MusicKit Ultra

Designing A Software Synthesizer

Iain Emslie, Christopher Hettrick

In partial fulfillment of the academic requirements of CSC461

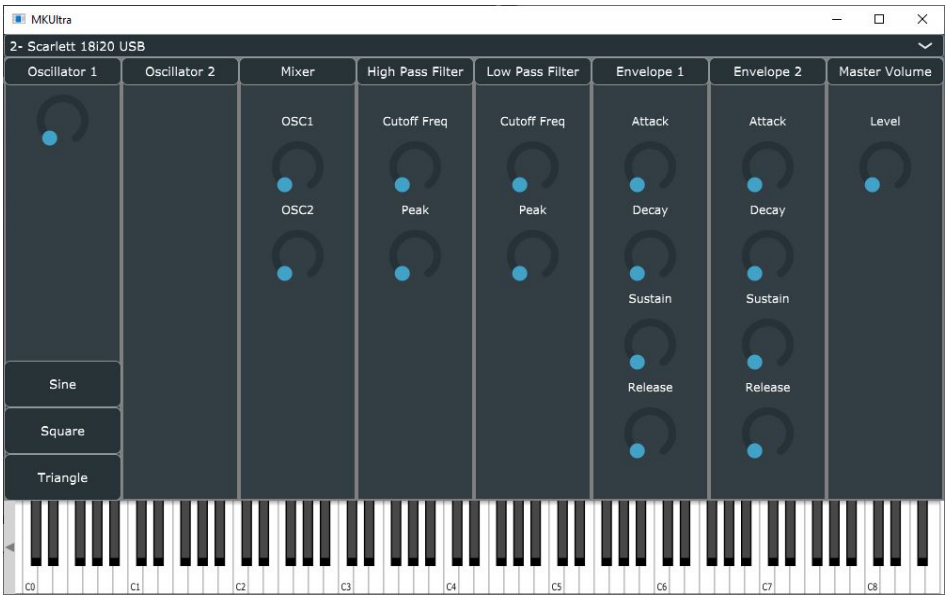
Department of Computer Science, University of Victoria

December 8th, 2019

1 Introduction

The intention of this project is to design and build a musical software synthesizer in order to better understand the ideas behind computer audio. This report will outline the design process as well as the concepts behind the components that make up our synthesizer. Our final version [1] used the Juce audio framework [2] which allowed us to more easily create a cross-platform program that had a graphical user interface.

Figure 1.1 - The User Window for the Synth



2 Design Process

The project began with the investigation of the best ways that we could create our own synthesizer. We knew that we wanted to create an interactive desktop-style application that would have a graphical user interface.

We decided to begin by building prototypes and building towards the final result through successive iterations. To begin with, we found a useful video series on building a simple

software synthesizer that allowed us to explore some of the core ideas of computer audio [3]. The first versions of our synthesizer were built by following these videos [4]. The author of the video series chose to use Microsoft Visual Studio which we continued to use throughout the project. In addition, we referred to the book *BasicSynth: Creating a Music Synthesizer in Software* by Daniel R Mitchell [5].

At first we wanted to build everything from scratch as much as possible. We explored a couple of ways of creating the GUI. One was using the Win32 API and another was using the WxWidgets library. Unfortunately, this proved to be beyond the scope of the project and did not provide simple cross-platform support. We experimented with different methods of interacting with the audio interface of the computer as well as the MIDI system [6]. On Windows it was relatively simple to write our own code that could receive and interpret MIDI messages but again this would not be compatible across different operating systems. We also tried using the PortAudio API [7] to interact with the sound hardware. Ultimately we decided that the Juce framework was the best option to get a working synthesizer that would satisfy our original criteria of graphical interaction and support across platforms.

We wanted to build something that would emulate hardware synthesizers which were designed to be easily used by musicians and not just engineers. This meant that the possibilities available to the end-user would be limited. Some of the different ways of synthesizing sounds are listed below:

1. Subtractive synthesis - Uses oscillators and filters to shape waveforms.
2. Additive synthesis - Sums together many oscillators to emulate the timbres created by the harmonic series of real instruments.
3. Frequency modulation - Creates complex waveforms by having oscillators which modulate the waveform of other oscillators.
4. Wavetable sequencing - Moves sequentially through pre-generated waveforms.
5. Sampling - Plays back audio recordings of real instruments or other sounds.

To limit the scope of the project we chose to follow the subtractive model since this was the method used in the older hardware instruments that we wanted to emulate.

3 Overview

The following section will outline the basic concepts behind audio synthesis that we learned while completing this project.

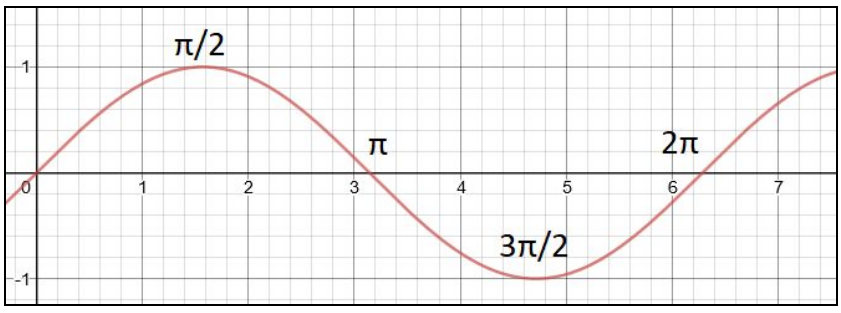
3.1 Oscillators

3.1.1 The Sine Wave

The sounds generated by traditional musical instruments such as pianos or violins are made up of oscillating sound waves which repeat at certain frequencies. The "Middle-C" on the piano, for example, has a fundamental frequency at 261.625565 Hertz. The different quality or timbre of such instruments is determined by the relative loudness of frequencies, above a fundamental pitch, known as partials or harmonics. These partials are multiples of the original fundamental frequency. This means that all pitched sounds can be represented as the sum of sine waves oscillating at multiples of a fundamental frequency [8].

The most basic waveform that can be generated is the sine wave. In software, we can create a tone generator by evaluating a sine wave function at intervals based on some sampling rate. Consider the graph of the basic sine function shown below:

Figure 3.1 - Graph of f(x) = sin(x)



The sine function gives us the y-coordinate of a point on the unit circle as it moves counterclockwise around the edge, beginning from the x-axis. When it is graphed, as in Figure 3.1, the values on the y-axis gives us the amplitude or loudness of a sound. The period determines the frequency or how often the waveform repeats within a given timespan. We can write the sine function as follows:

Figure 3.2 - Equation of the standard sine function

$$f(x) = Amplitude \cdot \sin(\frac{2\pi}{period}(x + phaseShift)) + verticalShift$$

The frequency of sounds that can be heard by humans is in the range of around 20 Hz up to 20,000 Hz. According to the Shannon-Nyquist sampling theorem, twice the highest frequency of a band limited audio source is the minimum necessary sampling rate to capture sound without a degradation of signal quality [9]. Following from this, the minimum sampling rate typically used in computer audio nowadays is 44,100 kHz.

We want to evaluate the amplitude of the sine function at each sample. First, we need to find the period based on the frequency of the sound that we want to generate. This can be found using the following formula:

Figure 3.3 - Period/Frequency Relation

$$timeOfPeriod = \frac{1}{frequency}$$

Now we want to find out how far we have moved along the x-axis of Figure 3.1 in the timespan of one sample. This gives us our phase increment, illustrated in the following formula, where f_s represents the sample rate and $\frac{1}{f}$ is the time of one period. 2π radians is the period of the sine function:

Figure 3.4 - Phase Increment

$$\varphi = \frac{2\pi}{f_s/f}$$

To determine the amplitude of the y-value at each sample we simply multiply our phase increment by our sample number and desired frequency, then evaluate the result using the sine function.

Besides the sine wave, there are other basic waveforms commonly used in audio synthesis such as the square wave, the triangle wave, and the sawtooth wave. These waveforms are often implemented as modifications of the sine function method outlined above. White noise is another useful waveform used by synthesizers, although it is randomly generated and is not periodic.

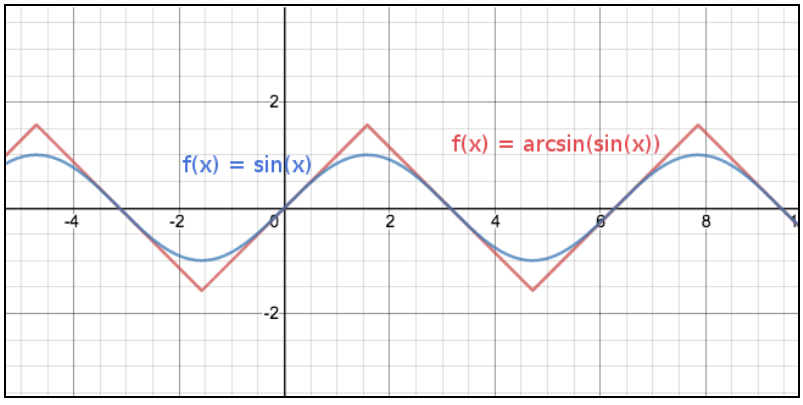
3.1.2 The Square Wave

The square wave was commonly used in early digital audio because it is a periodic waveform which instantaneously transitions from the highest amplitude value to the lowest. It was much easier to simply switch a voltage from high to low at a certain frequency than evaluating a sine wave function on early computers. Nowadays it conjures up the sound of retro video game arcades to the listener. For the square wave, we again use the sine function method but this time if the sound is above or below the middle amplitude threshold then the output is set to be either 1 or -1, respectively.

3.1.3 The Triangle Wave

The method we used to create our triangle wave was to evaluate the arcsine function with the result of the sine function outlined in the previous section. This results in the following graph:

Figure 3.5 - Triangle Waveform



3.1.4 White Noise

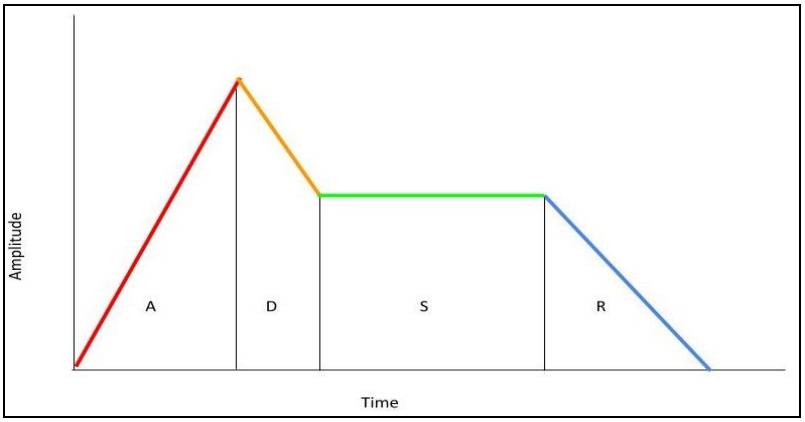
The familiar static we see on analogue television or hear on radios is the result of random signal noise. To emulate this effect in software we simply generate pseudo random floating point numbers within the range of 0 and 1. White noise can be used to create percussion sounds. For example, emulating the non-pitched sound of percussion cymbals. Filters and envelopes can be used to shape the sound to reach the desired result.

3.2 ADSR Envelope

Acoustic instruments such as pianos, guitars, and cellos all create sounds using vibrating strings. However, the method in which these strings are set in motion differs between each of them. For a piano, the player presses a key which activates a mechanical action resulting in a hammer striking the strings. The guitar player will pluck a string, while the cellist will typically use a bow. The envelope in a synthesizer is used to emulate the different ways that these sounds are created.

The key parameters of an envelope are attack time, decay time, sustain level and release time. The attack represents the time it takes for a sound to go from no amplitude to its highest amplitude. The plucking of a guitar string will have a much faster onset than that of the more gradual cello bow. The decay is the time it takes to go from the highest amplitude down to the sustain amplitude. For the piano, its loudest point will be the moment just as the hammer strikes the strings and then lessens to a sustain level. The sustain is the amplitude that is held the longest. A piano, with all its resonating strings, will sustain for a lot longer than a small guitar. Release is the time it takes for a sound to go from the sustain level to zero amplitude. Again, the guitar's sound will die off a lot faster than that of a piano or cello. Each of these stages can be represented in the following diagram:

Figure 3.5 - ADSR Envelope

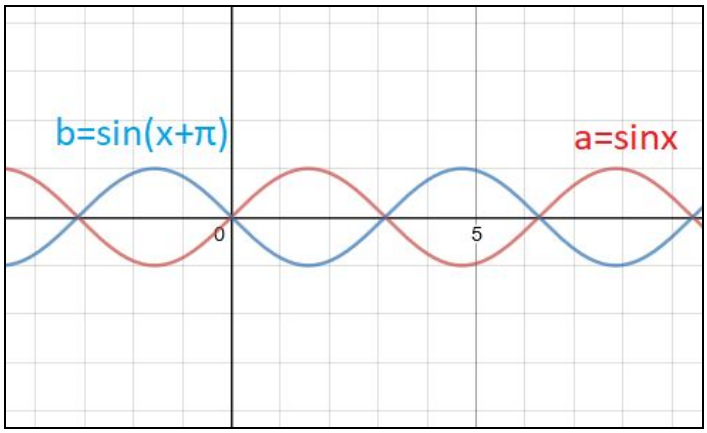


In software, an envelope can be created by modifying the amplitude (loudness) of samples based on the time since a key was pressed or released. A key press starts the envelope in the attack time. A key release transitions the envelope from the sustain level to the release time.

3.3 Filters

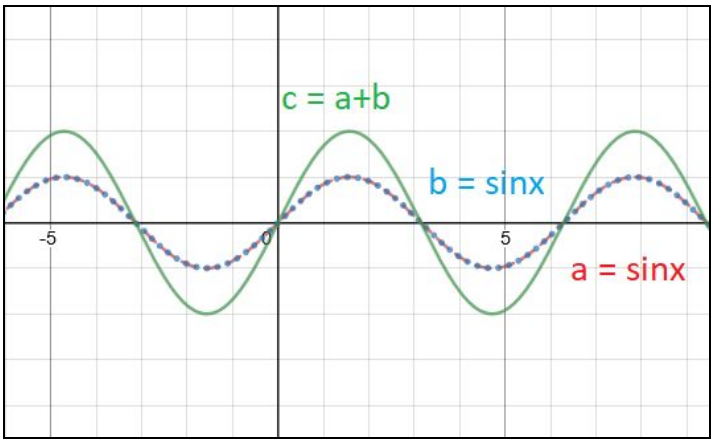
In additive synthesis, filters are used to change the frequency content of the sounds. Filters can be understood most simply as resulting from the addition of waveforms that are out of phase from one another. This idea is also used to create audio effects such as chorus, flanger and phaser. The following figure shows two waveforms which are phase shifted. When summed together they will cancel each other out resulting in no output.

Figure 3.6 - Phase Cancellation



If instead we have two waveforms, which are the same, then their sum will result in a doubling of the amplitude. This is shown in the figure below:

Figure 3.7 - Amplitude Doubling



To create a filter, the original signal is combined with a delayed (or phase shifted) signal thus attenuating or amplifying selected frequencies. In digital audio, samples can be stored in a buffer and then added back to the output signal at some delayed time. Different types of filtering effects can be created depending on how these delayed samples are combined with the original. A low-pass filter, which filters out frequencies higher than some cutoff frequency, can be created by adding a delayed sample to the current one. A high-pass filter, which filters out frequencies lower than some cutoff frequency, can be created by subtracting a delayed sample from the current one. If two delayed samples are added to the current sample then this creates a notch filter. Finally, if two delayed samples are subtracted from the current sample this creates a band-pass filter. The strength of the filtering effect can be attenuated by modifying the amplitude of the delayed sample.

3.4 Low Frequency Oscillator

Low frequency oscillators are used to modulate the waveform of other oscillators. They are typically below 20 Hz and are used to create a more interesting and natural sound. This can create a “vibrato” or wobble type effect. In our synthesizer this effect is created by evaluating a sine wave function and then multiplying the result with the sound generating oscillator.

3.5 Object Oriented Design

The design of the synthesizer went through a few iterations, each of which was a complete rewrite that encompassed the knowledge learned in the previous iterations. The final iteration is a polyphonic modular software synthesizer that responds to MIDI inputs and outputs audio data to the host operating system's native audio subsystem. The synthesizer utilizes the Audio, MIDI, and GUI subsystems of the Juce multi-platform application framework.

3.5.1 Audio Subsystem

The Juce audio subsystem requires the SynthesiserVoice class to be subclassed and to override a handful of its virtual methods, most critically the renderNextBlock() method. This callback method is called by the audio subsystem whenever it needs a new block of audio data to pass to the computer's sound card. For example, if the sine waveform is to be played by the synthesizer, then the sin() function is computed for each floating point sample in the outputBuffer. The delta increment between samples is calculated and added to the previous value to get the instantaneous value at the phase angle in the sine wave. renderNextBlock() requires nearly real-time response, so any method call that has unbounded response does not belong in this time-critical method.

3.5.2 MIDI Subsystem

The Juce MIDI subsystem is fully integrated into the Juce framework and works flawlessly in lockstep with the audio subsystem. A MidiMessageCollector object is responsible for receiving MIDI messages from the configured MIDI input source. These messages are received by the Collector via the removeNextBlockOfMessages() method, which then stores the messages in a MidiBuffer. This buffer of MIDI messages is processed synchronously with the accompanying audio buffer in the renderNextBlock() audio callback method.

3.5.3 GUI Subsystem

The Juce GUI subsystem fully utilizes the object oriented features of C++14. The fundamental unit of any GUI is the Component class. The special class AudioAppComponent is used as the

basis for audio applications, as it handles many of the unsavoury aspects of setting up the audio system. A parent Component object encapsulates itself and its child components. Each child can be a parent of other child components, and by further subclassing components in this way the whole design can be made very modular. This allows user interface design to be performed on a high level, as the low level details are handled through the class hierarchy. Subclassing continues down the hierarchy until fundamental components, such as rotary sliders, are utilized. The use of a rotary slider amounts to registering a callback lambda function that processes respective changes in value of the object variable that the slider represents. Either parent or child can be responsible for drawing each component through the use of the overridden paint() method.

3.6 Interaction with the Sound Card

A sound card is a particular type of digital to analog converter (DAC). It converts digital data into an analog voltage. All computers have a sound card of some type, whether it is an integrated part of the CPU or on the motherboard, a PCI SoundBlaster card, or an external high fidelity DAC.

Digital audio is represented in a sound card as a buffer filled with data that is either integer values or floating point values. An integer value normally is 32 bits in size and is signed. A sound card converts these values into a relatively appropriate analog voltage that drives connected speakers or headphones. More typically, sound cards handle floating point values between -1.0 and +1.0, which is what our synthesizer is configured to output to the sound card through the native audio subsystem.

Modern user audio software does not directly access the sound card. The host operating system has drivers for the sound card which conform to a generic application programming interface. This enables user audio software to be used on many different sound cards without alteration. Above the host operating system's audio API are cross-platform audio libraries, which we have used, that allow the user audio software to run on many different platforms and operating systems without alteration. Such libraries include JACK, PulseAudio, PortAudio, WASAPI,

Windows Core Audio API, Juce, and others. Our synthesizer uses the Juce audio library to achieve platform independence.

3.7 MIDI

Our synthesizer allows the user to control input through the computer keyboard, the onscreen keyboard via a mouse or touchscreen, or through an external MIDI controller. The following section will outline the basics of how MIDI messages and files are encoded.

The Musical Instrument Digital Interface protocol was developed in the 1980s as a digital successor to earlier analog methods of control and synchronization of hardware synthesizers. It was adopted as a standard by electronics manufacturers and remains the most widely used method for digitally controlling musical instruments. In computer audio, it is widely used to notate music and to control hardware and software synthesizers. MIDI files offer the advantage of being much smaller than actual sound recordings.

3.7.1 MIDI Transmission and Messages

MIDI data is transmitted as a big-endian unidirectional asynchronous bit stream at 31.25 Kbits per second. The data is encoded into multi-byte messages which begin with a single status byte followed by one or two data bytes. The first bit of a byte indicates whether it is a status or data byte. Status bytes denote the type of message that is being sent and what type of information is encoded in the following data byte(s).

MIDI messages can be classified as either channel messages or system messages. Channel messages are sent to one of 16 channels, which allows for the control of multiple separate instruments on the same MIDI connection. System messages are used to control global parameters such as the clock rate, the start of a sequence and the end of a sequence.

Digital instruments process MIDI data in real time. In order to synchronize timing between devices, the protocol uses clock messages. For every quarter note, based upon the global tempo, 24 clock messages are sent. Another method is to use tick messages which are sent at regular

timing intervals regardless of a song's tempo. When an instrument receives a "note on" message then a sound is played until a corresponding "note off" message is sent. Messages which control other parameters such as expression, volume, modulation, and sustain levels are also possible [10].

3.7.2 MIDI File Format

MIDI files contain data for one or more MIDI streams as well as the timing information for events. These files are broken up into "chunks," each of which consists of a 4-byte type, a 4-byte length and a variable length data section. There are two types of chunks: headers and tracks. A MIDI file contains a single header chunk followed by one or more track chunks.

Header chunks identify information such as the total data chunk length, the MIDI format, the number of tracks and timing information. The data section of track chunks contain a timing segment which indicates the number of 'ticks' since the previous event. This is followed by data which is either a MIDI message, Sysex or meta event. Sysex and meta can be used to encode data into the file which will not be interpreted as a MIDI message, for example track names or lyrics [11].

4 Conclusion

The intention of this project was to design and build a musical software synthesizer. We wanted to achieve a better understanding of the big ideas behind computer audio. A complete end-to-end design was realized in software that incorporates disparate knowledge in the areas of audio synthesis through the mathematics of periodic functions, digital representation of audio through protocols such as MIDI, graphical user interface design through the liberal application of object oriented inheritance and callback functions, as well as project planning, management, and efficient and effective teamwork.

The end result is the product of no less than three major redesigns and innumerous minor redesigns. The first major design was squarely using the native features of the Windows

operating system, such as the Win32 API. MIDI was manually and explicitly handled at a very low level. Communication with the audio hardware was through the olcNoiseMaker C++ library [12]. The graphical user interface was explored by using the WxWidgets library. The second major redesign was to replace the olcNoiseMaker audio library with the cross-platform PortAudio C library. This proved difficult in an attempt to implement real-time response and had no deep connection with the user interface. The final major redesign settled on the Juce framework for GUI, MIDI, and audio processing. This allowed us to focus on the core value proposition of this project, which is the understanding of audio generation and synthesis.

A project journal was maintained throughout the project development [13]. It outlined in some great detail both the insights gleaned through our studies as well as the struggles and challenges that we overcame through the journey of developing this software synthesizer.

Challenges we encountered amounted to learning the various libraries utilized, struggling with C++ build tools and environments, cross-platform idiosyncrasies, and being able to hold the whole design of the synthesizer in our heads at the same time.

Further work can be done on interface design, end-user usability features such as configuration menus, more filter and effect types such as flanger, phaser, and delay, and testing and deployment on multiple platforms, most importantly on mobile. These directions could be explored in directed studies, or personally for our own self-development benefit.

The project was a valuable contribution to our understanding of computer audio and software development. Iain began his degree with the intention of learning how to make programs like this so it was a great capstone project for his time at the University of Victoria. For Chris it was a worthwhile realization of his previous work in C++ programming and object oriented design, as well as his interest in music.

5 References

[1] https://github.com/iainemslie/CSC461Project

[2] https://juce.com/

[3] https://www.youtube.com/watch?v=tgamhuQnOkM&t=1130s

[4] https://github.com/iainemslie/CSC461Project/tree/master/Test1

[5] BasicSynth: Creating a Music Synthesizer in Software, Daniel R. Mitchell

[6] https://github.com/iainemslie/CSC461Project/blob/master/Test4/midi.cpp

[7] http://portaudio.com/

[8] BasicSynth, Chapter 5

[9] Fundamentals of Multimedia, Second Edition, Li, Drew, Liu

[10] The Complete MIDI 1.0 Detailed Specification

[11] https://www.csie.ntu.edu.tw/~r92092/ref/midi/

[12] https://github.com/OneLoneCoder/synth

[13] https://iainemslie.github.io/csc461project.html

6 Project Partner Contributions

6.1 Iain Emslie

- Project idea and proposal
- Audio synthesis technical research
- Design of software architecture
- Implementation and testing of multiple software versions to explore the viability of particular approaches
- Report outline and direction
- Significant contribution to report writing
- Project demo recording producer
- Project demo audio and slide content
- Project journal updates
- Project website implementation, hosting, and updates

6.2 Christopher Hettrick

- Design of software architecture
- Software repository maintenance
- Investigation and integration of various software libraries
- Graphic user interface design and implementation
- Testing cross-platform viability
- Report writing
- Project demo audio and slide content
- Project journal updates
- Project website updates