Spring 2023

Exploration of Digital Synthesis

Angelo Indre
adi19@uakron.edu

AN EXPLORATION OF DIGITAL SYNTHESIS

MARCH 3, 2023

ANGELO INDRE

Contents

Abstract

"An Exploration of Digital Synthesis" is a comprehensive investigation into the world of digital audio and music production. The paper explores the fundamental concepts of sound synthesis, including MIDI, virtual instruments (VSTs), and the JUCE framework. The central focus of the paper is the implementation of a custom synthesizer, which serves as a case study for the practical application of digital synthesis. The paper addresses the key question of how to create a functioning synthesizer from scratch, providing detailed insights into the programming and design process. Overall, the paper represents a significant contribution to the fields of digital audio and computer science, offering a valuable resource for both musicians and software developers alike.

History of Digital Music and Sound Recording

The earliest sound recordings date back to 1877. Air pressures generated by sound were funneled into a large brass conical fixture with a sensitive diaphragm that vibrates against a stylus. The stylus etches these sound waves into a wax cylinder which could be played back by rotating the cylinder and running a needle along the etching. These recordings were of low fidelity and music of the period was made to fit the recording capabilities. Loud, brassy instruments made the cleanest recording, so they were used more often than more delicate, softer instruments that did not record as well.

As time went on, recording changed and became more accurate, as sophisticated methods of recording were developed. The introduction of electric microphones and sound on film offered an increase in the range of sound frequencies that could be recorded. Next, the more sensitive magnetic tapes came about, and the fidelity increased even more.

In the present day, recordings are completely digital. In 1979, the first digitally recorded album *Bop Til You Drop* by Ry Cooder was released. From then forward, digital recording was the most popular means of making music or recording sound for any purpose. Compact discs came out around this time. They had the capability to record frequencies of sounds that were indistinguishable from the original sound source to the human ear.

Basics of Digital Audio

We know now that sound recording is primarily digital these days, but what does that mean? How do sound waves in the air get transformed into digital information that we can understand? I've explored and researched the pipeline of this process and outlined it in the next section.

A sound source causes periodic changes in air pressure. The continuous changes in air pressure over time are synonymous with sound. So, if we could record and reproduce the exact air pressure over time for a period, then we can replicate sounds. After being generated by a sound source, the first step for recording audio is the device that intercepts a raw sound wave: the microphone. The microphone generates a unique electrical signal based on the air pressure it intercepts. Different types of microphones accomplish this in different ways, but those hardware details are outside the scope of my research.

There is a key difference between digital audio and real sound waves: real sound waves are continuously changing, while digital audio must record a discrete set of data because continuous data is infinite and impossible to record. So how many discrete datapoints (called

samples[1]) does it take to accurately model the sound source? If we collected and reproduced a low number of samples per second, then our replica would be choppy and inauthentic, but with a higher frequency of samples, 44100Hz. To be specific, then we could create replicas that are indistinguishable from the original sound. 44100Hz. is the golden number where any recording with a sampling rate[2] below is noticeably different from the original. This golden number comes from the highest detectable frequency to human ears, being around 20kHz. To capture any sound, we must record at least 2 samples per wave period. Otherwise, the peaks and valleys of the sound wave will not be modeled appropriately.

44100 datapoints per second is a huge number. It raises the question of how much data we must store to authentically reproduce audio. How many bits does it take to record a sample? These bits are meant to represent the amplitude of a sound wave at that point in time. Sound intensity corresponds to this amplitude and is measured in decibels. Decibels are on a logarithmic scale such that an increase in the sound intensity of 6dB is perceived to be twice as loud as before said increase. 0dB is inaudible and sound waves higher than 85dB can harm human ears. So this 0-85 decibels is the dynamic range that we must cover with our samples in order to reproduce every safe intensity for human ears. Between 0dB and 85dB, sound intensity would double about 14 times. So, we must use at least 14 bits to cover our dynamic range. The final answer is at least 16-bit bit depth[3]. With 16 bits, we can store 2^16th unique

[1] Sample – A unit of audio data
[2] Sampling Rate – The rate at which samples of audio are recorded.
[3] Bit depth – The encoding length of a sound wave's intensity in digital audio.

wave amplitudes which is the minimum number required to reproduce sounds that our ears are incapable of distinguishing from original source.

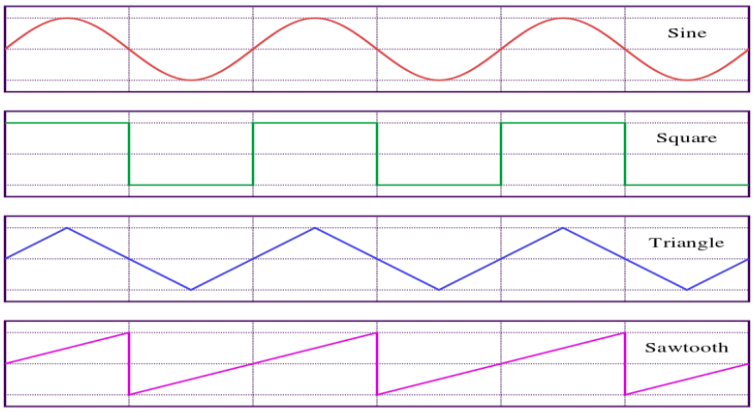
Say that for example, we are recording audio with a sampling rate of 44.1kHz, 16-bit depth, and 2 channels for stereo recording. That means that every second of recording requires 176.4kB and every minute of recording requires 10MB. This is the formula for the size of .wav audio files. Luckily, there exists the popular compressed .mp3 format which only costs 1/10th the storage .wav does.

Synthesizers

So now that we know what digital audio is, how do we make it from scratch? So far, we have learned what it takes to record a convincing copy of a sound source. Making the sound source ourselves would take less steps in theory. It is important to know that in the physics of sound, the frequency of a wave determines its pitch, and the amplitude determines its volume.

The most basic sound waveform is a sine wave. Knowing that frequency determines pitch and amplitude determines volume, you can predict and create the exact amplitude for a sine sound wave without playing back a recorded source. This is a very basic form of algorithmically generated sound. Now, if you assign specific frequencies to note names, and specific amplitudes to key press velocities, you can make a sine wave synthesizer operated by a midi controller.

Sound wave shapes and equations:



Below I've included pictures of two synthesizers I used as reference for building MySynth.

The left is called an Arturia MicroFreak, and the right is called a Moog Grandmother. One of the key distinctions between these two Keyboards is that the MicroFreak is digital while the Grandmother is analog. This means that Microfreak's internal states are determined by digital data while the latter has its states determined by physical electrical signals known as CV/gate's. This technology predates digital synths and assigns specific voltage levels to audio parameters. Since the invention of MIDI and other digital technological advancements, the CV/gate is less common and considered vintage nowadays. These digital synthesizers are much cheaper and easier to maintain than analog ones.

MIDI

How Does MIDI Work?

MIDI stands for Musical Instrument Digital Interface. It is, by design, a standard instrument language that can be understood across different manufacturers. MIDI was created in the 1980's when Ikutaro Kakehashi, the founder of Roland (one of the leading synthesizer manufacturers) reached out to other popular synth manufacturers and addressed the new issue that with an exponential increase in digital music hardware production, there needed to be a standard way for these digital technologies to communicate. The answer to this issue was MIDI standard introduced in 1982. It heavily influenced the way all digital music hardware was made for the next 4 decades. Every product concerned with digital music from then forward would lose a competitive edge if it was not built to support MIDI handling.

MIDI messages communicate information like the velocity with which a note is pressed, and the designated note number associated with which key was pressed, but what's the standard way this information is communicated so that all different controllers can be understood the same way by software?

A MIDI message is a tuple of 3 bytes (represented by integers in JUCE) and 1 double variable representing a time stamp on the message. I have a program called "The MIDI Control Center" which was provided to me when I purchased my Arturia Microfreak. This program allows you to manipulate a connected Arturia instrument from your computer. You can manage system-wide settings like lighting patterns and save banks or update the firmware of your device. One of the cool features of this program is called the MIDI Console. I can open the MIDI console and see that messages from my connected information are being logged to the console

at a rapid rate. MIDI is a form of digital signal processing. A program that uses MIDI is always awaiting messages and handles them in-place.

What do the three bytes represent?

Well, there must be more than just the three bytes and timestamp because these three bytes have different meaning depending on the TYPE of message of which there are 7.

- Note On
- Note Off
- Monophonic (Channel) Aftertouch
- Polyphonic (Key) Aftertouch
- Pitch Bend
- Program Change
- Control Change

Of this list, the most important to me in this project are Note On and Note Off, but I will also be concerned with Pitch Bend and Control Change for features I want to add down the line. But Note On and Note Off are key to making music (pun intended). For this type of message, byte 1 represents whether the message is note on or note off (seems like a waste of a byte), byte 2 is note number ranging from 0-127. 60 represents middle C on a normal piano and incrementing by halfsteps, the range covers a whole 88-key grand piano plus almost 2 octaves on both sides. The third byte represents the velocity with which the note was hit. This one also ranges from 0-127. Classically, lower velocities mean quieter sounds and higher velocities are louder ones.

The other 5 types of MIDI messages are all meant to communicate continuous change in a parameter. Note-On's and Note-Off's are one-time flags that change the state of the program, but these continuous change messages relay a smooth transition of a parameter over time. While keypresses can be thought of as buttons which have an event when pressed down and

lifted up, these continuous change parameters are communicating a parameter based on the position of a slider or mod wheel[4].

How Fast Does MIDI Need to Be?

The speed at which MIDI processing needs to occur to achieve a convincing acoustic emulation is quite rapid. In fact, the temporal delay between a keystroke and the corresponding sound emission must be imperceptible to both the performer and the audience. While there is no exact metric for this delay, it must be significantly lower than the threshold of audibility, typically around 20 milliseconds. This is crucial to create a truly immersive and convincing performance experience.

It is common for a program to listen to thousands of MIDI messages per second. Especially in cases of continuous change like pitch bend messages. The program that handles these messages must be sure not to drop any of them, especially Note-Off messages. Missing a Note-Off message can have the most noticeable impact on a performance. Missing a Note-Off message will on some patches cause a sound to continue indefinitely until the Note-Off message of the corresponding note number is processed. The presence of an unwanted sound is more noticeable than the absence of a sound one meant to play.

VSTs

Calling back to the difference between analog and digital synthesizers, certain parts of the hardware were digitized and the state of the instrument could be represented with digital data. Steinberg Media Technologies went further in 1996 by abstracting every physical

[4] Mod Wheel – An often spring-loaded wheel assigned to pitch bending. It moves on 1 axis and bounces back to a middle position when let go.

component of musical hardware into a digital version with VSTs. VSTs do everything that CV/gate logic can do with digital logic instead which is once again cheaper and easier to maintain than CV/gate alternatives. VSTs accept MIDI and audio signals as data, operate on them, and produce output in either MIDI or Audio signal form.

VSTis

A VST that has the concern of generating the sound instead of just operating it is classified as a VSTi (the "i" stands for instrument). This is the technical term for the program that I have been working on. Both VSTs and VSTis are commonly used in recording and performance settings. They require lots of processing power in order to meet the minimum requirements for low latency. It is recommended that a user has at least 8GB of ram and a processor clock speed of at least 2.0GHz but these specs vary depending on the complexity of the VST one wants to run.

The JUCE Framework

So, I've spent a lot of time so far reading about and working with the JUCE framework. It serves as a platform for audio plugin development that takes care of low-level details like porting to other operating systems. It also templates useful functions for creating your applications. The main goal of this project for me was to create a basic VSTi with MIDI and synthesis capabilities.

Building simple waveforms with a JUCE::Oscillator:

```
// return std::sin(x); // Sine Wave
// return x / juce::MathConstants<float>::pi; // Saw Wave
// return x < 0.0f ? -1.0f : 1.0f; // Square Wave
// return x; // Triangle Wave
// return static_cast<double>(rand(x)) / RAND_MAX; // Noise
```

Note that in JUCE, the “x” comes from a class called oscillator which will evaluate to a number between -1 and 1 and oscillate linearly between these values indefinitely.

JUCE Plugin Class Structure

A JUCE audio plugin has two classes of main concern. The Plugin Processor is responsible for sequential calls to a “processBlock()” function which is where the details of implementation of sound generation and manipulation go. The Plugin Editor is responsible for creating a GUI with buttons and sliders associated with parameters from the Processor.

When a user changes these sliders and buttons, they are updating a custom data structure from JUCE called a Value Tree State. This data structure is like a dictionary of statically typed nodes. Nodes can be accessed by their name and the type of data they store is determined at compile time. This structure also has predefined methods that make it easy to associate node values with the GUI.

JUCE Class Inheritance

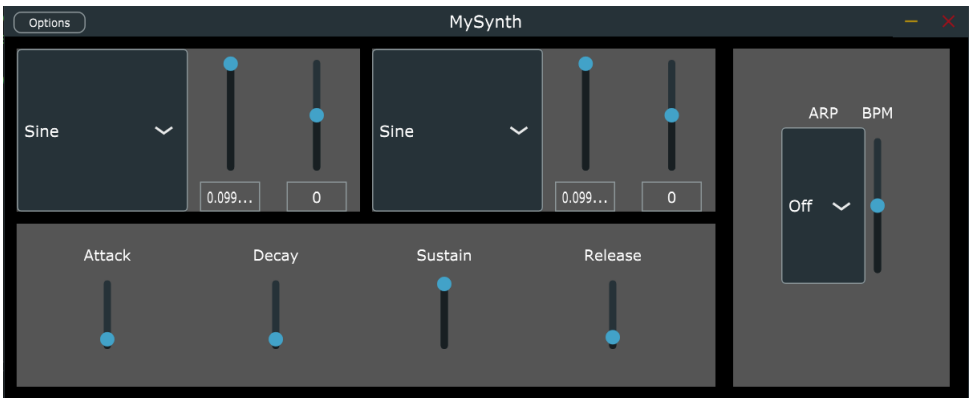
The JUCE framework offers lots of predefined template classes to developers so that they can redefine certain functions and the higher-level flow of control between classes is somewhat predetermined. Every class I defined for MySynth inherits from a template except for the ArpeggiatorData class which I wrote from scratch.

MySynth

MySynth is the custom synthesizer I built and the deliverable product of all this research and work. It features:

- 2 Oscillators with selectable waveshape, mixing, and tuning capabilities
- 5 voice polyphony[5]
- An ADSR Envelope[6]
- An Arpeggiator[7]

Here is a picture of the user interface for MySynth:

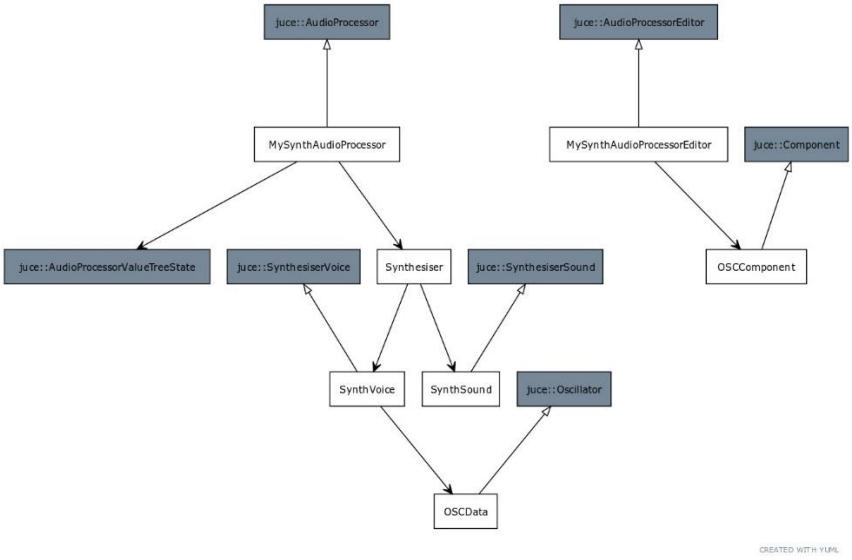


Recall the two synthesizers referenced above the Moog Grandmother and the Arturia MicroFreak. They each have a built-in keyboard and of course more parameters for users to manipulate. A synthesizer that does not have a built-in keyboard is known as a modular synthesizer. It is meant to be operated by an external controller of the user's choice. This is a strength since, for physical synthesizers, not having to support a built-in keyboard means you can have a more compact design and charge less for the product. But it is also a drawback since without an external controller, your synthesizer is rendered inoperable. Many consumer-grade

[5] Polyphony – Refers to the synthesizer's ability to play more than one pitch at a time rather than each Note-On message overwriting the last one.
[6] ADSR Envelope – Stands for Attack, Decay, Sustain, and Release. These 4 parameters change a sound wave's amplitude over time.
[7] Arpeggiator – Modifies held notes to be rearticulated in sequence periodically.

synthesizers give you the best of both worlds by offering a built-in keyboard for ease of use but also MIDI-routing capability to give users the ability to operate the “brains” of their synth with a different keyboard.

Below is a class Diagram of MySynth:



Some of the lower-level or repetitive features of MySynth such as the ADSR and Arpeggiator (which have similar roles as the oscillator) were omitted from the diagram to keep it readable. Each of the classes in white are ones I wrote myself. They all inherit from JUCE’s template classes (with the exception of my arpeggiator – not pictured). The diagram depicts two entities, one branches from the plugin processor and the other from the editor. They share data using the AudioProcessorValueTreeState which is also mentioned above. The editor holds a reference to the processor for easy access.

In terms of features, the two synths above have all the features of MySynth, as they are core features of any synth’s functionality. Both synths referenced have great separation of

concerns in their interface. The layout of knobs in the Moog Grandmother especially are visually appealing because of the borders enclosing knobs that belong to the same feature, and colors to offer more visual separation. The design of MySynth models this modularity by clearly separating the components with different colored padding. User interface design takes lots of time, though, and I would like to go back and spend more time tweaking and optimizing my layout.

The Audio Programmer Synthesizer

Awarding credit where credit is due, the most helpful resource for creating a synthesizer using the JUCE framework was the audio programmer community's "tapSynth" program, a documented tutorial for setting up component classes for a synthesizer. The class structure I employ, which separates concerns of editor components and processor data handlers, was found there.

The Arpeggiator Component

The last feature to be added to MySynth was the arpeggiator component. I'm most proud of my work on this component because I was able to apply the things I learned in building the other components to build this one independent of help from external resources. It manually watches the buffer of MIDI messages in my program and creates a second buffer of arpeggiated notes to match what the user plays.

Conclusion

Overall

To restate the goals of this paper, I wanted to research way how digital audio and MIDI worked and figure out a way to build a synthesizer for myself from scratch. I'm proud of my work this semester and my ability to communicate all that knowledge in this report. While I did

not meet all of my goals for the features of MySynth, I must acknowledge how far I got this semester, and of course, I plan to continue with development post-graduation.

Future Plans

For the purposes of the Honors Project, I have called "Code Complete" on MySynth, but as part of my goals were to make a plugin that I could use to perform with someday, I plan to resume development after graduation. Features I plan to add to MySynth in future development after I graduate include:

- More parameters to operate on sound waves, such as filters.
- Physical knob assignment of parameters to assignable knobs on MIDI controllers.
- Save banks for recording parameter presets.
- Redesigning the User Interface.

Resources

Information sources and articles

Sampling Rate and Bit Depth –

https://www.adobe.com/uk/creativecloud/video/discover/audio-sampling.html

The JUCE Framework – https://juce.com/

The Audio Programmer Synthesizer – https://github.com/TheAudioProgrammer/tapSynth

MySynth – https://github.com/aerdni99/MySynth

Image Sources

Wave shapes – https://quorumlanguage.com/media/dsp/waveforms.png

Arturia MicroFreak – https://media.sweetwater.com/api/i/q-82__f-webp__ha-e9c6f9ce9435409f__hmac-d5f776a1f9ca01af432f2d8dd3a6af7bdb80fd54/images/items/750/MicroFreak-large.jpg.auto.webp

Moog Grandmother – https://media.sweetwater.com/api/i/q-82__f-webp__ha-b3b80f8c9fac7983__hmac-1d18debd6bc9b8b8e4618a58baa48a685d6663db/images/items/750/GRANDMother-large.jpg.auto.webp