TITLE

Alexander Liebendörfer

November 18, 2013

Contents

1 Introduction

1.1 Description

1.1.1 Synthesizers

1.2 Motivation

1.3 Tools

1.3.1 Java and the Java Sound API

1.3.2 Swing

2 Implementation

2.1 Pipes and Modules

To achieve a completely modular synthesizer, we already need to give up a fixed circuit and have to use the module concept. On top of that, because of recursive loops, we need to actually let the data flow between the different modules, and not just sum the effects of the modules when needed. This means we need at least 1 buffer between every module. We also need a clear way to connect those modules, so a pipe system for connecting inputs and outputs of different modules will be necessary. And finally, some way to inject the current MIDI data and duration of every note as well as a way to capture the processed sound for output is needed.

2.1.1 Basic system

In this specific implementation, modules are pure functions, or atleast behave like such towards the outside. A module is always connected to other modules via so-called pipes, which contain the actual data. Modules can then load the data from their input pipes, process it, and dump the result into their output pipes. The main advantage of this system is that it's stable and will work with any wiring setup, even recursive loops. In such closed feed-back systems, there will be a propagation delay depending on the amount of pipes the data has to travel, and so no infinite recursion appears. Splitting data and doing parallel processing before merging it again is also implicitely supported, as every module and pipe only acts locally. One major issue is execution choice. A recursive "pull" query where every module calls the modules it depends on for their output cannot work, infinite recursion is a possibility. Even without recursion, there is always the risk that some part of the circuit did not get updated, which might affect the rest. I haven't found a clean solution to this without drawbacks, the one I chose just makes every module execute once in more or less random order (it is not truely random, in fact it is in order of creation), which in worse case makes a signal wait an extra frame between two modules but guarantees that every module is executed once and only once per frame, and done so in a consistent way.

Injection of user input and retrieval of the sound output is best done with modules dedicated to that. These modules possess only 1 input or output port respectively, and do not by themselves do anything. They just serve as convienient place to connect pipes to, and the main overarching engine can just read and write the contents of their pipes.

2.1.2 Dealing with Time

A tricky problem is dealing with time-sensitive modules, like envelopes or oscillators. The input does not implicitely declare it's source, so there needs to be a way for the pipes to also carry starting time of a signal. For this, I implemented a separate counter in every pipe, which showed the time at which that signal started. When a module is done with processing and copies the signal data into its output, it also copies in the counter. A negative time can be used as a marker that the pipe is inactive and that module doesn't even need to process anything. Every module can define a special pipe which is taken as reference for the source time, which is especially important when conflicts are possible. If a module does not define such a pipe, the highest starting time (the most recent signal) of all is assumed to be the reference.

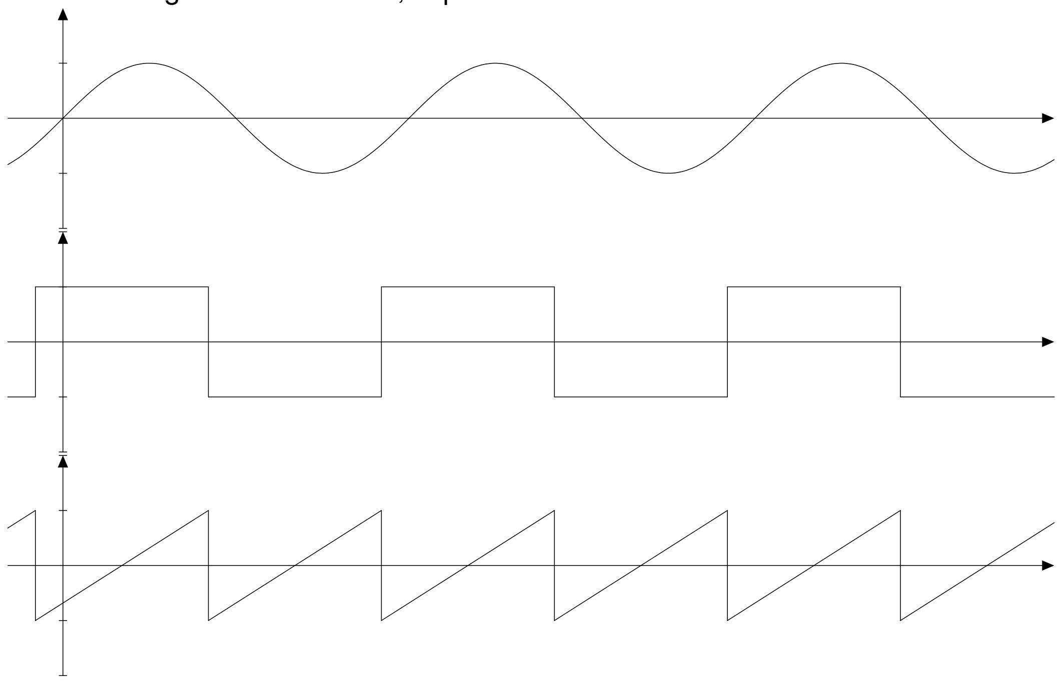
2.1.3 Stereo/Mono and Polyphony support

2.2 Oscillators

Oscillators are generators that can create a periodic waveform. They are the basis of additive and subtractive synthesis by creating a basic wave to be shaped and modified by every other module. Usually osillators create a few types of mathematically well-defined primitive waveforms, like sine or square waves. Apart from the wave function, they also possess two parameters, frequency and phase. Frequency describes the amount of oscillations per second, and a difference in frequency is percieved as a difference in pitch. Phase is very difficult to percieve audibly in most cases, it is the initial offset the oscillation started with. Contrarily to frequency, phase has a bounded range, usually from 0 to 2π or from 0 to 1, depending on the implementation.

In this synthesizer, the 3 most fundamental waveforms are supported. These are the sine wave, the ”square wave” or pulse wave, and the popular sawtooth-wave, also called saw-wave.

Figure 1: Sine wave, Square/Pulse wave and Sawtooth wave



2.2.1 Basic implementation

As already noted, oscillators only require two parameters, phase and frequency. It is quite easy to deal with stereo behavior, as the channels are independent, one can simply first calculate the left side and then the right side. Oscillators also depend heavily on the global time and on the pipe starting time. Through this the current time since starting to oscillate can be calculated trivially, and it is generally a good idea to do modulo the period of the wave as well. Afterwards, the resulting signal is calculated differently depending on the type.

Figure 2: Basic oscillator function implementations

```
double x = time * frequency + phase;
switch (osc_type)
{
        case SINE_WAVE:
                return Math.sin(x*2*Math.PI);

        case SQUARE_WAVE:
                if (x <= 0.5)
                {
                    return 1;
                }
                else
                {
                    return -1;
                }

        case SAW_WAVE:
                return x - Math.floor(x);
}
```

2.2.2 Anti-Aliased implementation

The previous approach worked fine for sine waves, but in both other waveforms there is a discontinuity somewhere. In the frequency range, this translates to an infinite frequency, inducing aliasing, as the Nyquist frequency cannot be infinite. There are many, many approaches to fixing this problem in general, it is a recurring obstacle in everything from 3D-graphics to font rendering, and it affects synthesis as well. Luckily, the two waveforms we need are special; their frequencies are easily represented as infinite sums. On top of that, we do not need a particularly fast solution, so we can settle for a mathematically simple and complete one.

The saw wave, in the frequency domain, is nothing but the sum of every harmonic with a 1/x decaying amplitude. (f = the base frequency, F_N = the Nyquist frequency)

$$H(f) = \frac{2}{\pi} \sum_{k=1}^{\infty} \frac{\sin(k * f)}{k} \tag{1}$$

Of course, instead of summing to infinity, we could stop summing as soon as $k * f > F_N$, therefore only including overtones under the Nyquist frequency.

$$H(f) = \frac{2}{\pi} \sum_{k=1}^{\frac{F_N}{f}} \frac{\sin(k * f)}{k} \tag{2}$$

Note: The $\frac{2}{\pi}$ normalises the result between -1 and 1.

The same can be done for square waves. Square waves have nearly an identical frequency sum, except only odd harmonics have a non-zero amplitude (this also doubles the normalizing factor, since the range is still -1 to 1 but only half the elements are summed).

$$H(f) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{\sin((2 * k - 1) * f)}{2 * k - 1} \tag{3}$$

becomes

$$H(f) = \frac{4}{\pi} \sum_{k=1}^{\frac{F_N}{f}} \frac{\sin((2 * k - 1) * f)}{2 * k - 1} \tag{4}$$

which again, only includes all parts of the square wave which are under the Nyquist frequency; a perfect solution, if we have the luxury of enough computing time to calculate this sum (which, for this synthesizer, we do).

In code, the final result looks like this:

Figure 3: Anti-Aliased oscillator function implementations

```
double x = time * frequency + phase;
switch (osc_type)
{
        case SINE_WAVE:
                return Math.sin(x*2*Math.PI);

        case SAW_WAVE:
                result = 0;
                // Sawtooth = infinite sum of all harmonics with A=1/n for
                    nth harmonic
                // Source: http://en.wikipedia.org/wiki/Sawtooth_wave
                for (int k=1; k*freq<Constants.SAMPLING_RATE/2; k++)
                {
                        result += Math.sin(x*2*Math.PI*k)/k;
                }
                // Keep result -1 <= x <= 1, not -1.0903783642160645 because
                    of imprecision
                return Math.min(1, Math.max(-1, 2*result/Math.PI));

        case SQUARE_WAVE:
                result = 0;
                // Square = infinite sum of odd harmonics with A=1/n for nth
                    harmonic
                // Source:
                    http://en.wikipedia.org/wiki/Square_wave#Examining_the_square_wave
                for (int k=1; k*freq<Constants.SAMPLING_RATE/2; k+=2)
                {
                        result += Math.sin(x*2*Math.PI*k)/k;
                }
                // Keep result -1 <= x <= 1, not -1.0903783642160645 because
                    of imprecision
                return Math.min(1, Math.max(-1, 4*result/Math.PI));
}
```

This code gives off perfect anti-aliasing and is fast enough for use in our synthesizer.

2.3 Filters

2.3.1 Implementation with FFT

2.3.2 Implementation using (2-pole) recursive filters

2.4 Other Primitives

2.4.1 Constants

2.4.2 Merger and Splitter

2.4.3 Adder and Multiplier

2.4.4 Range Modifier

2.5 GUI

3 Results

3.1 Capabilities and Limitations of this program

3.2 Lessons learned