Tore Banta
Thomas Bertschinger

Sunday, March 12, 2016
CS 312

A MIDI-Controlled "Analog" Synthesizer

We implemented a monophonic digital synthesizer that emulates many of the features found on analog synthesizers from the 1960s and 70s, such as the famous Minimoog, developed in 1969. The most important part of an analog synthesizer is the voltage-controlled oscillator, which can produce a sine wave, and frequently also saw, triangle, square, and pulse waves. An amplitude envelope is also a crucial part of synthesized sound, many early synthesizers used an ADSR envelope (Attack, Decay, Sustain, Release) to control the amplitude of the sound over time. The attack is the amount of time taken to get from silence to maximum volume, once the sound begins. The decay is the amount of time taken to get from the maximum volume down to the sustain level, which is generally less than the maximum volume. Finally, the release is the amount of time taken to go from the sustain level to silence.

A fundamental part of any synthesizer is signal filtering. A filter is used to remove unwanted frequencies from a signal, and it is one of the best tools to turn electronically generated sound into musical sound. Analog synthesizers also frequently include an LFO, or low-frequency oscillator. The frequency of this oscillator is below the range of human hearing – less than 20 Hz, but rather than being used to create sound, it is used to modify sound, by introducing vibrato, tremolo, or modifying other parameters of the synthesizer.

Our synthesizer depends on two primary oscillators, which produce the base wave that is modified before being output as sound. All other components of the synthesizer are used to modify the sound produced by these oscillators. Each oscillator can produce a sine, square, saw/ramp, or triangle wave. The user can also control the amount of each oscillator

present in the output signal with the ratio slider. We use additive synthesis to combine the waves produced by the two oscillators – adding them together sample by sample to produce the output.

The first way we affect the sound is with the low-frequency oscillator, which by default is connected to the frequency of the output wave, to produce a vibrato effect. The LFO can produce several different waveforms: sine, saw, triangle, and square giving a variety of different flavors to the sound. The user can also connect the LFO to the cutoff frequency of the filter, the aforementioned ratio between the two oscillators, and to the amount of distortion applied to the output signal.

The sound is made more interesting by applying an amplitude envelope, with controls for the attack, decay, sustain, and release. We can also control the sound with a number of different signal filters: low pass, high pass, and resonant filters, using algorithms from the MUSC 208 curriculum. These filters calculate each output sample from the current input sample and the previous two input and output samples, based on an input slider controlling the cutoff frequency of the filter and a dial to control the resonance.

We also affect the sound with two different types of distortion, overdrive and bitcrushing. Overdrive works by multiplying each sample of the audio signal by a fixed amount greater than 1, and then bringing down all samples whose absolute value exceeds 1, complicating the spectrum of the sound and increasing its volume. Bitcrushing distortion works by rounding all samples of the audio signal to only a few specified values between 1 and -1, producing a retro effect by limiting the output signal to a specified bit depth. If there is a bit depth of n, we will allow only 2^n possible values between 1 and -1 that samples can take.

To control the output of our synthesizer, we receive signals from a MIDI keyboard. Our program only interacts with three types of signals: note on (0x8n), note off (0x9n), and pitch bend (0xEn). When we receive a note on signal, we initiate the attack portion of our envelope, with the oscillators producing a sound whose frequency is determined by the MIDI note number. When we receive a note off signal, we initiate the release portion of the amplitude envelope, which brings the amplitude of the signal down to 0 over a period of time. We use pitch bend messages to modulate the pitch by an amount determined by the user with a slider.

Both the pitch bend control and the LFO, when attached to frequency, use cents. This is a logarithmic scale that is useful for audio applications. There are 100 cents in a semitone. To increase a frequency f by n semitones, we compute

$$f\left(C^{100n}\right),$$

where C is the value of one cent. This is advantageous over simply adding a fixed frequency amount, in that no matter how high or low the frequency, the variance is always perceived as a fixed interval.

The order of the operations we apply to our signal is important: first, the oscillators produce the output wave, which is then passed to distortion. Next, the wave goes through the amplitude envelope, followed by the filter. We opted to use distortion before applying the amplitude envelope because distortion has a strong effect on the amplitude of the signal. If we applied it after applying the envelope, we might eliminate the effect of the envelope. Similarly, because distortion greatly increases the number of harmonics in the signal, it is important to apply the filter after applying distortion.

The first challenge we encountered when building this application was finding a way to write code that produced sound smoothly and efficiently while simultaneously monitoring input from the user interface and the MIDI controller. This required parallel computing, creating separate threads within the program which read and stored user input, produced sound utilizing the input, and output the audio using a buffer of 512 samples at a time. We created one thread which is tied to a function which reads in MIDI signals using RtMidi and stores them in a shared queue, a second thread which handles output sound buffers using RtAudio, and our main program thread which takes user input from the graphical interface and stores it in global variables which are used by the audio callback function to produce the output buffers.

In order to improve the efficiency of our callback function, which is called nearly one hundred times per second, we chose to implement the different wave forms for our oscillator (sine, saw, square and triangle) by creating wave tables. These store one period of each wave using 1024 data points. These tables are built using mathematical formulas on program startup, and the audio callback function reads from the table in order to calculate a given sample being pushed into the output buffer. We utilized these wavetables for both of our primary oscillators as well as for the low-frequency oscillator.

Throughout the development of this program, we also struggled with ways to create smooth and natural sounding audio by reducing clicks in our output sound, as well as refining the behavior of our audio callback function to account for edge case usage such as multiple keys being held down at once, or transitioning between notes partway through an ADSR envelope. This required tracking down inconsistencies in our code to ensure the

waves produced by the output samples were made up of smooth continuous values, and remained within the limitations of our digital audio convertor.

In addition to these difficulties, we ran into a number of issues implementing each of the different output signal modifiers in our audio pipeline. Whether it was simple errors in calculation when developing our signal filters, or difficulty determining the most efficient way to reduce the bit depth of our output signal without casting our floating point samples to integer values we could bit shift, this project presented us with numerous unique challenges. Perhaps most frustrating was the deeply hidden bugs which caused our program to stop producing sound during normal, functioning operation, which we never truly managed to pinpoint the source of.

In the end, we managed to implement all of the functionality we specified in our initial proposal, in addition to some minor additions we thought of while working on the project. Among the different features that we implemented for this project, our personal favorites are the low-frequency oscillator and filters, which allowed for the potential to create a vast array of different sounds. Being able to control different components of the synthesizer automatically by using the LFO is incredibly powerful, and demonstrates the impressive range of sound we can produce with the limited set of tools we created.

In reflection, we wish we had enough time to debug the program to the point that it runs consistently without crashing, and has the potential to be easily deployed to other systems as a complete application package. We had also hoped to add some additional stretch features, such as other types of synthesis than additive, like amplitude and frequency modulation, or the ability to store and recall input presets to save particularly compelling sounds the user might discover. We had even envisioned the ability to record a

series of MIDI inputs and play them back on a continuous loop, allowing the user to effectively control the synthesizer within the GUI without the need to simultaneously use the MIDI controller.

Overall, we are very proud of the final version of this program, and the extent to which we were able to implement features we had originally hoped for. We also developed this application from scratch, using exclusively our own source code, with the exception of MIDI and audio frameworks (RtMidi, RtAudio) which we relied on for the I/O portion of our application, and the filter algorithms which we used from the MUSC 208 course website. We had to research about audio synthesis in order to be able to implement some of our advanced features without relying on example code or other assistance.

References

History of Audio Synthesizers
http://musictechstudent.co.uk/music-tech-history/history-of-the-synthesizer-timeline/

MUSC 208 Course Curriculum – Filter Algorithms
http://people.carleton.edu/~jellinge/m208w14/index.html

Information on Cents – Used to accurately modify frequency
http://www.phy.mtu.edu/~suits/cents.html