Warnings and Messages**

To aid the user a messaging system was implemented to guide the user through the connection process. These messages are displayed using a Toast. A toast will appear at the bottom of the application window and will stay on the screen for a set period of time providing information the user may not be able to see. Messages such as, Please select a Bluetooth Device, Device Connected, and Device could not connect are examples of toasts which provide the user with information they would otherwise not know.

## **5 Specifications & Dependancies**

Modern Linux Operating systems contain millions of lines of code before any third party programs are even installed. This, coupled with the ever increasing complexity of PC hardware leads to the realization that, with so many volatile points of failure, it is amazing computers

will even boot. While care has been taken to ensure that the software released as part of this project can function within most modern Linux distributions, it should be noted that there are still some dependencies that must be accounted for. For clarity, these dependencies have been broken down into hardware and software.

## **5.1 Hardware Dependencies**

While the implementation of this project is primarily of software package, it should be noted that users will require a few specific pieces of hardware in order to actually use this software functionally. This section describes this hardware.

### **5.1.1 CPU**

Synthesizing instrument voices with a realistic sound requires complex models which in turn lead to complex calculations. Each sample that is computed can require tens to hundreds of floating point calculations. Multiply this by 44100 samples per second, times each note in a chord and add that to the overhead to manage each process and object and it becomes clear that the the system's processor must be highly performant. For example, a dual core 1Ghz ARM processor was only able to concurrently synthesize 3 notes of the most complex instrument model before audio quality was audibly deteriorated while a dual core 2.6 Ghz Intel Core2 (x86 Architecture) was able to concurrently 8. True performance benchmarking was not performed within the scope of the project, but it has been observed that in order to run this software a moderately capably processor is required.

### 5.1.2 Soundcard

Astute readers will not that it is desirable, when synthesizing instrument voices, to be able to audibly perceive the sounds that are created. For this reason, it is pertinent that users obtain a soundcard and, furthermore, ensure that the soundcard they have obtained is supported by ALSA. The ALSA development team does work hard to support a vast variety of sound cards and users can check on the ALSA website if their card is supported. 5

### 5.1.3 USB Port

Many modern computers ship with at least one USB port. The reason a USB port is required is that most modern MIDI capable devices will send MIDI data over a serial USB connection, or else a MIDI to USB Serial adaptor can be easily purchased. This is not a strict requirement however, as MIDI commands can also be received by Bluetooth.

### 5.1.4 Bluetooth

Again, many modern computers will come with Bluetooth support. This is required for users who wish to receive MIDI data over Bluetooth, possibly (and probably) from the Android Application that was created as part of this project.

In order to achieve Bluetooth connectivity in a way that would ensure portability across various hardware platforms during testing, a bluetooth dongle was procured by the design team in order to mitigate the issue of the software having dependency on a particular bluetooth hardware implementation.

Although this design decision could reduce the flexibility of the software, Bluetooth functionality is generally considered to be fairly uniform across systems and the software should be functional for any system with functioning Bluetooth hardware that meets the requirements discussed in the bluetooth requirements section.

The Bluetooth dongle chosen was the Linksys.

## **5.2 Software Dependancies**

Since this project is, from a deliverables perspective, a software project and interfacing the hardware was outside the scope of the project, certain software must already be in place in order to run the software that has been created as part of this project. This section describes these software dependancies.

### **5.2.1 Linux Operating System**

One might imagine that in order to run software that has been designed to run in Linux, that they would require to have a Linux operating system installed on their computer. This assumption is correct. Fortunately, Linux distributions are widely available.

Despite this wide availability, however, there are a few specific features that must be present in the distribution of choice. The first is that sound must work and it must use the ALSA modules. This indicates that very old kernels using the OSS sound system will not work with this software. Users should research the distributions to see if sound works natively, as finding the appropriate modules and setting up the configuration scripts to load those modules can be a non trivial task. This is also the case with Bluetooth.

### 5.2.2 ALSA Development Package

Many C programs use various libraries in the form of header files. The software used with this project uses a set of headers that control the ALSA device driver. In order to use these headers the ALSA development kit must be installed. Fortunately, most Linux distributions allow this to be installed as a package called *libasound2-dev*. Users can have the package installed automatically by running the following command at the Linux terminal as long as their Linux distribution comes with the package manager installed and their computer is connected to the internet.

```
sudo apt-get install libasound2-dev
```

### 5.2.3 Java Runtime Environment

The top level process and user interface management program is written in java and so the compiled java classes need the Java Virtual Machine to run. The virtual machine, along with the required java packages, can all be installed using the package manager as well by running the command

```
sudo apt-get install openjdk-7-jre
```

This will install Java version 7 which is the newest version. Java version 6 will work as well, but lower versions are not supported.

## 5.2.4 Bluetooth

In order to use the synthesis software as is, the host system must be capable of opening an RFCOMM port to communicate with the Nexus 7 software. This functionality is fairly standard in bluetooth hardware, and most on-board chips or external dongle should support these features. The code included in the software, which can also be used for testing, consists of two portions:

1. Registering an RFCOMM Serial port
2. Opening this port and waiting for a connection as the Master.

Any bluetooth setup capable of performing these two tasks should be capable of using the software. On the system tested, this was performed with the following two commands and is included in the initialization script.

```
//Register the RFCOMM service, channel number replaces # sudo sdptool add channel=#  
SP
```

```
//Listen for a connection on the Serial Port and feed data to RFCOMM sudo rfcomm listen  
rfcomm* #channel#
```

For example, registering a port on channel 15 and then listening on that port:

```
sudo sdptool add channel=15 SP
```

```
sudo rfcomm listen rfcomm0 15
```

No other particular specifications for the bluetooth hardware required besides the capability to create and communication over an RFCOMM port and pair correctly with the Nexus 7 tablet. As bluetooth range and speed is specified by its governing body, any on-the market bluetooth hardware would suffice.

## 6 Final Product Prototype

The block diagram for the final design of the product is shown in figure XX below. This block diagram shows how all of the functional blocks described in the final design section have been integrated to form the final product. The final design itself runs on any capable computer running Linux Mint distribution.

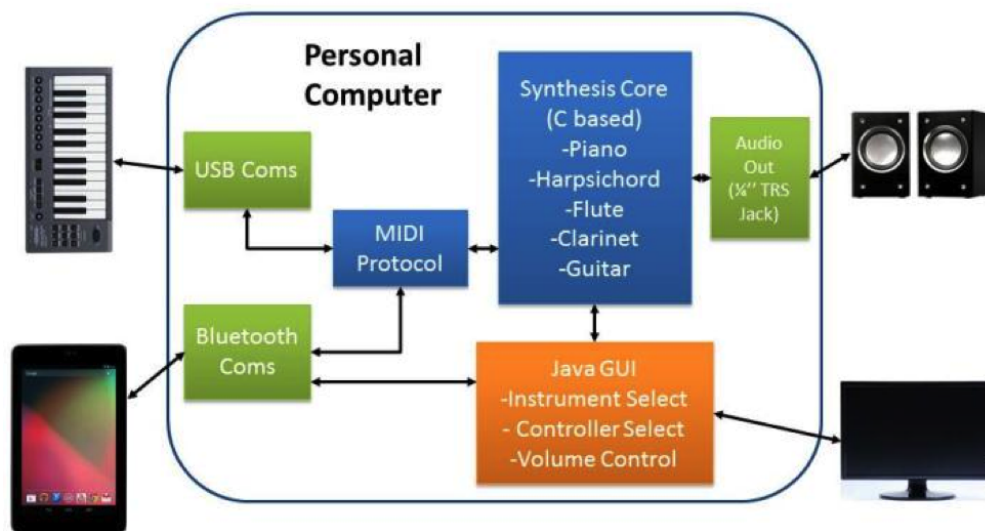
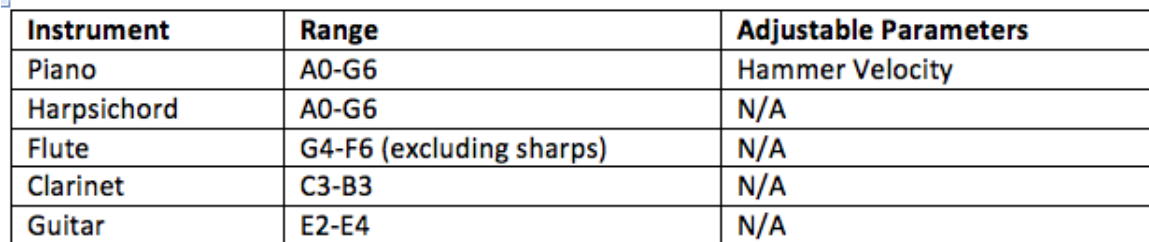


Figure 6.1: Android Application Interface

At the heart of the design is the synthesis core. The synthesis core contains the model for every instrument and can play every note in the specified range for each instrument. The syn-



thesis core also allows for 8 notes to be played at the same time, or 8-note polyphony. These note range and adjustable parameters for each instrument are summarized in the table below.



<b>Instrument</b>	<b>Range</b>	<b>Adjustable Parameters</b>
Piano	A0-G6	Hammer Velocity
Harpsichord	A0-G6	N/A
Flute	G4-F6 (excluding sharps)	N/A
Clarinet	C3-B3	N/A
Guitar	E2-E4	N/A

Figure 6.2: Android Application Interface

The synthesis core is controlled using the Java graphical user interface that is displayed onto the monitor of the computer. The GUI is shown in figure XX below. This GUI allows the user to start and stop the synthesis core, which controls whether notes can be played or not. From the GUI the user can also select which instrument is being played along with what volume the output is being played at. Finally the GUI allows the user to select what input is being used to control the synthesis core. The user has two options, Bluetooth and MIDI via USB.

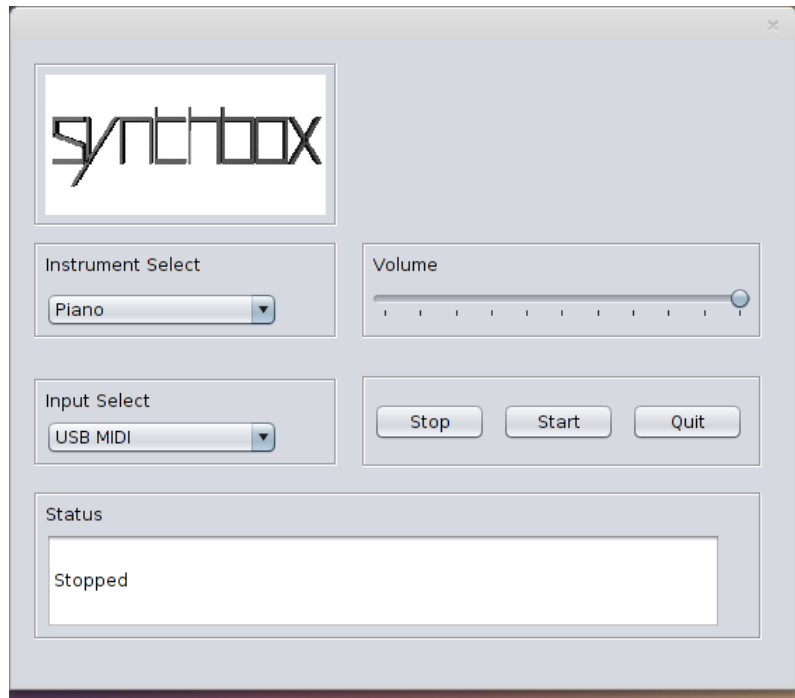


Figure 6.3: Graphical User Interface

If the user selects USB MIDI, then the synthesis core will be controlled using a physical MIDI device such as a keyboard or synth pad. If a keyboard is connected, then the notes on the keyboard will correspond to the notes that are being played. For the piano model, if the keyboard is capable of outputting the velocity of the key, then the attack of the hammer for a given note will depend upon how hard the user hits the key.

If the user selects Bluetooth, then the GUI will connect the synthesis core to the android application via a Bluetooth serial communication link. A screen shot of the android application is shown in the figure below.

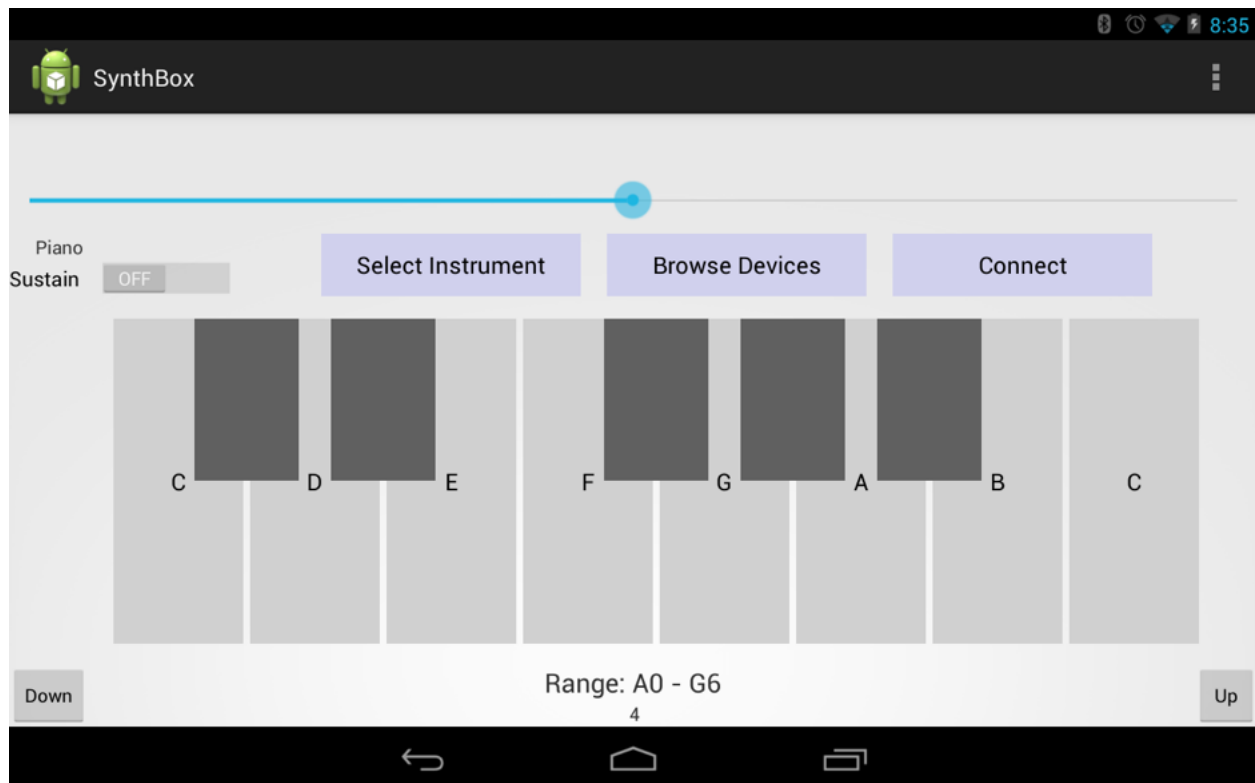


Figure 6.4: Android Application User Interface

In order for the application to connect to the synthesis core, the android user must choose the computer it would like to connect to via the Browse Devices button and then press the Connect button. Once the connection has been made, the player has the same control over the synthesis as they would if they were using the GUI. Using the application, the user can switch instruments using the Select Instrument button; and the volume can even be adjusted using the volume slider. For each instrument the playable range is displayed on the bottom of the application screen and when a note is played, it is displayed at the top of the application screen.

The application displays one octave of the keyboard and the Up and Down buttons can be used to control which octave is being played. When a key is pressed on the keyboard, the

application sends the note and the octave to the synthesis core to be played. Finally, for the piano model the hammer velocity is determined by where the user touches the key. For instance, if the key is touched at its bottom most point, this corresponds to a high hammer velocity. If the key is touched at its uppermost point, this corresponds to a very low hammer velocity.

Finally, once a note is selected via the Android application or USB MIDI device, the output sound is sent to the audio jack on the computer. This sound can be played out of built in speakers on the computer, or it can be sent to more powerful speakers for better sound quality.

So to summarize, our final design prototype is a real-time synthesizer with 5 voices (instruments), each capable of 8-note polyphony. The synthesizer is run on a personal computer running Linux via a Java GUI. The user can control the synthesis core using either a physical MIDI device connected via USB or by using the Android application that was developed for this synthesis core.

## 7 Conclusion and Future Work

The main goal of this product was to create an easy to use, real-time digital synthesizer that could accurately replicate the sounds made by real string and woodwind instruments. The use of digital waveguide synthesis to model the instruments allowed for synthesized instruments that accurately reproduced the sounds made by their real life counterparts. In total 5 instruments were created: a piano, a harpsichord, a clarinet, a flute and a guitar.

The use of C-code in Linux along with the ALSA sound architecture allowed for the synthesizer to be implemented in real-time with 8-note polyphony for every instrument, even the complex piano. A graphical user interface was developed in Linux which allowed the user to control the synthesizer along with what input was used to control the synthesis core, either a physical MIDI device or an Android application. Finally, an android application was created which allowed for wireless control of the instrument. This allowed the user to wirelessly select the instrument, play different notes, and adjust the volume of the synthesizer.

The final prototype that was developed met every one of the customer requirements and in some cases even exceeded them. So it can be concluded that this project was a success. However, if more time had been provided there are a few things that would have been changed in order to bring the product from a prototype stage to a software program that could be distributed.

The first improvement would be to the synthesis core. Currently the models produce fairly accurate sounding notes. However, if more time could be spent on the tuning of the individual models the output sound could be made even better. The second improvement that could be made would be to the Linux synthesis core. More time would be spent on the Linux program to ensure that the program remained stable, and redundant enough so that

it would never crash. Also, currently the application only works on the Mint distribution of Linux. Efforts would be made to make this code portable onto any distribution of Linux that supports Bluetooth and sound.

Although there were some improvements that could have been made, creating an accurate synthesizer that replicates real instruments is a non-trivial task. Further, adding the constraint of having to play the notes in real time makes this task even more complex. So in the 8-month timeline that was provided for this project our group is very proud of what we have accomplished. The synthesizer works well and rivals some inexpensive synthesizers that are on the market. If time had allowed for the improvements mentioned above, this synthesizer would have been competitive with most other software synthesis programs out there in terms of music quality and playability.

# A Testing and Verification

While many software projects opt to develop the entire software suite, or a subsection thereof, before analyzing the results through a series of predefined tests. However, due to the complexity of the overall software suite the design team realized that developing a complete series of tests could be error prone, as every unique point of failure in the system may not be addressable. For this reason, the instrument models, IO, Android application and software were tested individually throughout the product development cycle, rather than at the final integration phase. Often submodule test cases were not developed and instead the design team chose only to look at product testing from the perspective of the final product. This section, will commence with a discussion of some of the testing challenges and techniques used on the individual submodules and will conclude with a discussion of the tests used to verify the behaviour of the final system.

## A.1 Submodule Testing and Development

### A.1.1 Instrument Models and Individual notes

Creating a realistic sounding synthesis presents a unique challenge in that often the best way to judge the quality of the synthesized sound is through subjective auditory analysis. While fundamental frequency of the individual notes, and to a certain degree the harmonics, can be compared with recorded instruments during spectrographic analysis, the information regarding the timbre of the instrument cannot readily be deduced in this way. Listening to the sounds is the best way to determine the synthesis quality.

That said, the design teams finely trained ears were used to create realistic sounding instruments that are also in tune. To do this, iterative development of increasingly complex

instrument models were developed in MATLAB. Once a realistic model was developed for one note, the individual model parameter were adjusted so as to produce a series of notes having the correct intonation.

Some may describe this trial and error method as less than desirable, however it should be noted that developing a physical model of the instrument means that the effect of changing certain parameters becomes intuitive to the model developers. This led to an understanding of the effect of changing certain parameters of the models on the same level that a professional piano tuner may understand the effect of adjusting the parts within a real piano.

### **A.1.2 Synthesis Software**

As mentioned in the earlier sections the synthesis software was written in C, following the algorithms developed using MATLAB. Once the code to synthesize the sounds were ported to C the MATLAB debugger and gdb were used to verify the values of the individual variables through repeated iterations of the synthesis loop. Once the developers were satisfied that the sound producing algorithms had been successfully ported to C from MATLAB, auditory analysis of the sound was used for a second level of verification.

## **A.2 Overall System Testing**

The overall system was required to pass a series of tests in order to verify the correct operation. These tests, and the results that were eventually obtained, are described in the paragraphs below.



### **Individual Notes :**

It should be obvious that each note within the available instrument range should produce a sound that is analogous to that of the instrument being synthesized. It should have the correct volume level, timbre and intonation. Post development, each note was tested individually using input from both the USB MIDI controller and the Bluetooth connected Android Application to ensure proper sound synthesis.

### **Instrument Switching :**

There are two available methods for switching instruments. One is from the Android Application via Bluetooth and one is using the GUI. The mechanics behind the changing of instruments for each method are completely different and so individual testing needed to be performed for both methods. Each permutation of instrument changes was tested using both the GUI and the Android application was tested and found to be successful. Careful observation was also made during these tests to ensure that congruency between the GUI and the Android Application was maintained.

### **Multiple Concurrent Note Synthesis :**

The concurrent synthesis of multiple notes, commonly referred to as *playing chords*, is of crucial importance to users who wish to play complex songs using the system. Testing was performed, for each instrument, to ensure that chords having a number of notes from two to the maximum could be played simultaneously. Similarly, testing was also performed to ensure that playing greater than the maximum number of notes concurrently did not crash the system.

### **Input Select Switching :**

It is desirable for the user to be able to change input selection *on the fly*, which is

to say without having to stop and start the synthesizer. Input select changes were successfully tested while synthesis was being performed to ensure that this feature was available and functional.

### Volume Control :

As is the case with instrument selection, the volume of the system can be controlled using both the GUI and the Android Application. Volume adjustment was done using both input methods to ensure proper volume control. Also observation was made on both to ensure congruency between application and GUI.

## B Android Table

Midi Table			
Function	MIDI Values		
Play Note	144+Ch(0-4)	Note Value *	Note Velocity (0-128)
Stop Note	144+Ch(0-4)	Note Value *	0
Change Instrument	128+Ch(0-4)	0	0
Change Volume	144+Ch(0-4)	7	Volume (0-100)

Figure XX: Android Application Interface

## C User Manual

Note that the following instructions have been tested on Linux based systems running Ubuntu 10.04 and Mint 14, with root privileges. Other systems or non-superusers may not be able

to use the software as described.

## C.1 Starting the GUI and using the software

To start the Synthbox GUI and begin playing the synthesizer, please following the proceeding steps.

1. Connect a controller for the synthesizer, either a Physical MIDI keyboard, Nexus 7 Tablet, or a virtual MIDI keyboard (please see the Connecting a controller to the synthesizer section for instruction on how to complete this step).
2. Open a terminal window and navigate to the *GUI2* folder. If you have not yet compiled the Synthesizer software, do so now by typing the command `javac gui1.java` and press enter. Note that you may be promoted for the root password. Type the password and press Enter.
3. Run the JAVA program by entering the command `sudo java gui1`.
4. Select the appropriate controller from the dropdown box. The controller is highlighted in the figure below



5. Select the desired instrument from the dropdown box, highlighted in the figure below



6. Press Start

7. The synthesizer should now be playing. If it does not, see the troubleshooting section for possible solutions.

## **C.2 Connecting a controller**

The modular nature of the Synthbox software allows for any MIDI compliant device to control the Synthesizer software. The three possible types of connection are a physical MIDI controller connected to the system running the software via USB, running the Tablet app from a Nexus 7 Tablet, or connecting a virtual MIDI keyboard. This section will describe the process required to connect and use all three types of controllers.

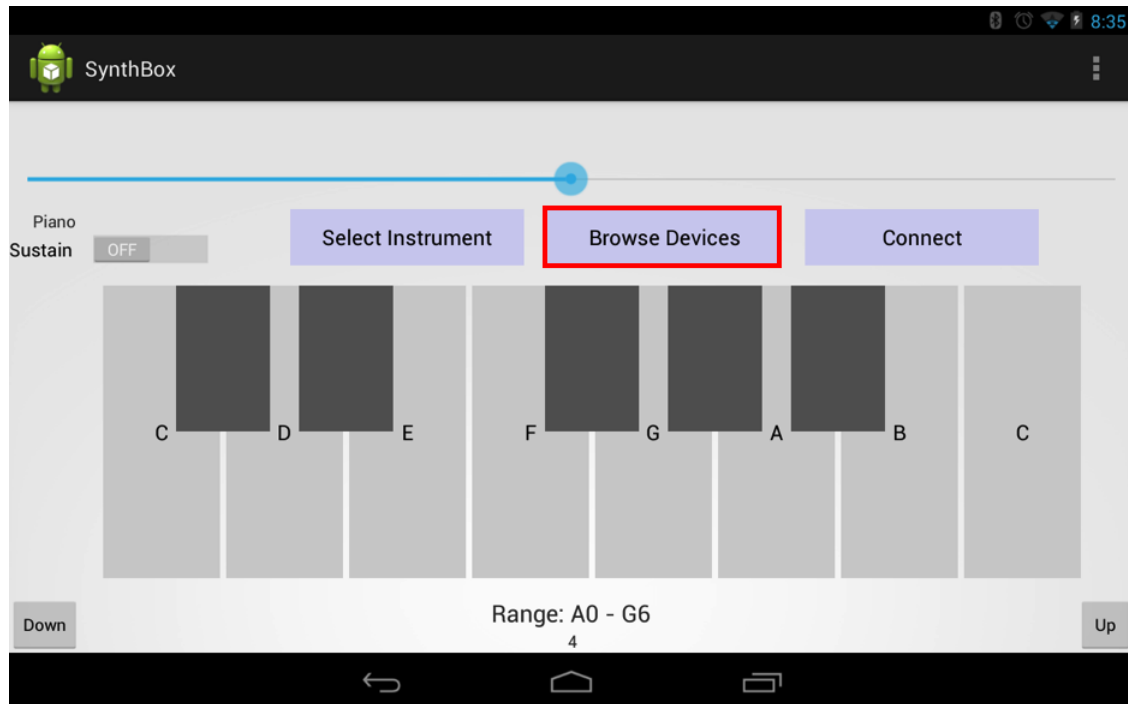
### **C.2.1 Connecting a MIDI controller**

1. Connect the MIDI controller to the system the Synthesizer is running on.
2. Ensure the device connected correctly, and is named `dmidi1`. A listing of functional devices can be found in the `/dev/` directory.
3. This should be all that is required to use a physical MIDI controller, provided you have access to the device and it is functioning correctly.

### **C.2.2 Connecting via Bluetooth**

1. Enable Bluetooth on both the Nexus 7 and host computer, and make both devices visible.
2. Pair the devices. This step is occasionally difficult with certain Linux setups, so refer to the troubleshooting section if necessary.

3. Start the GUI software as in Section 1 and select Bluetooth as a controller from the input select drop down menu.
4. On the Nexus 7 tablet app, press Browse Devices and find the host system's bluetooth device in the list of paired device.



5. After selecting the host system's bluetooth device, press Connect.
6. When the tablet indicates it has connected, you should press Start on the GUI.
7. The Nexus 7 software should now be connected to the Synthesizer and capable of playing notes.

### C.2.3 Connecting a Virtual MIDI device

In the event of a lack of a physical MIDI controller or Nexus 7 Tablet, a software based keyboard or MIDI controller could be connected to the Synthesizer software to provide effective

control.

1. Open a terminal window and create a virtual MIDI port by entering the command  
`sudo modprobe snd-virmidi`
2. Ensure the port has been created correctly and is named `dmidi1`
3. Open a software MIDI controller and connect it to this port.
4. Finally, open the GUI and start the software, selecting USB keyboard as the input.

## C.3 Troubleshooting

Although the Synthesizer software is robust, as with any software errors can occur and performance will vary depending on the system it is run on, especially due to the open source nature of Linux systems. The following section are some troubleshooting suggestions that may help in fixing any issues.

### C.3.1 Bluetooth troubleshooting

- Bluetooth has many known issues on Linux systems, especially newer versions of Ubuntu. (after 10.10). If bluetooth fails to function correctly, try to verify that pairing/a connection is possible with software or hardware that is known to be working on other systems. If pairing and connection is possible at all, pairing with the Nexus 7 tablet and synthesizer software should function as expected.
- Occasionally the Nexus 7 tablet will not connect to the host system, even when paired and bluetooth is turned on. Unpairing and repairing the devices is recommended if this happens.

- On Ubuntu 10.04, pairing must be initiated from the host system and not the Nexus 7 tablet in order for pairing to proceed successfully.
- Ensure that no other bluetooth devices are attempting to use the rfcomm0 port, as the initialization script used by the GUI software will only use rfcomm0; rename ports where required.

### **C.3.2 Nexus 7 Software troubleshooting**

The Nexus 7 tablet software should be fairly robust and work as expected provided bluetooth pairing is enabled. However, some errors have been known to occur sporadically, and this section attempts to provide troubleshooting tips in the event that these errors occur.

- Occasionally the software will freeze. Simply wait until the Android operating system force closes the application, and restart it.
- If Bluetooth pairing is not functioning as expected, disabling and re-enabling bluetooth usually solves the issue.
- In the event that neither of these issues are solved, restarting the Nexus 7 tablet completely should solve any additional issues.

### **C.3.3 GUI Troubleshooting**

The GUI software provided is designed to be portable and should be relatively platform independent, as it is JAVA based. However, it executes several C scripts and Shell files to perform the sound synthesis, and so errors can occur.

Some troubleshooting tips regarding the JAVA GUI are as follows:



- If the Java program will not compile or run, ensure you are using the correct commands, have the latest Java and JVM updates installed, and have root privileges.
- If the Java GUI does start but clicking Start does not initiate a sound, ensure the USB MIDI device or Bluetooth application is connected. These can be found in the `/dev` directory under the names `dmidi1` and `rfcomm0` respectively.
- If the synthesis engine stops when switching instruments, simply press start and stop a few times to restart the synthesis scripts.
- If the GUI is not started and the start button pressed with piano set as the first instrument, the GUI will crash on some systems.

## C.4 Using the Nexus 7 tablet to control other MIDI devices

A final feature of the Synthesis software developed is that with proper port routing, the Nexus 7 tablet can be used to control an external MIDI device. This is done by connecting the Nexus 7 Tablet to the host system as in Section 2, but running the Route software code. Then, the following steps must be followed to route the MIDI signals correctly.

1. Create a virtual MIDI port as in Section 2 using the command `sudo modprobe snd-virmidi`. Ensure the port is created as `midi1`.
2. Next, connect the external MIDI controller and ensure the port is registered correctly by viewing it in the `/dev/` folder. This port should be named `midi2`.
3. Run the commands `aconnect -i` and `aconnect -o` in a terminal to view all connected MIDI devices and their client number. Make note of the client number for the connected MIDI compliant device and of the Virtual MIDI port (defaulted to 0).

4. Perform a virtual mapping of the ports, mapping the output of the virtual MIDI port that the Nexus 7 will connect to (midi1) to the input of the external MIDI device (midi2) by entering the following command: `sudo aconnect #sender#:#port# #receiver#:#port#`

For example, if the Virtual MIDI port is client 20, port 0 and the external MIDI device is client 14, port 0, enter `sudo aconnect 20:0 14:0`

Graphical based MIDI routing programs could also accomplish the same function. The virtual MIDI mapping is required as the program will not function correctly if the input from the Nexus 7 is routed directly to the external MIDI device.

## D References

1. Smith, Julius O. Digital Waveguide Modeling of Musical Instruments, Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, 2012-12-10. Web published at <http://www-ccrma.stanford.edu/jos/waveguide/>
2. Rauhala ,J. et al., Tunable Dispersion Filter Design for Piano Synthesis, IEEE Signal Processing Letters, 13, 253 (2006).
3. Bank , B., Physics Based Sound Synthesis of the Piano, Masters Thesis, Helsinki University Of Technology Laboratory of Acoustics and Audio Signal Processing, 2000.
4. Sanders, S and Weiss, R, Synthesizing a Guitar Using Physical Modeling Techniques, Class Project, Columbia University, New York, NY, 2003.<http://www.ee.columbia.edu/ronw/dsp/>
5. Scavone, G. P., An Acoustic Analysis of Single-Reed Woodwind Instruments with an Emphasis on Design and Performance Issues and Digital Waveguide Modeling Techniques, PHD Dissertation, Stanford University, Stanford, California, 2000.

6. Hanninen, R, and Valimaki, V, An Improved Digital Waveguide Model of a Flute with Fractional Delay Filters, Nordic Acoustic Meeting, Helsinki, 1996.
7. Admin, Linux.org, 'What is Linux', Linux.org, 2012-07-20. Web published at <http://www.linux.org/article-is-linux>
8. Corbet, J., Allessandro, R. Kroah-Harman, G. 'Linux Device Drivers, 3rd Edition', O'Reilley Press, Sebastopol, California, 2005.
9. Unknown Author, 'How it works: Linux audio explained', tuxradar, 2010-04-10, Web Published at <http://tuxradar.com/content/how-it-works-linux-audio-explained>
10. Yaghmour, K. 'Building Embedded Linux Systems', O'Reilley Press, Sebastopol, California, 2003.