Developing a Versatile Audio Synthesizer
TJHSST Senior Research Project
Computer Systems Lab 2009-2010

Victor Shepardson

April 7, 2010

Abstract

A software audio synthesizer is being implemented in C++, capable of taking musical and nonmusical input information and using additive and FM methods of synthesis to achieve rich spectra, vibrato, tremolo, and smooth pitch change effects.

Keywords: additive synthesis, FM synthesis, digital oscillator

1 Introduction

Electronic sound synthesis has been of interest to musicians, electrical engineers and computer scientists for as long as it has been practical. Since the 1970s, synthesizers have evolved from primitive analog machines to sophisticated computer programs. Today, methods such as additive synthesis using Fourier transforms, sampling, physical modeling, frequency modulation and phase distortion can be implemented or emulated using software. The goal of this project is to create an easy-to-use piece of software for exploring multiple methods of sound synthesis using digital oscillators.

2 Background

Three methods are particularly relevant to this project: additive synthesis, FM synthesis, and synthesis by cross-coupled oscillators.

2.1 Additive Synthesis

All periodic functions can be decomposed into sine waves; an audio signal which behaves periodically over long enough time domains, therefore, can be represented by a collection of sine waves with different phase, frequency and amplitude called a spectrum (Moore). Additive synthesis exploits this fact to create audio signals by summing together sine waves or by using Fourier Transforms to convert spectra to audio signals. One implementation of additive synthesis–the one used in this project–is to use multiple digital oscillators to generate waveforms at different frequencies and superimpose them.

2.2 FM Synthesis

Frequency Modulation (FM) synthesis can produce a rich spectrum from just one tone by using it to modulate the the frequency of a second oscillator. This is the same method used to for radio transmission (where the carrier frequency is above the audio band) and vibrato effects (where the modulating frequency is below the audio band). In FM synthesis, the carrier and modulating frequencies are both in the audio band; the result is an output signal which contains the carrier frequency as well as many audible sideband frequencies. By varying the harmonic relationship between the modulating frequency and carrier frequency, and the amplitude of the modulating signal, output signals which are spectrally rich and dynamic over time can be produced (Chowning).

2.3 Cross Coupled Oscillators

In the case of cross coupled oscillators, two oscillators are linked together, the output of each modulating either the amplitude or frequency of the other. This can produce many kinds of temporally varying spectra, from insect-like buzzing sounds to running water to unpredictably shifting noise (Miranda).

3 Development

The purpose of this project is to produce software of some creative value. The final program should be capable of producing a wide variety of sounds

given musical and/or non musical input, and should be easy to manipulate for a user familiar with some of the underlying theory.

3.1 Preliminary Versions

Previous versions were implemented in Python. Early versions rely on hard-coding in values and sequencing statements within the program as methods of input. A later version uses a text based UI to allow external control through a terminal. The versions implemented in Python rely on frequency, envelope and waveform functions defined in the code. Frequency functions take a time argument and return a frequency in Hz. Envelope functions also take a time argument, and return a scalar amplitude. Waveform functions are defined using logic, arithmetic, and/or trigonometric functions; they take a phase argument between 0 and 1 and return a signal amplitude between -1 and 1. In the body of the program, a loop over time increments the phase parameter based on the instantaneous frequency returned by a frequency function and resets it when it exceeds 1. At each time step, waveform functions are called with phase parameters; the returned values are multiplied by envelope functions, and the result is stored in an array. Envelope shapes are visible in Figure 2. Computations use floating point values close to 0; the final audio signal is normalized to a maximum amplitude of 1, then converted to 32-bit signed integers and written to a WAV file. The file can then be played back by an separate media player or audio processing program. Figure 1 shows waveforms produced by an early version.

Later Python versions also have notation functions, capable of generating frequency and envelope functions. Musical information can be input as a string of notation and a few envelope parameters; corresponding frequency and envelope functions are generated and can be used together to synthezise simple chords and melodies.

3.2 Current Version

The current version was built from the ground up in C++. The basic method of synthesis is the same, with one key difference. Waveforms functions are sampled at a rate equal to twice the audio sample rate for a tone at 20 Hz, the low end of human hearing, and stored in memory. This trades a relatively small amount of memory for a drastic speed increase: complex waveform functions are called only once, not at every timestep.

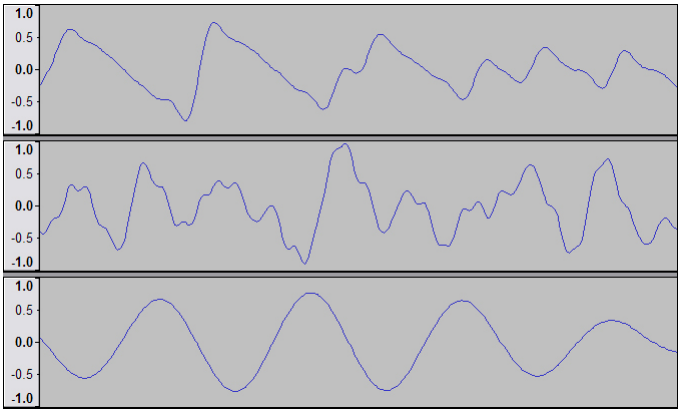

Figure 1: Waveforms produced by additive synthesis

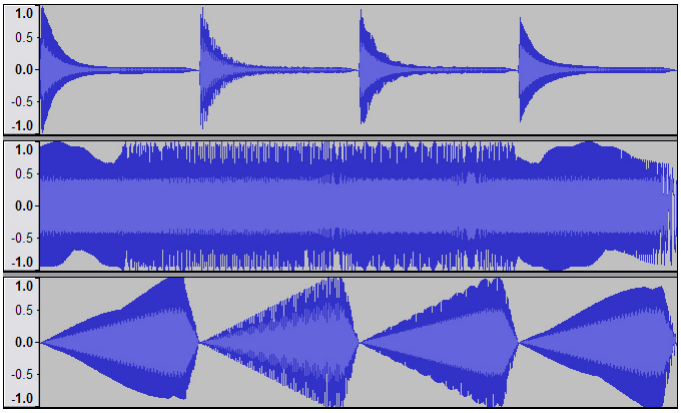

Figure 2: Audio output over several seconds

The primary difference between this version and older versions is the interface. Rather than using a collection of functions which must be stitched together in the body of the program, this version uses a collection of synthesizer element objects, instances of which can be created, altered and connected using a graphic interface. All elements inherit from an abstract class Element; this allows the main loop to polymorphically treat every type of Element equivalently, making it easy to introduce new varieties of Element. An Element has one output and some number of inputs (pointers to the outputs of other elements) as well as some number of constant parameters. Element has a private virtual function compute() which is defined by every subclass; compute() is some function of the Element's input values. Elements also have step() and update() functions, which call compute and store the returned value, and set the Element's output to the stored value, respectively. Elements can be linked together by setting the inputs Elements to the outputs of other Elements. A few basic Elements are: Oscillator, Constant, and Mix$_Sum$.

3.3 Oscillator

Oscillators have a waveform parameter (a pointer to a block of memory containing one of the discrete waveforms mentioned above), an amplitude input, and a frequency input. They also store their own phase parameter. Oscillator::compute() increments the phase parameter based on the audio sample rate and the value appearing at its frequency input and converts the current value of phase to a index in the array pointed to by its waveform parameter. It returns the value of the waveform at that index multiplied by the value appearing at the amplitude input.

3.4 Constant

Constants are a simple Element with no inputs and a single value parameter. Constant::compute() merely returns the value of its parameter.

3.5 Mix$_Sum$

$Mix_Sumshaveavariablenumberofinputs, specifiedatcreation, andagainparameter.Mix_Sum :: compute()returnsthesumofthevaluesattheinputs, multipliedbythegain.$

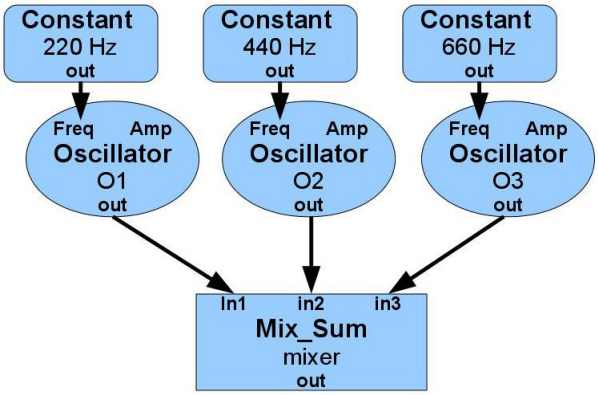

Figure 3: Simple additive synthesis

3.6 Synthesis using Elements

By creating and linking Elements, additive synthesis, FM synthesis, and feedback/cross couple oscillators can all be implemented. Figure 3 and Figure 4 show a few simple constructions.

3.7 More Elements

Two other Elements have been implemented so far: Ramp and Partial. Partial takes a single input and and one integer parameter; Partial::compute() returns the input multiplied by the parameter. Partial can be used as a gain stage, or, as its name implies, to generate partials of a fundamental frequency for additive synthesis. Ramp takes no inputs and four parameters: a start time, start value, end time, and end value. Ramp::compute() interpolates between the two points; Ramp can be used, for example, to generate a bell-like tone. In such a case, a declining Ramp would control both the amplitude of the output Oscillator and the modulating frequency in an FM setup.

4 Testing

Testing has been primarily by ear. This has been sufficient to confirm that the correct audio is being produced. To a smaller extent, visual and spectral analysis of output has been done using Audacity. When debugging file

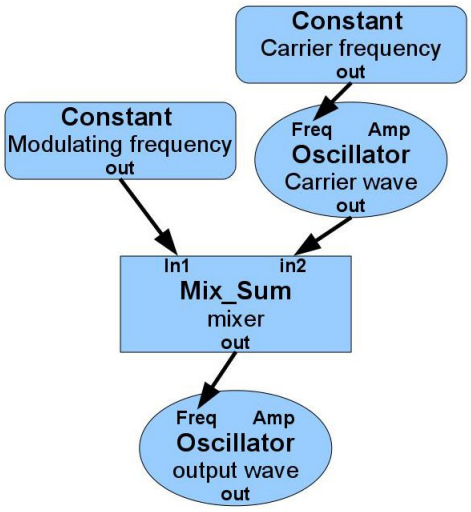

Figure 4: Simple FM synthesis

write, Okteta was used to examine file headers. The Python time module was used for some speed testing in Python versions; speed testing for C++ versions has thus far used the bash time command. The GUI is in progress, but testing consists of manipulating it manually to see where and how it breaks. Elements and their interactions have all been tested with statements in main(); the current version produces audio as expected. In order to test FM implementation, a bell like tone and a sweep of the modulating frequency were generated.

5 Extensions

The current focus is on getting simple Elements to work with an intuitive GUI. Time allowing, notation will be introduced to the C++ version.

6 Results

Those Elements implemented work properly, though the GUI cannot effectively control them yet. The core modular synthesizer, however, is capable of AM, FM, additive synthesis, and feedback.

7 Conclusion

The goal was to produce a creatively useful piece of software. At present, versions of the synthesizer are usable by the author, and can produce music in a range of timbres.

References

[1] Chowning, J., "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation". *Journal of the Audio Engineering Society 21(7)*, pp. 526-534, 1973.

[2] Miranda, E. R., "At the Crossroads of Evolutionary Computation and Music: Self-Programming Synthesizers, Swarm Orchestras and the Origins of Melody", *Evolutionary Computation 12(2)* pp. 137-158, 2004.

[3] Moore, R., *Elements of Computer Music*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[4] Pachet, F., "Description-Based Design of Melodies", *Computer Music Journal 33(4)*, pp. 56-68, 2009.

[5] Thielemann, H., "Untangling Phase and Time in Monophonic Sounds". arXiv:0911.5171v1, 26 Nov 2009.

[6] Valsamakis, N. and Miranda, E. R., "Iterative sound synthesis by means of cross-coupled digital oscillators", *Digital Creativity 16(2)*, pp. 79-92, 2005.

[7] Valsamakis, N. and Miranda, E. R., "Extended waveform segment synthesis, a nonstandard synthesis model for microsound composition", *Proceedings of Sound and Music Computing 05, Salerno (Italy)*, 2005.