Developing a Digital Synthesizer in C++

Peter Goldsborough
8A

Supervised by Prof. Martin Kastner

BG | BRG St.Martin
St.Martiner Straße 7
9500 Villach Austria

January 12, 2015

Abstract

I like KFC

Acknowledgements

I want to thank Professor Martin Kastner for supervising my thesis. His advice on how to structure and plan this thesis were very insightful. I felt like I had his full confidence throughout my final year.
I would also like to express my gratitude towards John Cooper, who helped me and gave me critical pieces of advice at critical times, about digital signal processing, about computer science and about writing this thesis. Thank you for being such a nice contact.

Also special thanks to Jari Kleimola, whose efforts to help out a random stranger from the internet I greatly appreciate. You are an excellent example of how the internet has made the world a smaller and friendlier place.

Villach, January 12, 2015,

Peter Goldsborough

Contents

Introduction

The dawn of the digital age has brought fundamental changes to numerous areas of science as well as our everyday lives. In general, one may observe that many phenomena and aspects of the physical world — hardware — have been replaced by virtual simulations — software. This is also true for the realm of electronic music production. Whereas music synthesizers such as the legendary "Moog" previously produced sounds through analog circuitry, digital synthesizers can nowadays be implemented with common programming languages such as C++.

The question that remains, of course, is: how? Where does one start on day zero? What knowledge is required? What resources are available off and online? What are the best practices? How can somebody with no previous experience in generating sound through software build an entire synthesis system, from scratch? Even though building a software synthesizer is itself a big enough mountain to climb, the real difficulty lies in finding the path that leads one up this mountain.

Two publications that are especially valuable when attempting to build a software synthesizer are *BasicSynth: Creating a Music Synthesizer in Software*, by Daniel R. Mitchell, and *The Scientist and Engineer's Guide to Digital Signal Processing*, by Steven W. Smith. The first book is a practical, hands-on guide to understanding and implementing the concepts behind digital synthesizers. It provides simple, straight-forward explanations as well as pseudo-code examples on a variety of subjects that are also discussed in this thesis. The real value of Mitchell's publication is that it simplifies and condenses years of industry practices and digital signal processing (DSP) knowledge into a single book. However, *BasicSynth* abstracts certain topics, such as digital filters, too much. Where its explanations do not suffice for a full, or at least practical, understanding, *The Scientist and Engineer's Guide to Digital Signal Processing* is an excellent alternative. Steven W. Smith's book explains concepts of Digital Signal Processing (DSP) in great detail, and is to some extent on the opposite end of the spectrum of explanation depth, when compared to *BasicSynth*. It should be noted, however, that Smith's publication is not at all focused on audio processing or digital music, but on DSP in general. (Mitchell, 2008) (Smith, 1999)

This thesis is intended to dicuss the most important steps on the path from zero lines of code to a full synthesis system, implemented in C++. While providing many programming snippets and even full class definitions[1], a special focus will be put on explaining the fundamental ideas behind the programming, which very often lie in the realm of mathematics or general digital signal processing. The reason for this is that once a theoretical understanding has been established, the practical implementation becomes a trivial task.

[1] C++ program samples of up to 50 lines of code are displayed in-line with the text, longer samples are found in Appendix A.

The first chapter will introduce some rudimentary concepts and phenomena of sound in the digital realm, as well as explain how digital sound differs to its analog counterpart. Subsequent chapters examine, among other things, how computer music is generated, modulated, filtered, synthesized and finally made audible. Images of sound samples in the time and frequency domain, as well as diagrams to abstract certain concepts, will be provided. Moreover, programming relationships, such as inheritance diagrams, between parts of the synthesizer implemented for this thesis, called *Anthem*, are also displayed when relevant. Lastly, excerpts of email or online forum exchanges will be given when they contributed to the necessary knowledge.

It should be made clear that this thesis is not a "tutorial" on how to program a complete synthesis system in C++. It is also not designed to be a reference for theoretical concepts of digital signal processing. Rather, practice and theory will be combined in the most pragmatic way possible.

Chapter 1

From Analog to Digital

Our everyday experience of sound is an entirely analog one. When a physical object emits or reflects a sound wave into space and towards our ears, the signal produced consists of an infinite set of values, spaced apart in infinitesimal intervals. Due to the fact that such a signal has an amplitude value at every single point in time, it is called a continuous signal. (Smith, 1999, p. 11) Figure 1.1 displays the continuous representation of a sine wave.

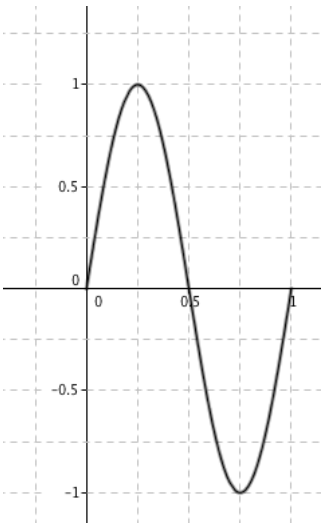


Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

While continuous signals and the idea of an infinite, uncountable set of values are easy to model in mathematics and physics — the analog world, computers — in the digital world — effectively have no means by which to represent something that is infinite, since computer memory is a finite resource. Therefore, signals in the digital domain are discrete, meaning they are composed of periodic *samples*. A sample is a discrete recording of a continuous signal's amplitude, taken in a constant time interval. The process by which a continuous signal is converted to a discrete signal is called *quantization*, *digitization* or simply *analog-to-digital-conversion* (Smith, 1999, p. 35-36). Quantization essentially converts an analog function of amplitude to a digital function of location in computer memory over time (Burk, Polansky, Repetto, Roberts and Rockmore, 2011, Section 2.1). The reverse process of converting discrete samples to a continuous signal is called *digital-to-analog-conversion*. Figure 1.2 shows the discrete representation of a sine wave, the same signal that was previously shown as a continuous signal in Figure 1.1. (Mitchell, 2008, p. 16-17)

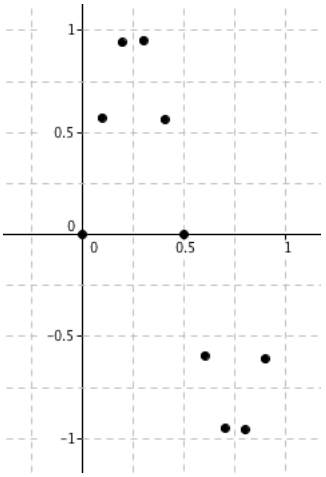


Figure 1.2: The discrete representation of a typical sine wave.

1.1 Sample Rate

The sample rate (often referred to as sampling rate or sampling frequency), commonly denoted by f_s, is the rate at which samples of a continuous signal are taken to quantize it. The value of the sample rate is measured in Hertz (Hz) or samples per second. Common values for audio sampling rates are 44.1 kHz, a frequency originally chosen by Sony in 1979 that is still used for Compact Discs, and 48 kHz, the standard audio sampling rate used today. (Mitchell, 2008, p. 18) (Colletti, 2013) The reciprocal of the sample rate yields the sampling interval, denoted by T_s and measured in seconds, which is the time period after which a single sample is taken from a continuous signal:

$$T_s = \frac{1}{f_s} \tag{1.1}$$

The reciprocal of the sample interval again yields the sampling rate:

$$f_s = \frac{1}{T_s} \tag{1.2}$$

1.2 Nyquist Limit

The sample rate also determines the range of frequencies that can be represented by a digital sound system. The reason for this is that only frequencies that are less than or equal to one half of the sampling rate can be "properly sampled". To sample a signal "properly" means to be able to reconstruct a continuous signal, given a set of discrete samples, exactly, i.e. without any *quantization errors*. This is only possible if the frequency of a signal allows at least one sample per cycle to be taken above the equilibrium and at least one sample below. The value of one half of the sample rate is called the *Nyquist frequency* or *Nyquist limit*, named after Harry Nyquist, who first described the Nyquist limit and associated phenomena together with Claude Shannon in the 1940s, stating that "a continuous signal can be properly sampled, only if it does not contain frequency components above one half the sampling rate". Any frequencies above the Nyquist limit lead to *aliasing*, which is discussed in the next section. (Smith, 1999, p. 40)
Given the definition of the Nyquist limit and considering the fact that the limit of human hearing is approximately 20 kHz (Cutnell & Johnson, 1998, p. 466), the reason for which the two most common audio sample rates are 40 kHz and above is clear: they were chosen to allow the "proper" representation of the entire range of frequencies audible to humans, since a sample rate of 40 kHz or higher meets the Nyquist requirement of a sample rate at least twice the maximum frequency component of the signal to be sampled (the Nyquist limit), in this case ca. 20 Khz.

1.3 Aliasing

When a signal's frequency exceeds the Nyquist limit, it is said to produce an *alias*, a new signal with a different frequency that is indistinguishable from the original signal when sampled. This is due to the fact that a signal with a frequency component above the Nyquist limit no longer has one sample taken above and one below the zero level for each cycle, but at arbitrary points of the original signal. When these points are connected, they yield an entirely different signal. For example, if the sinusoid depicted in Figure 1.3, with a frequency of 4 Hz, is sampled at a sample rate of 5 Hertz, shown in Figure 1.4, meaning the frequency of the continuous signal is higher than the Nyquist limit (here 2.5 Hz), the reconstructed signal, approximated in Figure 1.5, will look completely different from the original sinusoid. "This phenomenon of [signals] changing frequency during sampling is called aliasing, [...] an example of improper sampling". (Smith, 1999, p. 40)

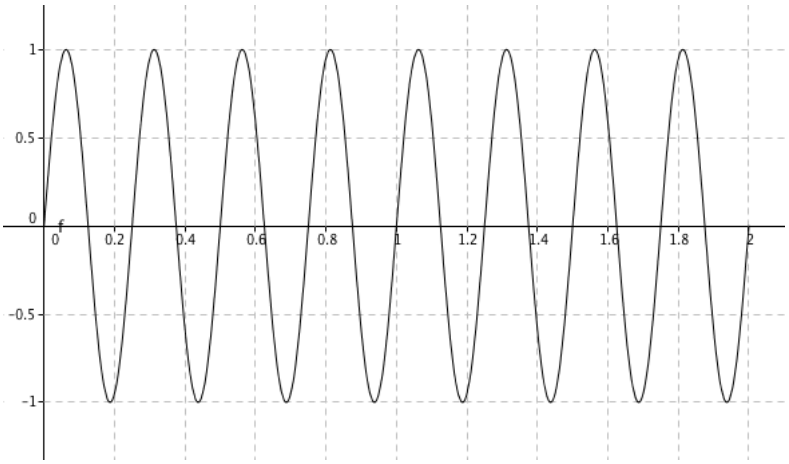


Figure 1.3: A sinusoid with a frequency of 4 Hz.

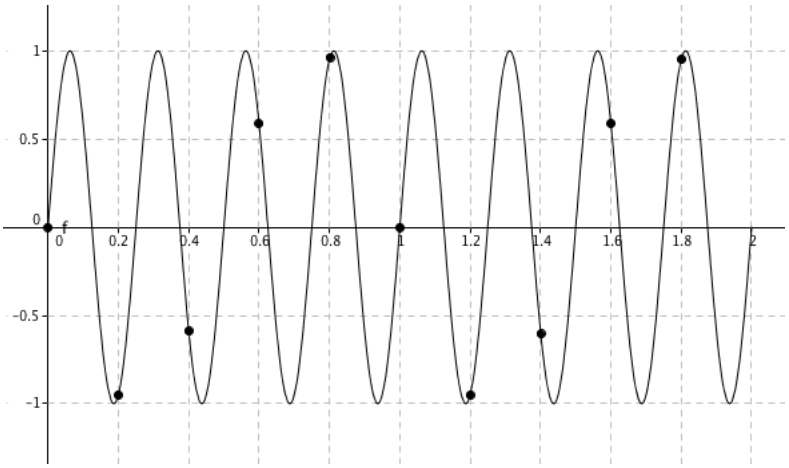


Figure 1.4: A sinusoid with a frequency of 4 Hz, sampled at a sample rate of 5 Hz. According to the Nyquist Theorem, this signal is sampled improperly, as the frequency of the continuous signal is not less than or equal to one half of the sample rate, in this case 2.5 Hz.

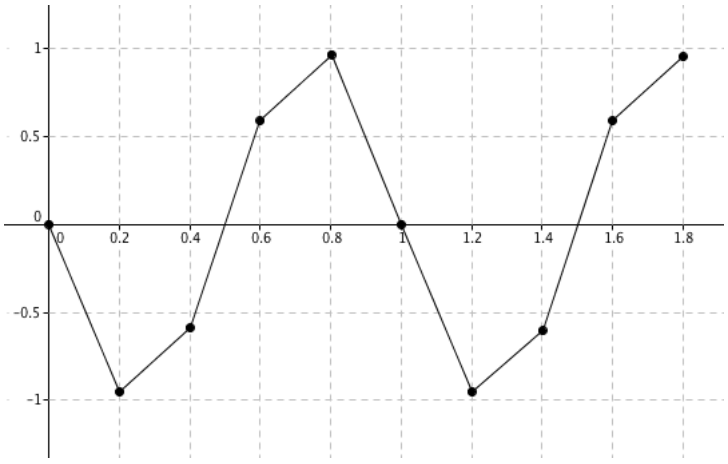


Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper sampling. These signals are said to be *aliases* of each other.

Chapter 2

Generating Sound

The following sections will outline how digital sound can be generated in theory and implemented in practice, using the C++ programming language.

2.1 Simple Waveforms

The simplest possible audio waveform is a sine wave. As a function of time, it can be mathematically represented by Equation 2.1, where A is the maximum amplitude of the signal, f the frequency in Hertz and ϕ an initial phase offset in radians.

$$f_s(t) = A\sin(2\pi ft + \phi) \tag{2.1}$$

A computer program to compute the values of a sine wave with a variable duration, implemented in C++, is shown in Listing A.1. Another waveform similar to the sine wave is the cosine wave, which differs only in a phase offset of 90°or $\frac{\pi}{2}$ radians, as shown in Equation 2.2.

$$f_c(t) = A\cos(2\pi ft + \phi) = A\sin(2\pi ft + \phi + \frac{\pi}{2}) \tag{2.2}$$

Therefore, the program from Listing A.1 could be modified to compute a cosine wave by changing line 22 from:

```
22 | double phase = 0;
```

to

```
22 | double phase = pi/2.0;
```

2.2 Complex Waveforms

Now that the process of creating simple sine and cosine waves has been discussed, the generation of more complex waveforms can be examined. Generally, there are two methods by which complex waveforms can be created in a digital synthesis system: mathematical calculation or additive synthesis.

2.2.1 Mathematical Calculation of Complex Waveforms

In the first case — mathematical calculation, waveforms are computed according to certain mathematical formulae and thus yield *perfect* or *exact* waveforms, such as a square wave that

```
1  double* square(const unsigned int period)
2  {
3      // the sample buffer
4      double * buffer = new double [period + 1];
5
6      // time for one sample
7      double sampleTime = 1.0 / period;
8
9      // the midpoint of the period
10     double mid = 0.5;
11
12     double value = 0;
13
14     // fill the sample buffer
15     for (int n = 0; n < period; n++)
16     {
17         buffer[n] = (value < mid) ? -1 : 1;
18
19         value += sampleTime;
20     }
21
22     return buffer;
23 }
```

Table 2.1: C++ code to generate and return one period of a square wave, where `period` is the period duration in samples. Note that this function increments in sample time, measured in seconds, rather than actual samples. This prevents a one-sample quantization error at the mid-point, since time can always be halved whereas a sample is a fixed entity and cannot be broken down any further.

is equal to its maximum amplitude exactly one half of a period and equal to its minimum amplitude for the rest of the period. While these waveforms produce a very crisp and clear sound, they are rarely found in nature due to their degree of perfection and are consequently rather undesirable for a music synthesizer. Nevertheless, they are considerably useful for modulating other signals, as tiny acoustical imperfections such as those found in additively synthesized waveforms can result in unwanted distortion which is not encountered when using mathematically calculated waveforms. Therefore, exact waveforms are the best choice for modulation sources such as Low Frequency Oscillators (LFOs), which are discussed in later chapters. (Mitchell, 2008, p. 71)
The following paragraphs will analyze how three of the most common waveforms found in digital synthesizers — the square, the sawtooth and the triangle wave — can be generated via mathematical calculation.

Square Waves

Ideally, a square wave is equal to its maximum amplitude for exactly one half of a period and equal to its minimum amplitude for the other half of the same period. Equation 2.3 shows how to calculate a single period of a square wave, where the independent variable t as well as the period T can be either in samples or in seconds. A mathematical Equation for a full, periodic square wave function is given by Equation 2.4, where t is time in seconds and the frequency f in Hertz. An equivalent C++ computer program is shown in Table 2.1.

$$f(t) = \begin{cases} 1, & \text{if } 0 \leq t < \dfrac{T}{2} \\ -1, & \text{if } \dfrac{T}{2} \leq t < T \end{cases} \tag{2.3}$$

$$f(t) = \begin{cases} 1, & \text{if } \sin(2\pi f t) > 0 \\ -1, & \text{otherwise} \end{cases} \tag{2.4}$$

Sawtooth Waves

An ideal sawtooth wave descends from its maximum amplitude to its minimum amplitude linearly before jumping back to the maximum amplitude at the beginning of the next period. A

```
1  double* sawtooth(const unsigned int period)
2  {
3      // the sample buffer
4      double * buffer = new double [period];
5
6      // how much we must decrement the
7      // index by at each iteration
8      double incr = -2.0 / period;
9
10     double value = 1;
11
12     for (int n = 0; n < period; n++)
13     {
14         buffer[n] = value;
15
16         value += incr;
17     }
18
19     return buffer;
20 }
```

Table 2.2: C++ code to generate one period of a sawtooth wave function, where `period` is the period duration in samples.

mathematical equation for a single period of such a sawtooth wave function, calculated directly from the phase, is given by Equation 2.5 (Mitchell, 2008, p. 68). Alternatively, the function can depend on time or on samples, as shown by Equation 2.6, where T is the period. A computer program to compute one period of a sawtooth wave is given in Table 2.2.

$$f(\phi) = \left\{ -\frac{\phi}{\pi} + 1, \quad \text{if } 0 \leq \phi < 2\pi \right. \tag{2.5}$$

$$f(t) = \left\{ -\frac{2t}{T} + 1, \quad \text{if } 0 \leq t < 1 \right. \tag{2.6}$$

Triangle waves

A triangle wave can be seen as a linear sine wave. It increments from its minimum amplitude to its maximum amplitude linearly one half of a period and decrements back to the minimum during the other half. Simply put, "[a] triangle wave is a linear increment or decrement that switches direction every π radians" (Mitchell, 2008, p. 69). A mathematical definition for one period of a triangle wave is given by Equation 2.7, where ϕ is the phase in radians. If ϕ is kept in the range of $[-\pi; \pi]$ rather than the usual range of $[0; 2\pi]$, the subtraction of π can be eliminated, yielding Equation 2.8. If the dependent variable is time in seconds or samples, Equation 2.9 can be used for a range of $[0; T]$, where T is the period, and Equation 2.10 for a range of $[-\frac{T}{2}; \frac{T}{2}]$. A C++ implementation is shown in Table 2.3.

$$f(\phi) = \left\{ 1 - \frac{2|\phi - \pi|}{\pi}, \quad \text{if } 0 \leq \phi < 2\pi \right. \tag{2.7}$$

$$f(\phi) = \left\{ 1 - \frac{2|\phi|}{\pi}, \quad \text{if } -\pi \leq \phi < \pi \right. \tag{2.8}$$

$$f(t) = \left\{ 1 - \frac{4|t - \frac{T}{2}|}{T}, \quad \text{if } 0 \leq t < 1 \right. \tag{2.9}$$

$$f(t) = \left\{ 1 - \frac{4|t|}{T}, \quad \text{if } -\frac{T}{2} \leq t < \frac{T}{2} \right. \tag{2.10}$$

```
double* triangle(const unsigned int period) const
{
    double* buffer = new double[period];

    double value = -1;

    // 4.0 because we're incrementing/decrementing
    // half the period and the range is 2, so it's
    // actually 2 / period / 2.
    double incr = 4.0 / period;

    // Boolean to indicate direction
    bool reachedMid = false;

    for (unsigned int n = 0; n < period; n++)
    {
        wt[n] = value;

        // Increment or decrement depending
        // on the current direction
        value += (reachedMid) ? -incr : incr;

        // Change direction every time
        // the value hits a maximum
        if (value >= 1 || value <= -1)
        { reachedMid = !reachedMid; }
    }

    return buffer;
}
```

Table 2.3: C++ program to compute one period of a triangle wave.

2.2.2 Additive Synthesis

The second method of generating complex waveforms, additive synthesis, produces waveforms that, despite not being mathematically perfect, are closer to the waveforms found naturally. This method involves the summation of a theoretically infinite, practically finite set of sine and cosine waves with varying parameters. Additive synthesis is often called Fourier Synthesis, after the 18th century French scientist, Joseph Fourier, who first described the process and associated phenomena of summing sine and cosine waves to produce complex waveforms. This calculation of a complex, periodic waveform from a sum of sine and cosine functions is also referred to as a Fourier Transform or a Fourier Series, both part of the Fourier Theorem. In a Fourier Series, a single sine or cosine component is either called a harmonic, an overtone or a partial. All three name the same idea of a waveform with a frequency that is an *integer multiple* of some fundamental pitch or frequency. (Mitchell, 2008, p. 64) Throughout this thesis the term *partial* will be preferred.
Equation 2.13 gives the general definition of a discrete Fourier Transform and Equation 2.14 shows a simplified version of Equation 2.13. Table 2.4 presents a C++ struct to represent a single partial and Listing A.2 a piece of C++ code to compute one period of any Fourier Series.

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty}(a_n \cos(\omega n t) + b_n \sin(\omega n t))$$

Equation 2.13: Formula to calculate an infinite Fourier series, where $\frac{a_n}{2}$ is the center amplitude, a_n and b_n the partial amplitudes and ω the angular frequency, which is equal to $2\pi f$.

The following paragraphs will examine how the three waveforms presented in Section 2.2.1 can be synthesized additively.

$$f(t) = \sum_{n=1}^{N} a_n \sin(\omega n t + \phi_n)$$

Equation 2.14: Simplificiation of Equation 2.13. Note the change from a computationally impossible infinite series to a more practical finite series. Because a cosine wave is a sine wave shifted by 90°or $\frac{\pi}{2}$ radians, the $\cos$ function can be eliminated and replaced by an appropiate $\sin$ function with a phase shift ϕ_n.

```
1  struct Partial
2  {
3    Partial(unsigned short number, double ampl, double phsOffs = 0)
4      : num(number), amp(ampl), phaseOffs(phsOffs)
5    { }
6
7    /*! The Partial's number, stays const. */
8    const unsigned short num;
9
10   /*! The amplitude value. */
11   double amp;
12
13   /*! A phase offset */
14   double phaseOffs;
15 };
```

Table 2.4: C++ code to represent a single partial in a Fourier Series.

Square Waves

When speaking of additive synthesis, a square wave is the result of summing all odd-numbered partials (3rd, 5th, 7th etc.) at a respective amplitude equal to the reciprocal of their partial number ($\frac{1}{3}$, $\frac{1}{5}$, $\frac{1}{7}$ etc.). The amplitude of each partial must decrease with increasing partial numbers to prevent amplitude overflow. A mathematical equation for such a square wave with N partials is given by Equation 2.11, where $2n-1$ makes the series use only odd partials. A good maximum number of partials N for near-perfect but still naturally sounding waveforms is 64, a value determined empirically. Higher numbers have not been found to produce significant improvements in sound quality. Table 2.5 displays the C++ code needed to produce one period of a square wave in conjuction with the `additive` function from Listing A.2. Figure 2.1 shows the result of summing 2, 4, 8, 16, 32 and finally 64 partials.

$$f(t) = \sum_{n=1}^{N} \frac{1}{2n-1} \sin(\omega(2n-1)t) \tag{2.11}$$

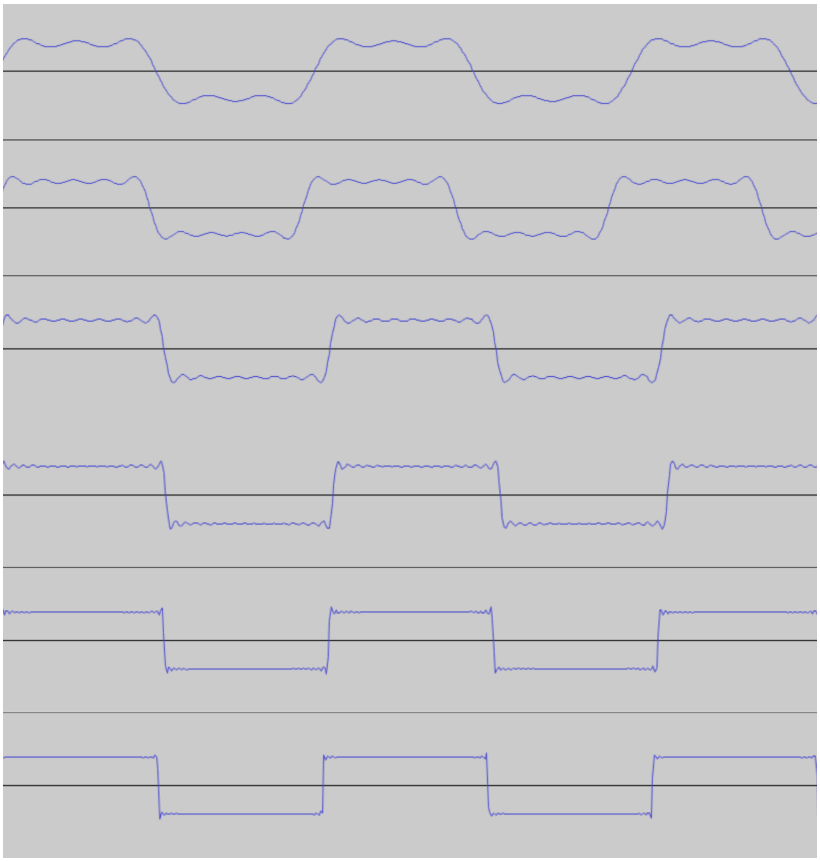


Figure 2.1: Square waves with 2, 4, 8, 16, 32 and 64 partials.

```
1 std::vector<Partial> vec;
2
3 for (int i = 1; i <= 128; i += 2)
4 {
5   vec.push_back(Partial(i, 1.0/i));
6 }
7
8 double* buffer = additive(vec.begin(), vec.end(), 48000)
```

Table 2.5: C++ code for a square wave with 64 partials.

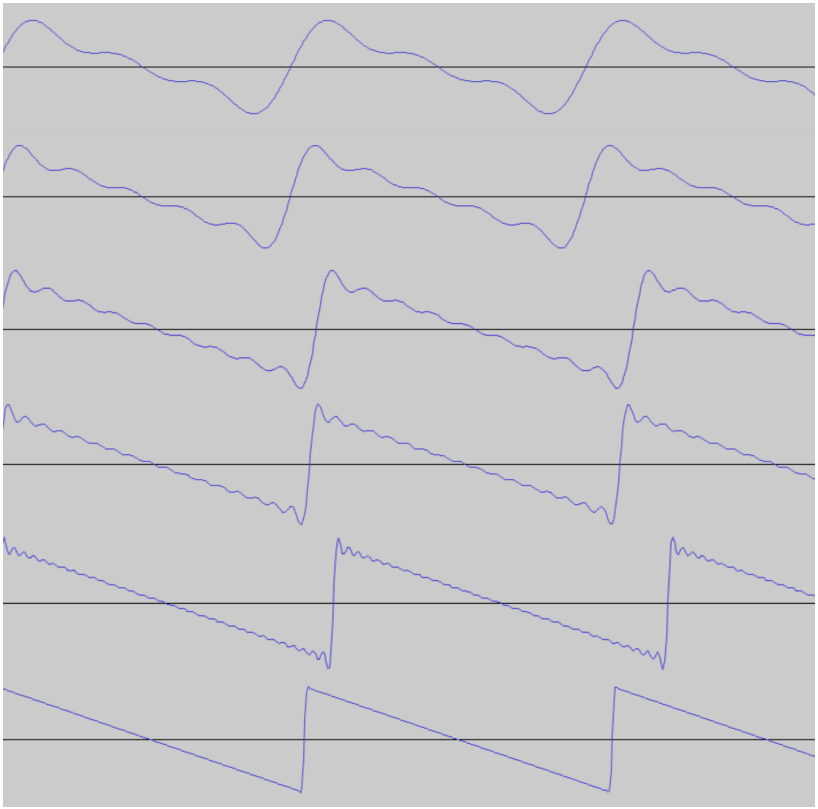


Figure 2.2: Sawtooth waves with 2, 4, 8, 16, 32 and 64 partials.

```
1 std::vector<Partial> vec;
2
3 for (int i = 1; i < 64; ++i)
4 {
5   vec.push_back(Partial(i, 1.0/i));
6 }
7
8 double* buffer = additive(vec.begin(), vec.end(), 48000)
```

Table 2.6: C++ code for a sawtooth wave with 64 partials.

Sawtooth Waves

A sawtooth wave is slightly simpler to create through additive synthesis, as it requires the summation of every partial rather than only the odd-numbered ones. The respective amplitude is again the reciprocal of the partial number. Equation 2.12 gives a mathematical definition for a sawtooth wave, Figure 2.2 displays sawtooth functions with various partial numbers and Table 2.6 shows C++ code to generate such functions.

$$f(t) = \sum_{n=1}^{N} \frac{1}{n} \sin(\omega n t) \tag{2.12}$$

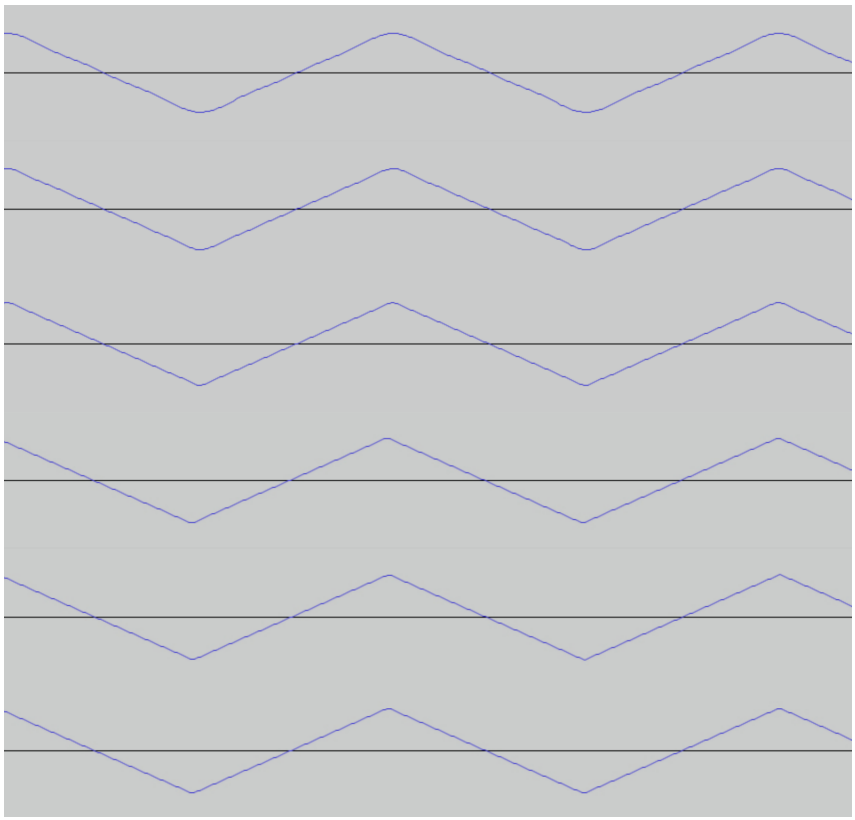


Figure 2.3: Triangle waves with 2, 4, 8, 16, 32 and 64 partials. Note that already 2 partials produce a very good approximation of a triangle wave.

Triangle Waves

The process of generating triangle waves additively differs from previous waveforms. The amplitude of each partial is no longer the reciprocal of the partial number, $\frac{1}{n}$, but now of the partial number squared: $\frac{1}{n^2}$. Moreover, the sign of the amplitude alternates for each partial in the series. As for square waves, only odd-numbered partials are used. Mathematically, such a triangle wave is defined as shown in Equation 2.13 or, more concisely, in Equation 2.14. Figure 2.3 displays such a triangle wave with various partial numbers and Table 2.7 implements C++ code to compute a triangle wave.

$$f(t) = \sum_{n=1}^{\frac{N}{2}} \frac{1}{(4n-3)^2} \sin(\omega(4n-3)t) - \frac{1}{(4n-1)^2} \sin(\omega(4n-1)t) \tag{2.13}$$

$$f(t) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)^2} \sin(\omega(2n+1)t) \tag{2.14}$$

```
1  std::vector<Partial> vec;
2
3  double amp = -1;
4
5  for(int i = 1; i <= 128; i += 2)
6  {
7    amp = (amp > 0) ? (-1.0/(i*i)) : (1.0/(i*i));
8
9    vec.push_back(Partial(i,amp));
10 }
11
12 double* buffer = additive(vec.begin(), vec.end(), 48000);
```

Table 2.7: C++ code for a triangle wave with 64 partials.

2.3 The Gibbs Phenomenon and the Lanczos Sigma Factor

Examining Figure 2.4, which displays an additively synthesized square wave function with 64 partials, one may observe that additive synthesis, the summation of sine waves to produce complex waveforms, produces an overshoot — slight ripples or "horns" — at the ends of each peak and trough of a waveform. This is known as the Gibbs Phenomenon, named after the American scientist Josiah Willard Gibbs who first described it in 1898, and is "the result of summing a finite series of partials rather than the infinite series of partials specified by the Fourier transform" (Mitchell, 2008, p. 67). Acoustically, this artifact has little influence on the sound of the produced waveform. However, for modulation, this imperfection may render the resulting sound unsatisfactory. A common way to reduce the Gibbs Phenomenon is to apply the Lanczos Sigma (σ) Factor to each partial of a Fourier Series. This is often called *sigma-approximation*. The definition of the Lanczos Sigma Factor is given in Equation 2.15, where n is the current partial number and M the total number of partials in a Fourier Summation. In the `additive` function shown in Listing A.2, the Lanczos Sigma Factor is implemented in lines 48 to 54. Figure 2.5 shows the same square wave from Figure 2.4 after sigma-approximation.

$$\sigma = sinc(\frac{n\pi}{M}) = \frac{\sin(\frac{n\pi}{M})}{\frac{n\pi}{M}} \tag{2.15}$$

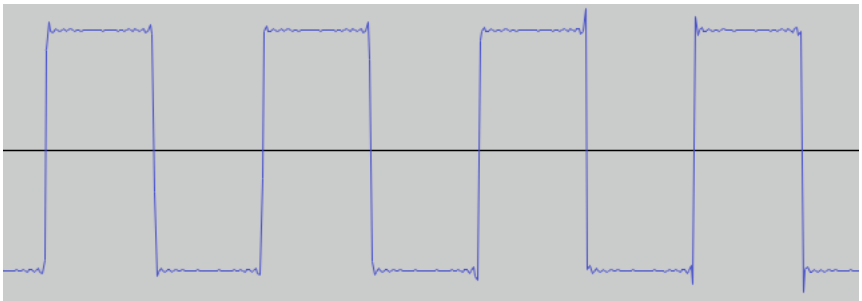


Figure 2.4: An additively synthesized square wave with 64 partials before sigma-approximation.

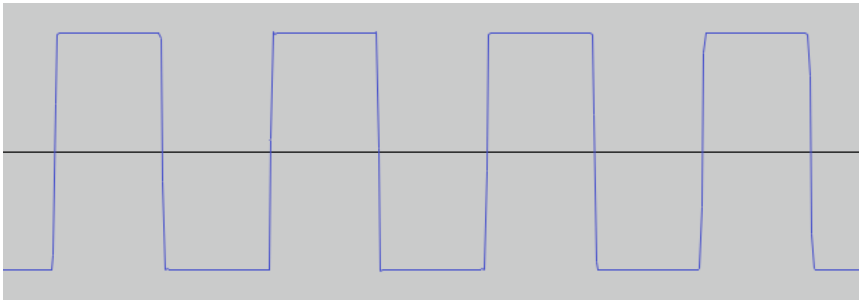


Figure 2.5: An additively synthesized square wave with 64 partials after sigma-approximation.

2.4 Wavetables

Following the discussion of the creation of complex waveform, the two options for playing back waveforms in a digital synthesis system must be examined: continuous real-time calculation of waveform samples or lookup from a table that has been calculated once and then written to disk — a so-called Wavetable. To keep matters short, the second method was found to be computationally more efficient and thus the better choice, as memory is a more easily expended resource than computational speed.

2.4.1 Implementation

A Wavetable is a table, practically speaking an array, in which pre-calculated waveform amplitude values are stored for lookup. The main benefit of a Wavetable is that individual samples need not be computed periodically and in real-time. Rather, samples can be retrieved simply by dereferencing and subsequently incrementing a Wavetable index. If the Wavetable holds sample values for a one-second period, the frequency of the waveform can be adjusted during playback by multiplying the increment value by some factor other than one. "For example, if [one] increment[s] by two [instead of one], [one can] scan the table in half the time and produce a frequency twice the original frequency. Other increment values will yield proportionate frequencies." (Mitchell, 2008, p. 80-81) The *fundamental increment* of a Wavetable refers to the value by which a table index must be incremented after each sample to traverse a waveform at a frequency of 1 Hz. A formula to calculate the fundamental increment i_{fund} is shown in Equation 2.16, where L is the Wavetable length and the f_s the sample rate. To alter the frequency f of the played-back waveform, Equation 2.17 can be used to calculate the appropriate increment value i.

$$i_{fund} = \frac{L}{f_s} \tag{2.16}$$

$$i = i_{fund} \cdot f = \frac{L}{f_s} \cdot f \tag{2.17}$$

2.4.2 Interpolation

For almost all configurations of frequencies, table lengths and sample rates, the table index i produced by Equation 2.17 will not be an integer. Since using a floating point number as an index for an array is a syntactically illegal operation in C++, there are two options. The first is to truncate or round the table index to an integer, thus "introducing a quantization error

```
sample = table[integral] + ((table[integral + 1] - table[integral]) * fractional)
```

Table 2.8: An interplation algorithm in pseudo-code.

```
template<class T>
double interpolate(T table [], double index)
{
    long integral = static_cast<long>(index);  // The truncated integral part
    double fractional = index - integral;    // The remaining fractional part

    // grab the two items in-between which the actual value lies
    T value1 = table[integral];
    T value2 = table[integral+1];

    // interpolate: integer part + (fractional part * difference between value2 and value1)
    double final = value1 + ((value2 - value1) * fractional);

    return final;
}
```

Table 2.9: Full C++ template function to interpolate a value from a table, given a fractional index.

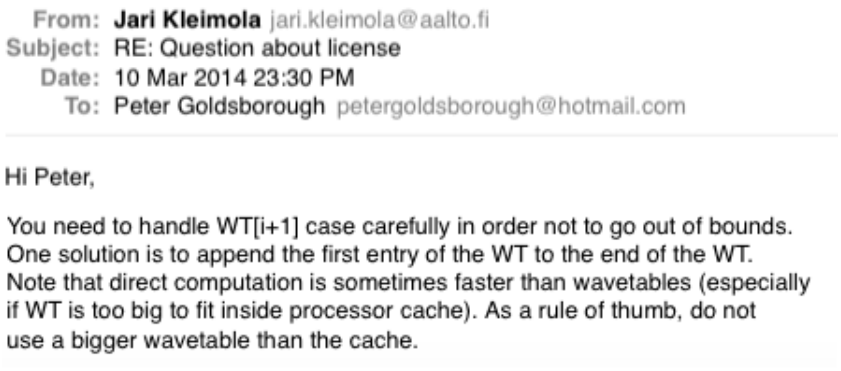
into the signal [...]". (Mitchell, 2008, p. 84) This is a suboptimal solution which would result in a change of phase and consequently distortion. The second option, interpolation, tries to approximate the true value from the sample at the current index and at the subsequent one. Interpolation is achieved by summation of the sample value at the floored, integral part of the table index, $\lfloor i \rfloor$, with the difference between this sample and the sample value at the next table index, $\lfloor i \rfloor + i$, multiplied by the fractional part of the table index, $i - \lfloor i \rfloor$. Table 2.8 displays the calculation of a sample by means of interpolation in pseudo-code and Table 2.9 in C++. (based on pseudo-code, Mitchell, 2008, p. 85)

2.4.3 Table Length

The length of the Wavetable must be chosen carefully and consider both memory efficiency and waveform resolution. An equation to calculate the size of a single wavetable in Kilobytes is given by Equation 2.18, where L is the table length and N the number of bytes provided by the resolution of the data type used for samples, e.g. 8 bytes for the double-precision floating-point data type `double`.

$$Size = \frac{L \cdot N}{1024} \tag{2.18}$$

Daniel R. Mitchell advises that the length of the table be a power of two for maximum efficiency. (Mitchell, 2008, p. 86) Moreover, during an E-Mail exchange with Jari Kleimola, author of the 2005 master's thesis "Design and Implementation of a Software Sound Synthesizer", it was discovered that "as a rule of thumb", the table length should not exceed the processor cache size. A relevant excerpt of this e-mail exchange is depicted in Figure 2.6. Considering both pieces of advice, it was decided that a Wavetable size of 4096 (2^{12}), which translates to 32 KB of memory, would be suitable.
One important fact to mention, also discussed by Jari Kleimola, is that because the interpolation algorithm from Table 2.9 must have access to the sample value at index $i+1$, i being the current index, an additional sample must be appended to the Wavetable to avoid a `BAD_ACCESS` error when i is at the last valid position in the table. This added sample has the same value as the



From: **Jari Kleimola** jari.kleimola@aalto.fi
Subject: RE: Question about license
Date: 10 Mar 2014 23:30 PM
To: Peter Goldsborough petergoldsborough@hotmail.com

Hi Peter,

You need to handle WT[i+1] case carefully in order not to go out of bounds. One solution is to append the first entry of the WT to the end of the WT. Note that direct computation is sometimes faster than wavetables (especially if WT is too big to fit inside processor cache). As a rule of thumb, do not use a bigger wavetable than the cache.

Figure 2.6: An excerpt of an E-Mail exchange with Jari Kleimola.

first sample in the table to avoid a discontinuity. Therefore, the period of the waveform actually only fills 4095 of the 4096 indices of the Wavetable, as the 4096th sample is equal to the first.

2.5 Noise

A noisy signal is a signal in which some or all samples take on random values. Generally, noise is considered as something to avoid, as it may lead to unwanted distortion of a signal. Nevertheless, noise can be used as an interesting effect when creating music. Its uses include, for example, the modeling of the sound of wind or the crashing of water waves against the shore of a beach. Some people enjoy the change in texture noise induces in a sound, others find noise relaxing and even listen to it while studying. (Mitchell, 2008, p. 76) Unlike all audio signals[1] presented so far, noise cannot[2] be stored in a Wavetable, as it must be random throughout its duration and not repeat periodically for a proper sensation of truly *random* noise. Another interesting fact about noise is that it is common to associate certain forms of noise with colors. The *color* of a noise signal describes, acoustically speaking, the *texture* or *timbre* [4] of the sound produced, as well as, scientifically speaking, the spectral power density and frequency content of the signal.

White noise is the most frequencly encountered form, or color, of noise. It is a random signal in its purest and most un-filtered form. In such a signal, all possible frequencies are found with a uniform probability distribution, meaning they are distributed at equal intensity throughout the signal. The association of noise with colors actually stems from the connection between white noise and white light, which is said to be composed of almost all color components at an approximately equal distribution. Figure 2.7 shows a typical white noise signal in the time domain, Figure 2.8 gives a close-up view of Figure 2.7, Figure 2.9 displays the frequency spectrum of a white noise signal and Table 2.10 shows the implementation of a simple C++ class to produce white noise.

[1]The term "waveform" would be incorrect here, as noise is not periodic and thus cannot really be seen as a waveform.

[2]Noise could theoretically be stored in a Wavetable, of course. However, even a very large Wavetable destroys the randomness property to some extent and would thus invalidate the idea behind noise being truly random.

[4]Timbre is a rather vague term used by musicians and audio engineers to describe the properties, such as pitch, tone and intensity, of an audio signal's sound that distinguish it from other sounds. The *Oxford Dictionary of English* defines timbre as "the character or quality of a musical sound or voice as distinct from its pitch and intensity".

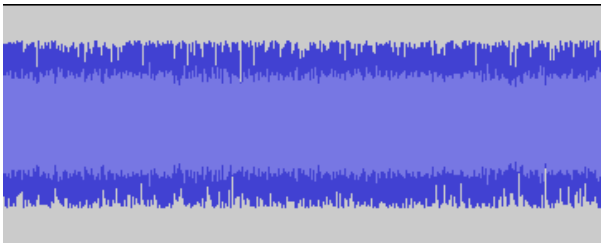


Figure 2.7: A typical white noise signal.

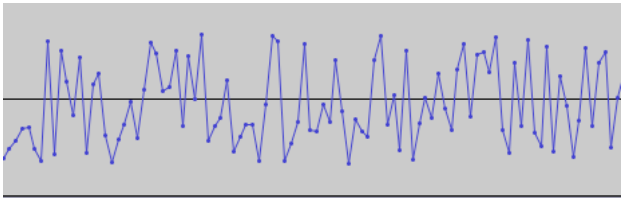


Figure 2.8: A close-up view of Figure 2.7. This Figure shows nicely how individual sample values are completely random and independent from each other.

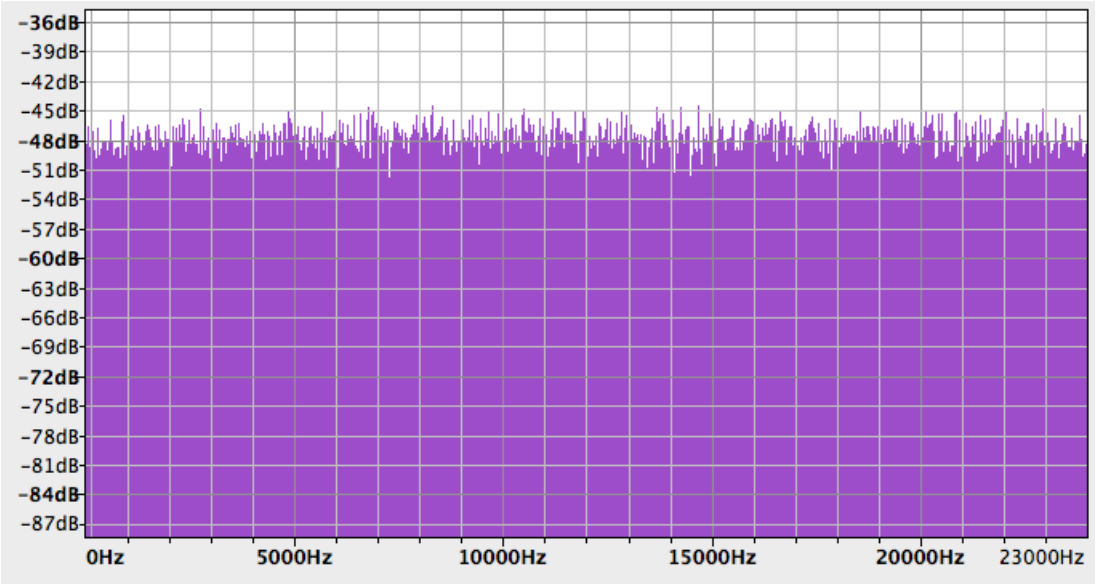


Figure 2.9: The signal from Figures 2.7 and 2.8 in the frequency domain. This frequency spectrum analysis proves the fact that white noise has a "flat" frequency spectrum, meaning that all frequencies are distributed uniformly and at (approximately) equal intensity.

```cpp
#include <random>
#include <ctime>

class Noise
{
    Noise()
    : dist_(-1,1)
    {
        // Seed random number generator
        rgen_.seed((unsigned)time(0));
    }

    double tick()
    {
        // Return noise sample
        return dist_(rgen_);
    }

private:

    /*! Mersenne-twister random number generator */
    std::mt19937 rgen_;

    /*! Random number distribution (uniform) */
    std::uniform_real_distribution<double> dist_;
};
```

Table 2.10: A simple C++ class to produce white noise. `rgen_` is a random number generator following the Mersenne-Twister algorithm, to retrieve uniformly distributed values from the `dist_` distribution in the range of -1 to 1. `tick()` returns a random white noise sample.

Chapter 3

Modulating Sound

One of the most interesting aspects of any synthesizer, digital as well as analog, is its capability to modify or *modulate* sound in a variety of ways. To modulate a sound means to change its amplitude, pitch, timbre, or any other property of a signal to produce an, often entirely, new sound. This chapter will examine and explain two of the most popular means of modulation in a digital synthesis system, Envelopes and Low Frequency Oscillators (LFOs).

3.1 Envelopes

When a pianist hits a key on a piano, the amplitude of the sound produced increases from zero, no sound, to some maximum amplitude which depends on a multitude of factors such as how hard the musician pressed the key, what material the piano string is made of, the influence of air friction and so on. After reaching the maximum loudness, the amplitude decreases until the piano string stops oscillating, resulting in renewed silence. To model such an evolution of amplitude over time, digital musicians use a modulation technique commonly referred to as an "Envelope".

3.1.1 Envelope segments

The following sections outline the creation of and terminology used for single segments of an envelope.

ADSR

A common concept associated with Envelopes is "ADSR", which stands for "Attack, Decay, Sustain, Release". These four terms name the four possible states an Envelope segment can take on. An Attack segment is any segment where the initial amplitude, at the start of a segment, is less than the final amplitude, at the end of the segment — the amplitude increases. Conversely, a Decay segment signifies a decrease in amplitude from a higher to a lower value. When the loudness of a signal stays constant for the full duration of an interval, this interval is termed a "Sustain" segment. While the three segment types just mentioned all describe the modulation of a signal's loudness when the key of a piano or synthesizer is still being pressed, the last segment type, a "Release" segment, refers to the change in loudness once the key is *released*. Figure 3.1 depicts an abstract representation of a typical ADSR envelope. Figure 3.2 shows a 440 Hz sine wave before the application of an ADSR envelope and Figure 3.3 displays the same signal after an ADSR envelope has been applied to it.

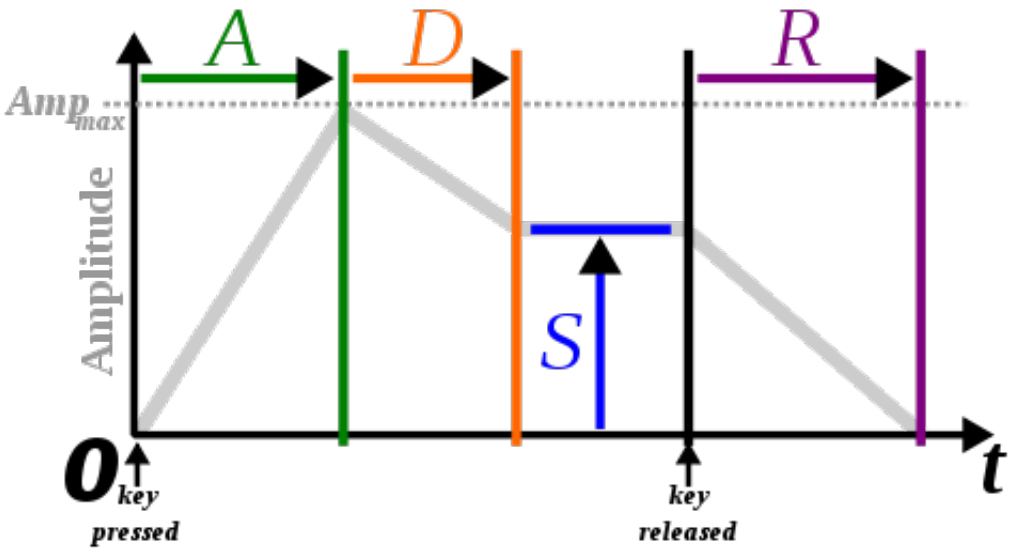


Figure 3.1: An Envelope with an Attack, a Decay, a Sustain and finally a Release segment. Source: `http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/ADSR_parameter.svg/500px-ADSR_parameter.svg.png`

Figure 3.2: A 440 Hz sine wave.

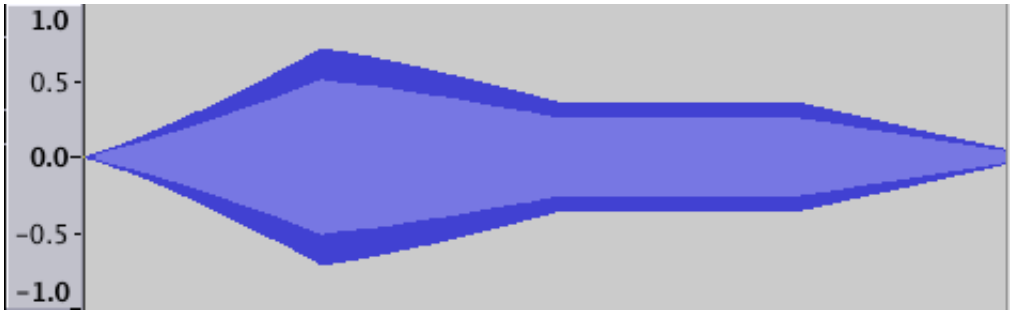


Figure 3.3: The same signal from Figure 3.2 with an ADSR Envelope overlayed on it.

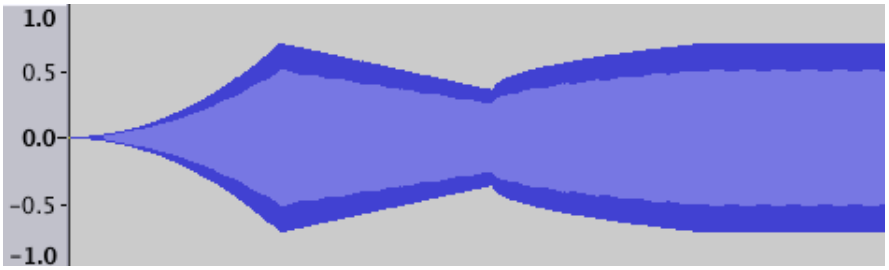


Figure 3.4: An envelope where r is equal to 2, then to 1, then to 0.5.

Mathematical Definition and Rate Types

Mathematically, an Envelope segment can be modeled using a simple power function of the general form presented in Equation 3.1, where a_{final} is the final amplitude at the end of the segment, a_{start} the initial amplitude at the beginning of the segment and r the parameter responsible for the shape or "rate" of the segment. When r, which must kept between 0 and ∞ (practically speaking some value around 10), is equal to 1, the segment has a linear rate and is thus a straight line connecting the initial amplitude with the final loudness. If r is greater than 1, the function becomes a power function and consequently exhibits a non-linear increase or decrease in amplitude. Lastly, if r is less than 1 but greater than 0, the function is termed a "radical function", since any term of the form $x^{\frac{a}{b}}$ can be re-written to the form $\sqrt[b]{x^a}$, where the numerator a becomes the power of the variable and the denominator b the radicand. Figure 3.4 displays an envelope whose first segment has $r = 2$, a quadratic increase, after which the sound decays linearly, before increasing again, this time r being equal to $\frac{1}{2}$ (a square root function). A C++ class for single Envelope segments is shown in Listing A.3.

$$a(t) = (a_{final} - a_{start}) \cdot t^r + a_{start} \tag{3.1}$$

3.1.2 Full Envelopes

Creating full Envelopes with a variable number of segments requires little more work than implementing a state-machine, which checks whether the current sample count is greater than the length of the Envelope segment currently active. If the sample count is still less than the length of the segment, one retrieves Envelope values from the current segment, else the Envelope progresses to the next segment. Additionally, it should be possible for the user to loop between segments of an Envelope a variable number of times before moving on to the release segment. Table 3.1 displays two member functions of an Envelope class created for this thesis, which return an Envelope value from the current Envelope segment and allow for the updating of the sample count.

Envelopes have many uses. Some require flexible segments which allow for the adjusting of individual segments' lengths, others need all segments to have a constant length. Some give the user the possibility to modulate individual segments by making them behave like a sine function, others do not. Fortunately, C++'s inheritance features make it very easy and efficient to construct such a variety of different classes that may or may not share relevant features. The inheritance diagram for the final `Envelope` class created for the purpose of this thesis reflects how all of these individual class can play together to yield the wanted features for a class. This inheritance diagram is displayed in Figure 3.5.

```
1  void EnvSegSeq::update()
2  {
3      currSample_++;
4      currSeg_->update();
5  }
6
7  double EnvSegSeq::tick()
8  {
9      if (currSample_ >= currSeg_->getLen())
10     {
11         // Increment segment
12         currSeg_++;
13
14         // Check if we need to reset the loop
15         if (currSeg_ == loopEnd_ && (loopInf_ || loopCount_ < loopMax_))
16         { resetLoop_(); }
17
18         // If we've reached the end, go back to last segment
19         // which will continue to tick its end amplitude value
20         else if (currSeg_ == segs_.end()) currSeg_--;
21
22         // else change
23         else changeSeg_(currSeg_);
24     }
25
26     return currSeg_->tick();
27 }
```

Table 3.1: Two member functions of the EnvSegSeq class (Envelope Segment Sequence).

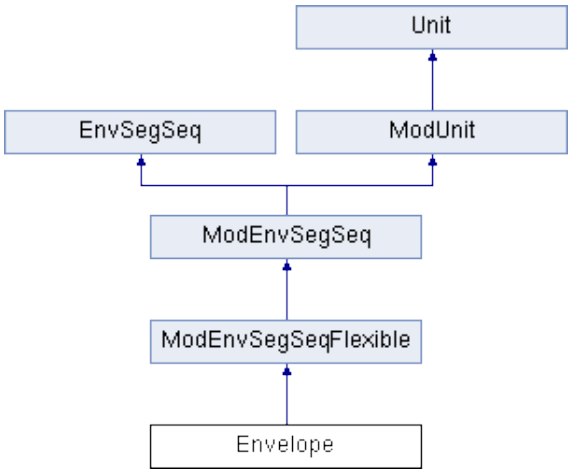


Figure 3.5: The inheritance diagram for the `Envelope` class. The `Unit` and `ModUnit` classes are two abstract classes that will be explained in later parts of this thesis.

3.2 Low Frequency Oscillators

A Low Frequency Oscillator, abbreviated "LFO", is an oscillator operating at very low frequencies, typically in the range below human hearing (0-20 Hz), used solely for the purposes of modulating other signals. The most common parameter to be modulated by an LFO in a synthesizer is the amplitude of another oscillator, to produce effects such as the well known "vibrato" effect. In this case, were the frequency of the LFO to be in the audible range, one would use the term Amplitude Modulation (AM), which is also a method for synthesizing sound. Equation 3.2 shows how an LFO can be used to change the amplitude of another oscillator[1]. Figure 3.6 shows a 440 Hz sine wave, Figure 3.7 displays the same sine wave now modulated by a 2 Hz LFO and in Figure 3.8 the frequency of the LFO has been increased to 20 Hz to produce a vibrato effect. It should be noted that an LFO can also modulate any other parameter, such as the rate of an Envelope segment.

$$O(t) = (A_{osc} + (A_{lfo} \cdot sin(\omega_{lfo}t + \phi_{lfo}))) \cdot sin(\omega_{osc}t + \phi_{osc}) \tag{3.2}$$

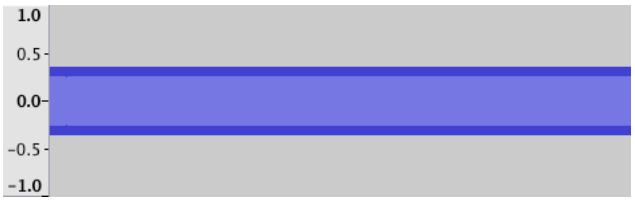


Figure 3.6: A 440 Hz sine wave.

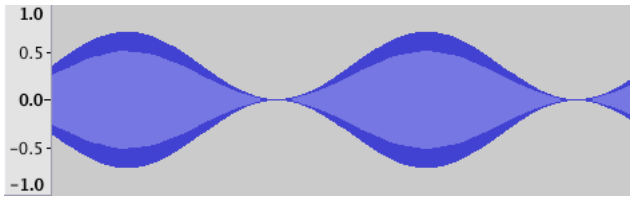


Figure 3.7: The sine wave from Figure 3.6 modulated by an LFO with a frequency of 2 Hertz. Note how the maximum amplitude of the underlying signal now follows the waveform of a sine wave. Because the LFO's amplitude is the same as that of the oscillator (the "carrier" wave), the loudness is twice as high at its maximum, and zero at its minimum.

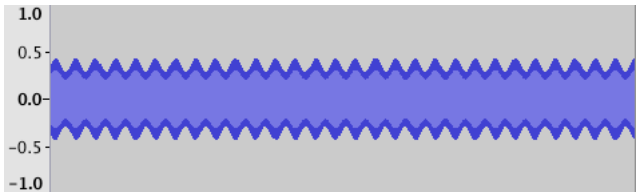


Figure 3.8: The sine wave from Figure 3.6 modulated by an LFO with a frequency of 20 Hertz. This signal is said to have a "vibrato" sound to it.

[1]This is the same equation as for AM.

Concerning the implementation of an LFO in a computer program, the process could be as simple as re-naming a class used for an oscillator:

```
1 | typedef OscillatorClass LFOClass.
```

In the synthesizer created for this thesis, called *Anthem*, the distinction between an LFO and an oscillator is the possibility to modulate an LFO's parameters, for example using an Envelope or another LFO, whereas the `Oscillator` class is an abstract base class whose sole purpose is to be an interface to a Wavetable. This property of the `Oscillator` class is used by both the `LFO` and the `Operator` class, who both inherit from the `Oscillator` class. The `Operator` class is the sound generation unit the user actually interfaces with from the Graphical User Interface (GUI) of *Anthem*. It is derived from the `Oscillator` class because it ultimately needs to generate sound samples, but it is its own class because it has various other features used for sound synthesis. These features are discussed in a later chapter. The relationships just described lead to the inheritance diagram shown in Figure 3.9.

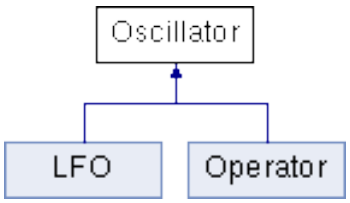


Figure 3.9: Inheritance diagram showing the relationship between an `LFO`, an `Operator` and their base class, the `Oscillator` class.

3.2.1 LFO Sequences

A common sight in many modern digital synthesizers, such as Native Instruments'[2] "Massive" synthesizer, is a form of step-sequencer for Low Frequency Oscillators. This modulation unit is essentially a synthesis between the concept of an Envelope and that of an LFO. It has a variable number of segments of fixed length, each with their own LFO to change the waveform of the segment. Figure 3.10 displays what Native Instruments calls a "Stepper".
For the purpose of this thesis, such an LFO sequence was created. Regarding the implementation in C++, there is little difference between an LFO Sequence and an Envelope Segment Sequence, except for the fact that now each individual segment has an LFO that can be accessed through the interface of the class to change the parameters or the waveform of each LFO. One of the two parameters that deserves to be discussed, however, is the frequency or "rate" of such an LFO sequence. Just like the frequency of an oscillator describes the number of cycles its waveform completes every second, the rate of an LFO Sequence determines how often the entire sequence is traversed in one second. Practically speaking, this involves adjusting the length of individual segments in such a way that the combined length equals the wanted duration. To give an example: if the user wishes to change the rate of an LFO Sequence with 10 segments to 1 Hz, each individual segment must have a duration of 0.1 seconds (100 milliseconds), so that at the end of the sequence, exactly 10 segments $\cdot\, 0.1$ seconds $= 1$ second will have passed. The length of a single segment can be derived from Equation 3.3, where r is the rate of the LFO Sequence and N the number of segments in the sequence.

[2] "Native Instruments is a technology company that develops software and hardware for music production and DJ-ing", based out of Berlin, Germany. Source: *Wikpedia:* `http://en.wikipedia.org/wiki/Native_Instruments`. Homepage: `http://www.native-instruments.com/en/`

Figure 3.10: An LFO sequence, here named a "Stepper". Source: `http://www.askaudiomag.com/articles/massives-performer-and-stepper-lfos`

$$l_{seg} = \frac{1}{r \cdot N} \tag{3.3}$$

The second parameter worthy of mentioning is the frequency of individual segments' modulation LFOs. As mentioned, the frequency of an oscillator or of an LFO sets the cycles per second of their waveform. However, when the length of a single segment in an LFO sequence is only very small, e.g. 100 milliseconds, having an LFO modulate the segment at a common frequency of, say, 1 Hz, would mean that only one tenth of the LFO's period is completed. To have the LFO complete a full cycle, the user would have to adjust the frequency of the LFO in such a way that it can complete one period within the duration of the segment (in this case, 10 Hz). This may still seem like a manageable task for the user. However, once parameter values become a little more complicated, giving birth to cases where the user has to figure out that to have 5 cycles in a segment with a duration of 0.4 seconds, the frequency needs to be exactly 12.5 Hz, the objective reader must admit that this is not intuitive behaviour. Therefore, it was decided that for modulation LFOs in LFO sequences, the rate should not be defined as cycles per second, but as cycles per *segment*. This means that setting the rate of a segment's LFO to 5 Hertz results in 5 cycles per second regardless of the segment's duration. Naturally, the task of calculating the correct frequency for the LFO still has to be performed by the computer program. The real frequency can be calculated using Equation 3.4 when the duration d is in seconds and Equation 3.5 when the duration is in samples, as is the case for a digital synthesizer. In both Equations $f_{displayed}$ stands for the frequency the user specifies, the cycles per segment, and f_{real} for the actual frequency in cycles per second. In Equation 3.5, f_s denotes the sample rate. Table 3.2 shows how the calculation of the two parameters just dicussed is implemented in the `LFOSeq` class.

$$f_{real} = \frac{f_{displayed}}{d} \tag{3.4}$$

$$f_{real} = \frac{f_s}{d} \cdot f_{displayed} \tag{3.5}$$

```
void LFOSeq::setRate(double Hz)
{
    if (Hz < 0 || Hz > 10)
    { throw std::invalid_argument("Rate cannot be less than zero or greater 10!"); }

    rate_ = Hz;

    resizeSegsFromRate_(rate_);
}

void LFOSeq::resizeSegsFromRate_(double rate)
{
    // get the period, divide up into _segNum pieces
    segLen_ = (Global::samplerate / rate) / segs_.size();

    // Set all segments' lengths
    for (int i = 0; i < segs_.size(); i++)
    {
        segs_[i].setLen(segLen_);

        // Scale frequency of mods according to length
        setScaledModFreq_(i);
    }
}

void LFOSeq::setScaledModFreq_(seg_t seg)
{
    // Set scaled frequency to frequency of lfo
    lfos_[seg].lfo.setFrequency(getScaledModFreqValue(lfos_[seg].freq));
}

double LFOSeq::getScaledModFreqValue(double freq) const
{
    // Since the rate is in cycles per segment
    // and not cycles per second, we get the
    // "period" of the segment and multiply that
    // by the rate, giving the mod wave's frequency.

    if (! segLen_) return 0;

    // To go from samples to Hertz, simply
    // divide the samplerate by the length
    // in samples e.g. 44100 / 22050 = 2 Hz
    double temp = Global::samplerate / static_cast<double>(segLen_);

    // Multiply by wanted frequency
    return freq * temp;
}
```

Table 3.2: Relevant member functions from the `LFOSeq` class for calculating the correct rate for individual segments as well as for the entire sequence.

3.3 The ModDock system

The majority of digital synthesizers, such as Propellerhead's[3] *Thor* or Native Instruments'[2] *FM8* synthesizer, implement modulation in a static way. Instead of making it possible to use an LFO or an Envelope to modulate any parameter in a synthesis system, units, e.g. an oscillator, have dedicated LFOs and Envelopes, which modulate only one parameter — mostly amplitude — and only for this unit. On the other hand, some synthesizers like *Massive*, also created by Native Instruments, implement a system where LFOs and Envelopes can be used by a variable number of units, for a variable number of parameters. For this thesis and the synthesizer created for it, such a system, here called the "ModDock" system, was emulated. The following sections will outline the process of defining and implementing the ModDock system.

3.3.1 Problem statement

In short, the ModDock system should make it possible to modulate a variable number of parameters of a variable number of units of the synthesizer, with any of a fixed number of LFOs, Envelopes or similar *Modulation Units*. Consequently, each unit in the synthesis system should have what will be known as a "ModDock", a set of "docks", the number of which depends on the number of parameters the unit makes it possible to modulate, where the user may insert a Modulation Unit. Due to the fact that many units may be modulated by one Modulation Unit, the Modulation Unit must not update its value until all units have been modulated by it. For example, the sample count of an Envelope must stay the same until all dependent units have retrieved the current Envelope value from it. Moreover, a unit should be able to adjust the depth of modulation by a Modulation Unit, i.e. it should be possible to have only 50% of an LFO's signal affect the parameter it is modulating. Finally, there should be a way of *side-chaining* Modulation Units so that one Modulation Unit in a dock modulates the depth of another Modulation Unit in that same dock, the signal of which may again side-chain the depth of another Modulation Unit in that ModDock. The final modulation of a parameter will be the average of all modulation values of a ModDock affecting that parameter.

3.3.2 Implementation

In order to let units of the synthesizer created for this thesis share certain behaviour and member functions, as well as to distinguish between the traits and tasks certain units have that others do not, it was necessary to develop an inheritance structure that would satisfy these requirements. In *Anthem*, a *Unit* is defined as any object with modulateable parameters. A Unit has necessary member variables and functions to permit its parameters to be modulated by other *Modulation Units*. A Modulation Unit, short *ModUnit*, shall be defined as any Unit that can modulate a parameter of a Unit. The ModUnit class includes the *pure virtual*[4] `modulate` method, which takes a sample as one of its arguments and then returns a modulated version of that sample. The form of modulation depends entirely on the implementation of the `modulate` method by the class that inherits from the ModUnit class, as the ModUnit class itself does not implement any standard way of modulating a sample. Figure 3.11 displays the inheritance diagram for the Unit class.

[3] "Propellerhead Software is a music software company, based in Stockholm, Sweden, and founded in 1994." Source: *Wikipedia* `http://en.wikipedia.org/wiki/Propellerhead_Software`. Homepage: `https://www.propellerheads.se`

[4] In C++, a pure virtual function is a function that does not have any standard implementation and thus requires the derived classes of the class that declares the pure virtual function to implement that function on their own. A

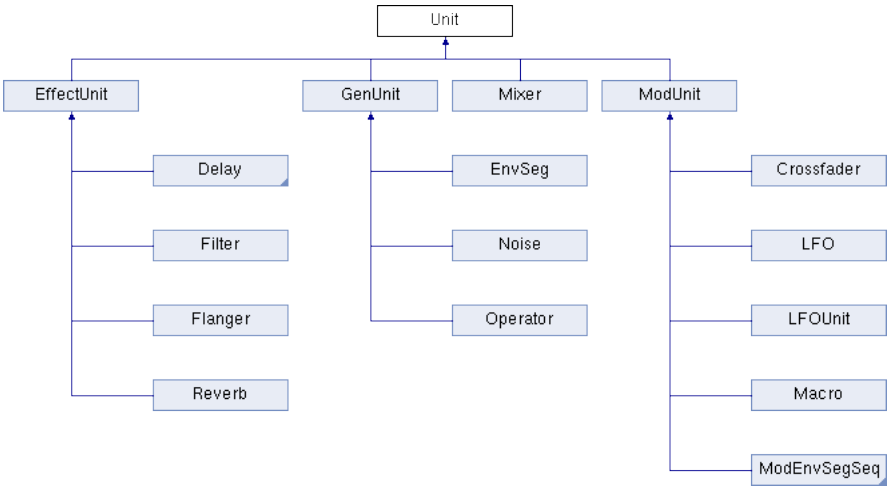


Figure 3.11: The inheritance diagram of the Unit class. GenUnits are Units that generate samples. EffectUnits apply some effect to a sample, e.g. a sample delay.

The `modulate` method

All classes derived from the ModUnit class must implement the `modulate` method, which takes the sample to modulate, a depth value between 0 (no modulation) and 1 (full modulation) as well as a maximum boundary as its arguments. In the case of LFOs, the maximum boundary parameter determines the value that is added to the sample. For example, when the amplitude of a unit is modulated, the maximum boundary is 1, meaning that the value added to the sample is in the range of $[-A_{LFO} \cdot 1; A_{LFO} \cdot 1]$, whereas for the rate parameter of Envelope segments, where the maximum boundary is 10, the range is $[-A_{LFO} \cdot 10; A_{LFO} \cdot 10]$. The declaration of the `modulate` method in the ModUnit class is given below.

```
1 |   virtual double modulate(double sample, double depth, double maximum) = 0;
```

This method was the main motivation behind the creation of the ModUnit class and is especially important for ModDocks, as it makes it possible to polymorphically access the `modulate` method of any ModUnit through a pointer-to-ModUnit (`ModUnit*`). This means that each ModUnit can have its own definition of what it means to modulate a sample. Table 3.3 shows the definition of the `modulate` method in the LFO and Table 3.4 in the Envelope class.

```
1 | double LFO::modulate(double sample, double depth, double maximum)
2 | {
3 |   return sample + (maximum * Oscillator::tick() * depth * amp_);
4 | }
```

Table 3.3: The implementation of the `modulate` method for LFOs.

class that declares a pure virtual method is termed an *abstract* class and may not be instantiated.

```
double Envelope::modulate(double sample, double depth, double)
{
   return sample * tick_() * depth * amp_;
}
```

Table 3.4: The implementation of the `modulate` method for Envelopes. Note that for the Envelope class, the paramter `maximum` is not relevant, which is why it is never used. This shows that all that matters is that the method returns a modulated sample — what "modulate" means is up to the class that implements it.

ModDocks

A ModDock is simply a collection of ModUnits. The ModDock collects all the modulated samples of individual ModUnits in the dock and returns an average over all samples. For example, if one LFO adds a value of 0.3 to the amplitude parameter of a Unit with a current amplitude of 0.5 and another subtracts a value of 0.1, the final amplitude of that Unit will be 0.6, as $\frac{(0.5+0.3)+(0.5-0.1)}{2} = 0.6$. Moreover, this means that if the two LFOs were to add/subtract the same absolute value but with a different sign, for example -0.4 and 0.4, the net difference would be 0 and the amplitude would remain 0.5. Something else the ModDock takes care of is boundary checking and value adjustment. Meaning that, continuing the example given above, were an LFO to add a value of ± 1 to the base amplitude of 0.5, the amplitude would not oscillate in the range $[-1.5; 1.5]$. Rather, the ModDock ensures that the value trespasses neither the maximum boundary nor the minimum boundary, which is another parameter supplied to the ModDock. Therefore, the amplitude value would remain in the optimal range of $[-1; 1]$. Alongside the maximum and the minimum boundary, the Unit who owns the ModDock can pass the current base value of the parameter to be modulated to the ModDock. In the aforementioned example, the base value would be 0.5. Also, the ModDock can store the depth of modulation of individual ModUnits. Because both the `depth` and the `maximum` parameter of the `modulate` method for ModUnits are now stored in an instance of the `ModDock` class, the `modulate` method has a much simpler declaration in the `ModDock` class:

```
  double modulate(double sample);
```

What this simplification of the `modulate` method requires, however, is that the maximum and minimum boundary as well as an initial base value are passed to the relevant ModDock in the construction of the Unit that owns it. Additionally, the ModDock must be notified whenever the base value changes. Table 3.5 shows how these parameters may be passed to a ModDock and updated when necessary.

```
AnyUnit::AnyUnit()
{
    modDock.setHigherBoundary(1);
    modDock.setLowerBoundary(0);
    modDock.setBaseValue(0.5);
}

void AnyUnit::setAmp(double amp)
{
    modDock.setBaseValue(amp);
}
```

Table 3.5

Sidechaining

One of the most interesting and equally complicated tasks encountered when creating the ModDock system was the implementation of side-chaining. Side-chaining makes it possible to have one ModUnit in a ModDock modulate the depth parameter of another ModUnit in that same ModDock. Borrowing from the terminology of digital communication, a ModUnit that side-chains another ModUnit is termed a "Master" and the ModUnit being side-chained is called a "Slave". Moreover, it was decided that any ModUnit that is not a Master shall be called a non-Master. Therefore, a non-Master is either a Slave or not involved in any side-chaining relationship at all — a normal ModUnit. It should be noted that the signal of a Master does not affect the final modulation value of a ModDock directly, i.e. the modulation value of a Master is not taken into consideration during the averaging process described above, but only indirectly, by modulating the depth of a Slave. Therefore, a ModUnit can be either a Master and contribute to the final modulation indirectly or be a non-Master and contribute to the final value directly. It should be noted that it is entirely possible to have one Master modulate multiple Slaves and for one Slave to have multiple Masters. Figure 3.12 depicts these relationships in a flow-chart. Figure 3.13 shows a scan of early sketches created while implementing side-chaining.

What Figure 3.13 also displays is that there are two possible ways to implement side-chaining. The first method, which was ultimately not chosen, is to give each ModUnit in the ModDock its own ModDock where Masters could be inserted. The second method involves internally linking Masters and Slaves within the ModDock. This second implementation was finally decided to be the better one as it does not require the instantiation of a full new ModDock for each ModUnit, while providing the same functionality. Listing A.4 shows all private members of the `ModDock` class. Special attention should be payed to the `ModItem` struct, which is essential to the implementation of side-chaining and is very similar to the concept of a Linked-List. Each `ModItem` stores the aforementioned pointer-to-ModUnit to access the `modulate` method of the ModUnit. Moreover, there is one vector of indices for all Masters of that particular ModItem and one vector of indices for all of its Slaves. These indices refer to the positions of Masters/Slaves in the vector where all ModItems are stored, the `modItems_` vector. When the user sets up a side-chain relationship between two ModItems of a ModDock, the index of the Slave is added to the Slave vector of the Master and the index of the Master is inserted into the Master vector of the Slave. Should the user wish to "un-side-chain" two ModItems, their indices are removed from the each other's appropriate vector. Furthermore, the `ModItem` struct holds a baseDepth variable. This variable is similar to the baseValue of the ModDock, in the sense that is the ModItem's original depth which serves as the base value for modulation by other ModItems. The modulation of a Slave by its Masters is implemented in the exact same way that a parameter of a Unit is modulated by a ModDock's ModUnits. Modulated Slave samples are summed and averaged over their number. Listing A.5 gives the full definition of the `modulate` method of the ModDock class. In lines 9 to 35, Slaves are modulated by their Masters. Subsequently, in lines 39 to 62, the sample passed to the function from the Unit who owns the ModDock is modulated by all non-Masters and then finally returned.

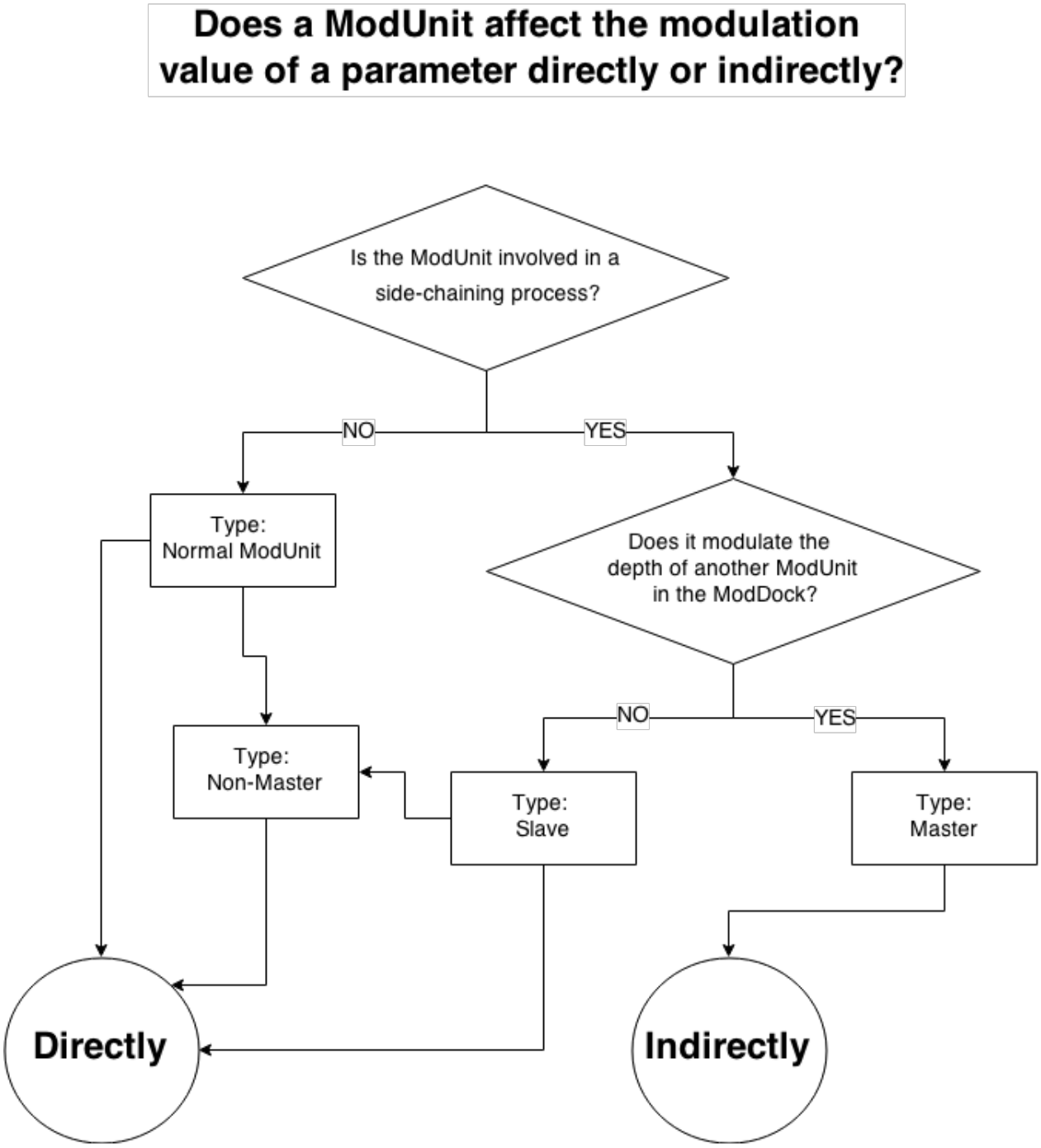

Does a ModUnit affect the modulation value of a parameter directly or indirectly?
Is the ModUnit involved in a side-chaining process?
NO
YES
Type: Normal ModUnit
Does it modulate the depth of another ModUnit in the ModDock?
NO
YES
Type: Non-Master
Type: Slave
Type: Master
Directly
Indirectly

Figure 3.12

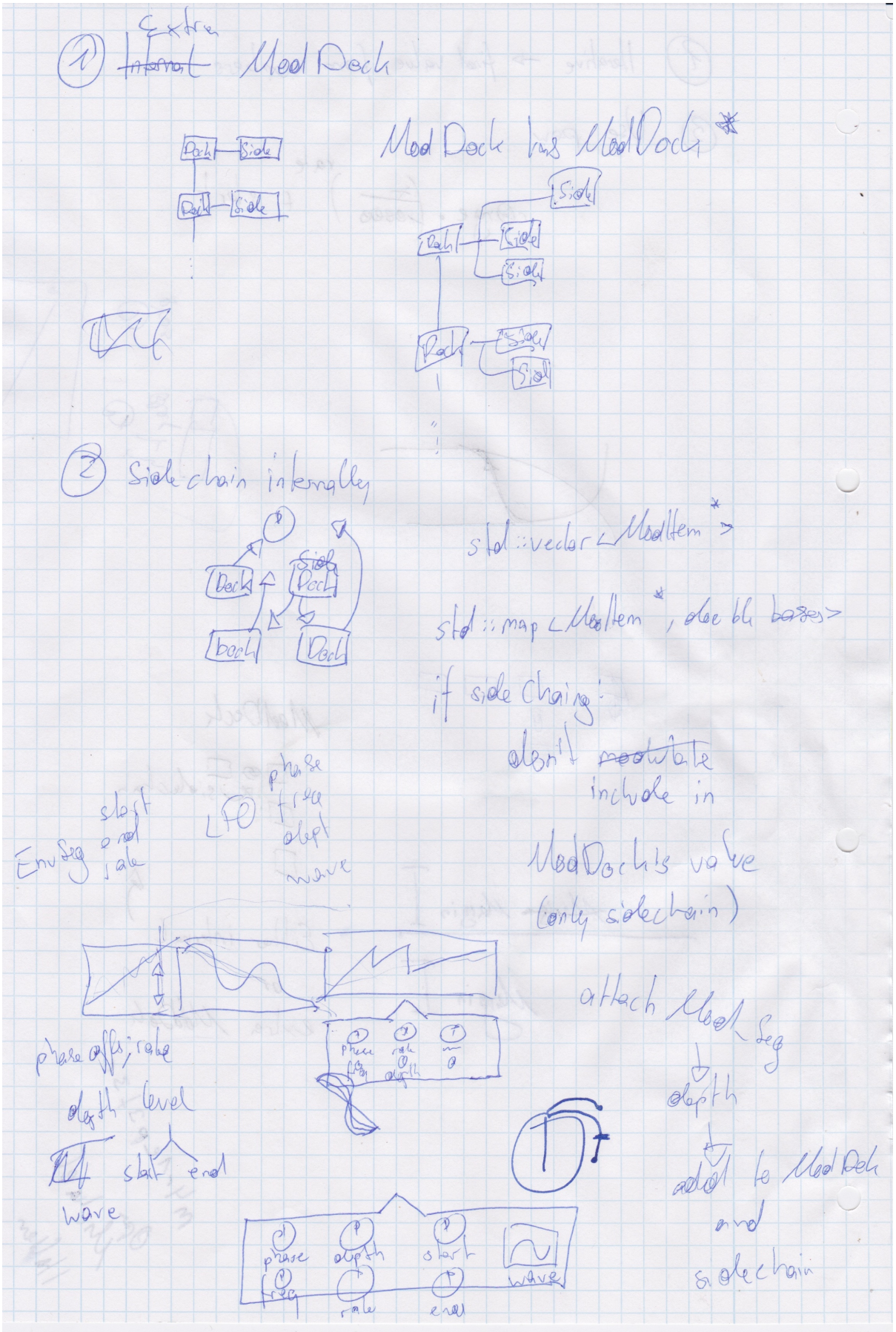

1
Extra
ModDock
ModDock has ModDock*
Dock
Side
2
Side chain internally
std::vector<ModItem*>
std::map<ModItem*, double bl bases>
if side chain:
don't modulate
include in
ModDock's value
(only sidechain)
attach ModLfo
depth
add to ModDock
and
sidechain
EnvSeg
start
end
rate
LFO
phase
freq
depth
wave
phase offs; rate
depth level
start end
wave
phase
depth
start
freq
rate
end
wave

Figure 3.13

Chapter 4

Filtering Sound

A feature any digital synthesizer should include is the possibility to filter sound. In digital signal processing, filtering a signal means to "alter [its] frequency spectrum [...] by amplifying or attenuating selected frequencies". (Mitchell, 2008, p. 102)

The discussion of digital filters is closely linked to the concept of a signal's phase. More concretely, it must be examined how the phase relationship between two *input* signals influences the properties of the *output* signal produced when the two input signals interfere either constructively or destructively. A simple examples of phase relationships from the analog realm is the situation where signals A and B are exactly identical and consequently have a phase difference $\Delta\phi$ of 0. If these signals are summed, the resulting signal C will have an amplitude twice as high or low as that of signal A or B in every single point of its signal — the input signals interfere *constructively*. Conversely, if signals A and B have a phase difference of 180°or π radians, the two signals interefere *destructively* and result in signal C having a constant amplitude of exactly 0, as the amplitude values of signals A and B cancel each other out in each point of the signal.

In digital systems, where signals are represented discretely, i.e. with periodically recorded samples, a phase difference can be modeled by a *sample delay*. (Mitchell, 2008, p. 103) For example, if a signal that is periodically represented by 10 samples is summed with a version of itself that is delayed by 5 samples, that version will effectively be phase-shifted by 180°and thus the summation of these two signals will result in silence. Had only one sample been delayed instead of five, the phase difference $\Delta\phi$ would have only been 36°or $\frac{2\pi}{10}$ radians — one tenth of a period. The amount of phase difference $\Delta\phi$ depends on both the number of samples a signal is represented by, so the sample rate of the system, and the number of samples that are delayed. An equation to calculate the phase difference in radians, given these parameters, is shown in Equation 4.1, where f is the frequency of the signal, f_s the sample rate and N the number of samples that are delayed. If 2π is replaced by 360, this Equation yields the same phase difference in degrees.

$$\Delta\phi = \frac{N \cdot f}{f_s} \cdot 2\pi \tag{4.1}$$

From Equation 4.1 it can be deduced that a sample delay causes different phase relationships for different frequencies of the original signal. To give an example: let one sample be delayed in a system with a sample rate f_s equal to 20 kHz. In such a system, a signal with a frequency of 1 Hz is represented by 20000 samples. Consequently, summing such a signal with a version

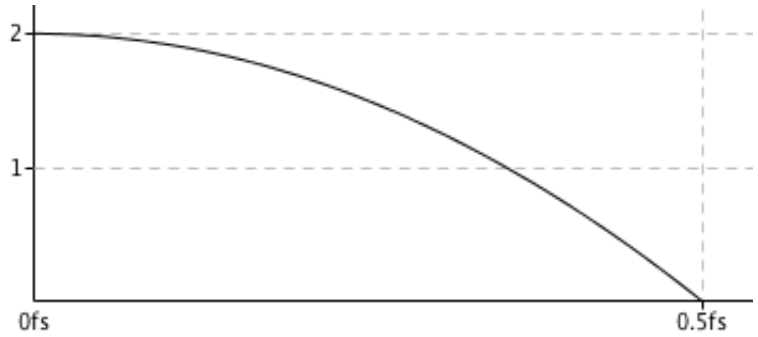


Figure 4.1: The frequency response of a one-sample-delay filter. The most constructive interference is at 0 Hz, a signal that is constant with $f(t) = 1$. Such a signal with a frequency of 0 Hz is also called DC, for direct current, because in analog circuits a DC current is constant. The most destructive interference occurs at $\frac{f_s}{2}$ Hz.

of itself delayed by one sample would make the resulting, filtered signal almost twice as loud, as $\Delta\phi$ is only about 0.018°. However, a signal with the maximum possible frequency of 10 Khz (the Nyquist limit) is represented by only 2 samples. Therefore, delaying one sample equals a phase shift of 180°, which results in silence when the signal is summed with the delayed version of itself. This introduces the concept of a filter's *frequency response*, which describes how a filter influences a signal's frequency spectrum. For the filter just described, the frequency response would look approximately like Figure 4.1.

One way to change the frequency response of a filter is to weight samples of the input signal as well as the delayed samples with *filter coefficients* (Mitchell, 2008, p. 105, 110). Whereas previously a one-sample delay could be modeled by Equation 4.2 (Mitchell, 2008, p. 104), where y_n is the output, x_n the current and x_{n-1} the delayed sample, Equation 4.3, where a and b are the respective filter coefficients, must be used when samples are weighted with coefficients. (Mitchell, 2008, p. 103) Visually, such a one-sample delay can be represented with the diagram shown in Figure 4.2.

$$y_n = x_n + x_{n-1} \tag{4.2}$$

$$y_n = bx_n + ax_{n-1} \tag{4.3}$$

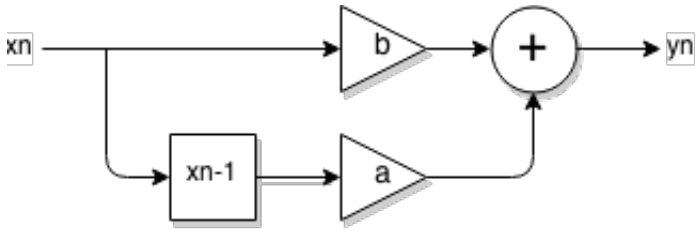


Figure 4.2: A one-sample delay with filter coefficients a and b. Triangles are amplifiers/attenuators. Samples are summed at the circle with the + at its center.

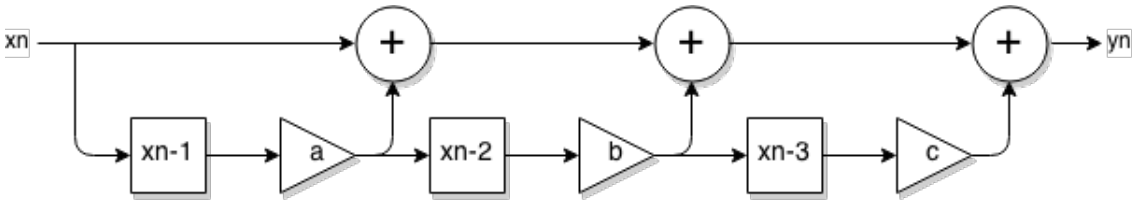


Figure 4.3: A three-sample delay filter.

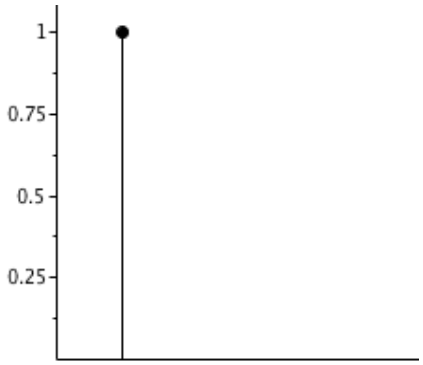


Figure 4.4: A delta function, also called a unit impulse. The first sample has an amplitude of 1 while all other samples are equal to 0.

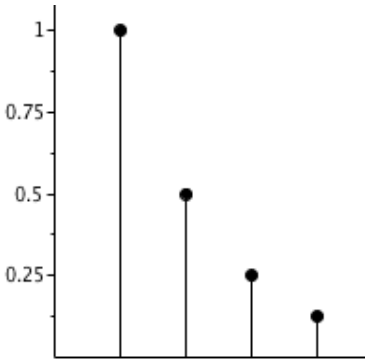


Figure 4.5: The impulse response of the three-sample delay filter depicted in Figure 4.3.

4.1 FIR and IIR Filters

While a filter's frequency response describes how the frequency content of a signal is affected by that filter, its influence on a signal in the time domain is given by the filter's *impulse response* or *kernel*. More precisely, an impulse response determines what a filter outputs when its input is a delta function, δ, which has a normalized *impulse*, meaning a sample with an amplitude of 1, as its first sample while all other subsequent samples have an amplitude of 0. The δ function is also referred to as the *unit impulse*. (Smith, 1999, p. 108) For example, if the filter coefficients of the three-sample delay filter shown in Figure 4.3 are all equal to 0.5 ($a = b = c = 0.5$), leading to Equation 4.4, then Figure 4.5 gives the impulse response of this filter when it is fed the δ function depicted in Figure 4.4. What this impulse response shows is that if a sample x_n with an amplitude of 1 is input into this filter, the first sample output will be $1 + \frac{0}{2} + \frac{0}{2} + \frac{0}{2} = 1$. The second sample to exit this filter will then be $0 + \frac{1}{2} + \frac{0}{2} + \frac{0}{2} = 0.5$, where, following the definition of the δ function given above, the new x_n is equal to 0, while the previous x_n that was equal to 1 has moved into the position of x_{n-1}. The third sample will be equal to $0 + \frac{0}{2} + \frac{0.5}{2} + \frac{0}{2} = 0.25$ and finally the fourth sample has a value of $\frac{1}{8}$.

$$y_n = x_n + \frac{x_{n-1}}{2} + \frac{x_{n-2}}{2} + \frac{x_{n-3}}{2} \tag{4.4}$$

The operation that was just performed on the δ function is called "convolution". Convolution has its own mathematical symbol: $*$. To convolve a signal means to apply an impulse response to every of its samples. Table 4.1 shows a C++ program to perform convolution (based on pseudo-code, see Mitchell, 2008, p. 107).

```cpp
#include <vector>

std::vector<double> convolve(const std::vector<double>& signal,
                             const std::vector<double>& impulseResponse)
{
    std::vector<double> output(signal.size(),0);

    for(std::vector<double>::size_type i = 0, endI = signal.size();
        i < endI;
        ++i)
    {
        for(std::vector<double>::size_type j = 0, endJ = impulseResponse.size();
            j < endJ;
            ++j)
        {
            output[i + j] += signal[i] * impulseResponse[j];
        }
    }

    return output;
}
```

Table 4.1: A C++ function to convolve a signal vector with an impulse response vector. It should be noted that to convolve a signal vector of size m with an impulse response vector of size n, the output vector must have a minimum size of $m + (n - 1)$.

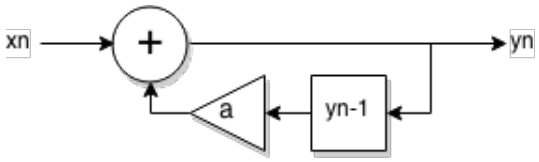


Figure 4.6: An IIR delay filter.

The filters described so far are called Finite Impulse Response (FIR) filters, because their impulse response is of finite size, like the one shown in Figure 4.5, caused by the Filter depicted in Figure 4.3. Given that in this three-sample delay filter all filter coefficients have the same value of 0.5, the diagram could actually be re-arranged and simplified if the three delays are replaced by a single *recursive* delay. Instead of processing an input sample, a recursive delay is fed an output sample, which it then amplifies or attenuates and feeds back into the signal. This re-arranged filter, also called a *feedback loop*, is shown in Figure 4.6. Because this feedback loop could theoretically go on forever, such a filter is called an Infinite Impulse Response (IIR) filter. Equation 4.5 gives a mathematical definition for the filter shown in Figure 4.6. To prove that this filter results in the same response as the FIR filter from before, it can be fed a δ function. Because y_{n-1} is initially 0, the first output y_n will simply be $1 + \frac{0}{2}$. For the second sample, y_{n-1} takes on the value of the previous output, which was 1. Therefore, the second output sample is $0 + \frac{1}{2} = 0.5$, the next sample $0 + \frac{0.5}{2}$ and so on. Because this is an IIR filter, this process could theoretically go on ad infinitum. x_n is equal to 0 for all samples other than the first because of the way the δ function is defined.

$$y_n = x_n + \frac{y_{n-1}}{2} \tag{4.5}$$

4.2 Bi-Quad Filters

A common arrangement of FIR and IIR filters is a so-called bi-quad filter. A "bi-quad filter combines one and two sample feedforward delays (FIR) with one and two sample feedback delays (IIR)". (Mitchell, 2008, p. 109) Figure 4.7 shows such a bi-quad filter. The Equation to calculate the sample output of a bi-quad filter is given by Equation 4.6.

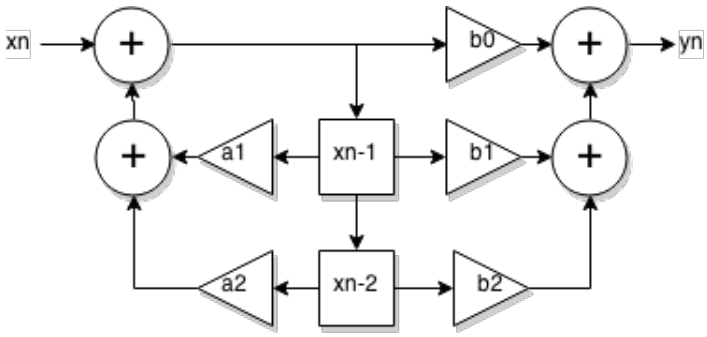


Figure 4.7: A bi-quad filter.

$$y_n = b_0(x_n - a_1 y_{n-1} - a_2 y_{n-2}) + b_1 x_{n-1} + b_2 x_{n-2} \tag{4.6}$$

4.2.1 Implementation

In the synthesizer created for this thesis, the `Filter` class implements a bi-quad filter. Table 4.2 shows the `Filter` class' `process` method, which takes an input sample and returns a filtered output sample.

```
1  double Filter::process(double sample)
2  {
3      double temp = sample
4      - (coefA1_ * delayA_)
5      - (coefA2_ * delayB_);
6
7      double output = (coefB0_ * temp)
8      + (coefB1_ * delayA_)
9      + (coefB2_ * delayB_);
10
11
12     // Store values into delay line
13     delayB_ = delayA_;
14     delayA_ = temp;
15
16     return output;
17 }
```

Table 4.2: C++ function to filter an input sample and return an output sample. This function implements the bi-quad filter equation given by Equation 4.6.

4.3 Filter Types

Bi-quad filters can be used to create a variety of different filter types and corresponding frequency responses, simply by adjusting the filter's coefficients. When examining a frequency response, the term **passband** is used for the frequency range that should ideally not be affected by the filter. Conversely, the **stopband** are those frequency components the filter should, ideally, silence completely. The frequency at which the transition from minimum to maximum attenuation occurs is called the **cutoff frequency** and is denoted by f_c. Because no filter is perfect, this transition usually does not happen precisely at the specified cutoff frequency. Rather, there is a certain **transition band**, which is the range of frequencies where the transition takes place. Within this transition band, the cutoff frequency is defined as the frequency component where the amplitude reaches $\frac{\sqrt{2}}{2}$ (0.707...) or -3dB in power[1]. If the transition band is narrow, the filter is said to have a fast **roll-off**. If it is rather wide, the roll-off is said to be slow. Figure 4.8 depicts a typical frequency response and labels the relevant bands and frequencies. It should be noted that bi-quad filters and IIR filters in general show quite slow roll-off. (Smith, 1999, p. 268)

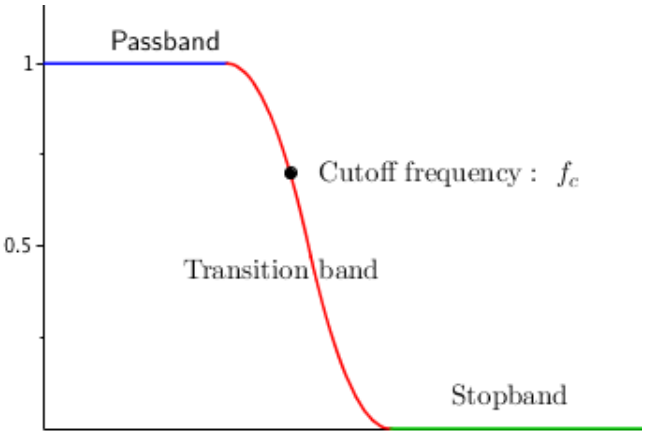


Figure 4.8: The frequency response of a filter with a relatively fast roll-off.

[1]A signal's power is measured in decibels (dB), which is a logarithmic unit of measurement. 20dB equals an increase in amplitude by one order of magnitude ($\cdot 10$) and -20dB a decrease by one order of magnitude ($\cdot 0.1$).

4.3.1 Low-Pass Filters

Low-Pass filters have their passband in the lower frequency ranges and their stopband in the higher ranges. A visualization of a low-pass bi-quad filter's frequency response when applied to a white noise signal[2] is given in Figure 4.8.

4.3.2 High-Pass Filters

A high-pass filter is a low-pass filter whose spectrum has either been *inverted* (flipped top-for-bottom) or *reversed* (flipped left-for-right) (Smith, 1999, p. 271). As a consequence, high-pass filters let high frequencies pass, while stopping lower frequencies. Figure 4.10 shows an appropriate frequency response.

4.3.3 Band-Pass Filters

Band-pass filters let frequency components in a specific range, e.g. 3000 to 5000 Hz, pass, while stopping all other frequencies lower or higher than the range of the passband. Such a filter can be created by first sending a signal through a low-pass and then through a high-pass filter. The passband of the filter created will then be the range of frequencies where the passband of the high-pass filter intersects that of the low-pass filter, since all other frequencies are in a stopband. Figure 4.11 shows the frequency response of a band-pass filter.

4.3.4 Band-Reject Filters

A band-reject filter, often referred to as a "notch" filter, can be seen as the opposite of a band-pass filter. While a band-pass filter stops all frequencies except for those immediately around the cutoff frequency, a band-reject filter stops only the frequencies at the cutoff frequency and lets all others pass. Besides using the correct filter coefficients for a bi-quad filter, a band-reject filter can also be created by using a low-pass and a high-pass filter *in parallel*. This involves copying a signal, then sending one copy of the original signal through a low-pass filter and the other copy through a high-pass filter. The final signal is the sum of the signal output by the high-pass filter and the signal output by the low-pass filter. Figure 4.12 displays the frequency response of a band-reject filter.

[2] A white noise signal is used here because it has a flat frequency spectrum when un-filtered.

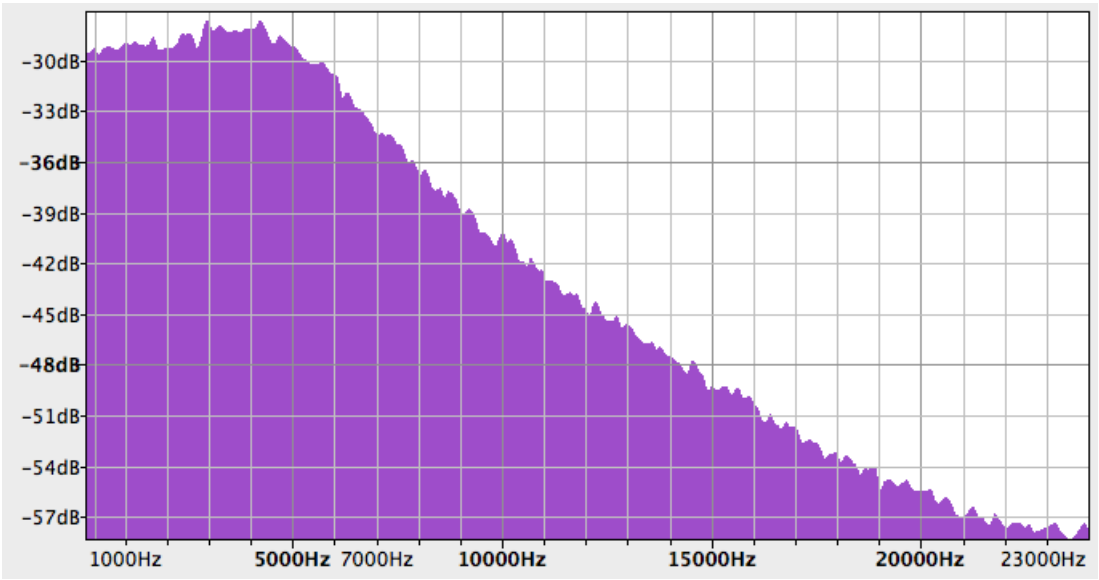


Figure 4.9: The frequency response of a low-pass bi-quad filter where $f_c = 5000$ Hz.

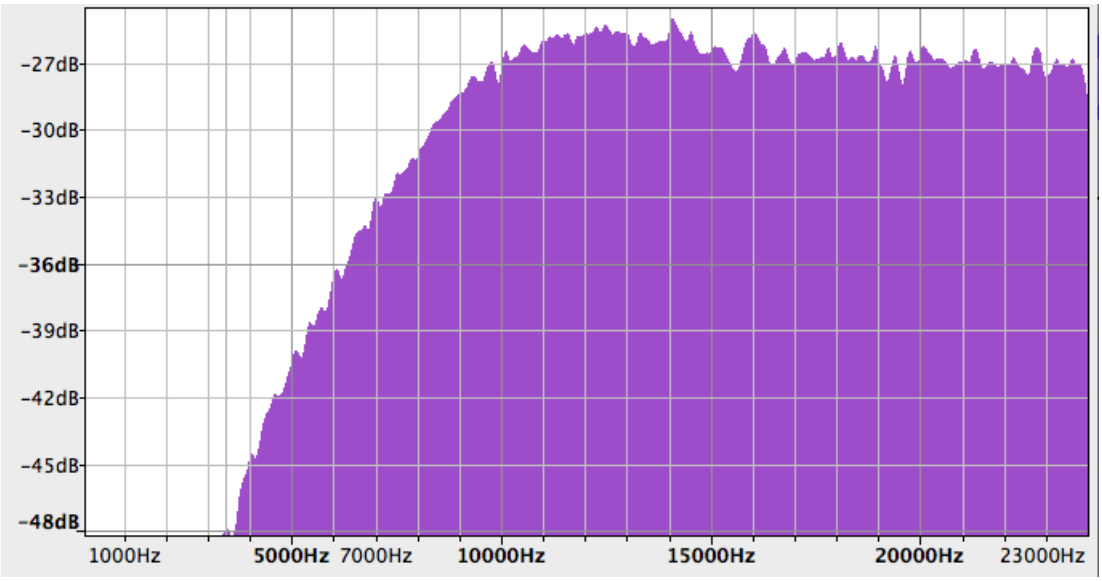


Figure 4.10: The frequency response of a high-pass bi-quad filter where $f_c = 10000$ Hz.

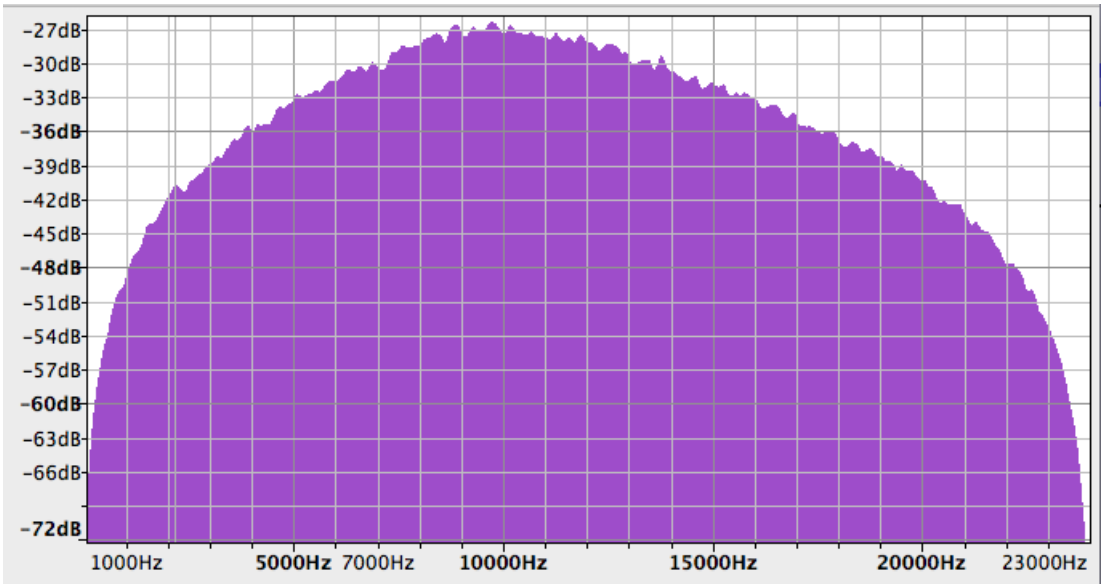


Figure 4.11: The frequency response of a band-pass bi-quad filter where $f_c = 10000$ Hz.

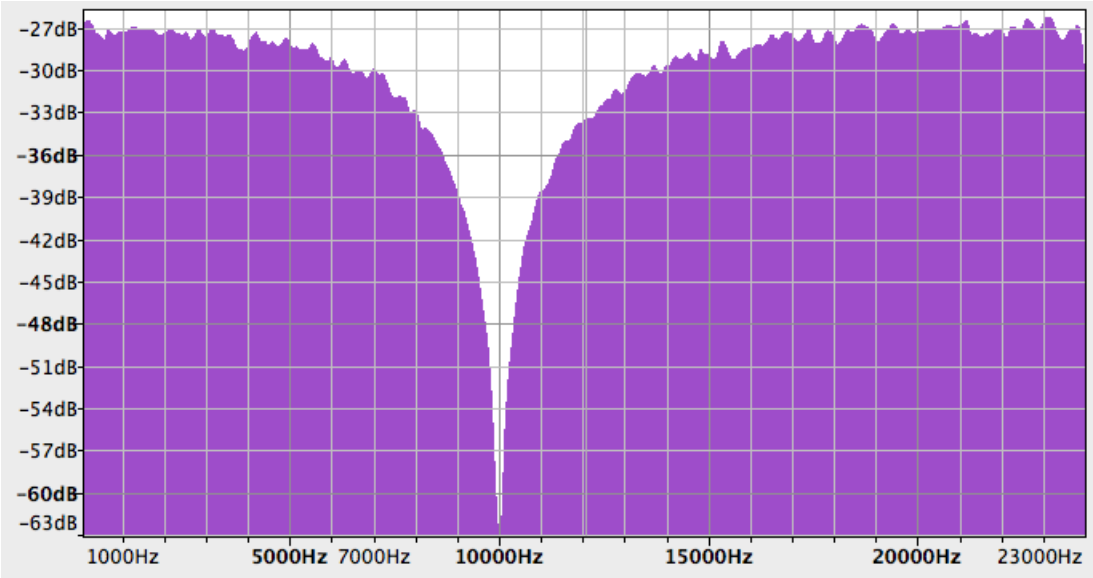


Figure 4.12: The frequency response of a band-reject bi-quad filter where $f_c = 10000$ Hz.

4.3.5 All-Pass Filters

A very special type of filter is the so-called all-pass filter. As its name implies, an all-pass filter lets all frequencies pass and stops none. What could possibly be the use of such a filter? The answer lies in the second alteration a filter performs on a signal: a change in phase. All-pass filters are used to change a signal's phase while leaving its amplitude un-altered at all frequencies.

4.4 Filter Coefficients

Let $\omega = \frac{2\pi f_c}{f_s}$ and $\alpha = \frac{\sin(\omega)}{2Q}$, where Q is the filter's *Quality Factor*[3], then Table 4.3 gives the values the various filter coefficients of a bi-quad filter must take on to achieve the wanted frequency response. These values were determined by Robert Bristow-Johnson from analog filter circuits. (Bristow-Johnson, `http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt`, accessed 4 January 2015)

Filter Type	a_1	a_2	b_0	b_1	b_2
Low-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1-\cos(\omega)}{2(1+\alpha)}$	$\frac{1-\cos(\omega)}{1+\alpha}$	$\frac{1-\cos(\omega)}{2(1+\alpha)}$
High-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1+\cos(\omega)}{2(1+\alpha)}$	$\frac{-(1+\cos(\omega))}{1+\alpha}$	$\frac{1+\cos(\omega)}{2(1+\alpha)}$
Band-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{\sin(\omega)}{2(1+\alpha)}$	0	$\frac{-\sin(\omega)}{2(1+\alpha)}$
Band-Reject	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1}{1+\alpha}$	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1}{1+\alpha}$
All-Pass	$\frac{-2\cos(\omega)}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{1-\alpha}{1+\alpha}$	$\frac{-2\cos(\omega)}{1+\alpha}$	1

Table 4.3: Filter coefficients for a bi-quad filter.

[3]The Quality Factor can be said to control to some extent the bandwith of a filter's transition band as well as the amplitude peak of the passband before transitioning to the stopband.

Chapter 5

Effects

While the digital revolution made it possible to create popular effects like Echos, Flangers or Reverbs more efficiently using logic circuits and digital signal processing, most effects found in digital synthesizers today have a long history of analog implementation. Nevertheless, they are still popular today and many digital synthesizers include them in their systems. The following sections will describe four common effects and explain their implementation.

5.1 Delay Lines and the Delay Effect

Delay lines store samples and *delay* their retrieval by a certain number of sample times. Just like delay lines are at the foundation of many digital filters, they are also the primary building block of a great number of different effects. Consequently, they must be examined and discussed thoroughly before attempting to create any other digital effects. However, having a well-founded understanding of delay lines also makes the implementation of many effects a trivial task. It should be noted that when a delay line is used as an effect simply to delay samples, it will be referred to as a "Delay" effect.

5.1.1 Simple Delay Lines

If the time by which a sample is delayed is constant for each sample, a delay line can be implemented using a simple First-In-First-Out (FIFO) data-structure such as a Queue. Figure 5.1 visualizes such a delay line and Table 5.1 shows a C++ implementation. (Mitchell, 2008, p. 118)

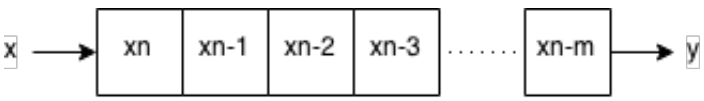


Figure 5.1: Visualization of a simple FIFO delay line.

```
class Delay
{
public:

    typedef unsigned long index_t;

    Delay(index_t length)
    : line_(new double [length]()),
      length_(length)
    { }

    ~Delay()
    {
        delete [] line_;
    }

    double process(double sample)
    {
        // Retrieve output sample
        double output = line_[0];

        // Move all elements forward by one
        for (index_t i = 0, j = 1; j < length_; ++i, ++j)
        {
            line_[i] = line_[j];
        }

        // Push the new sample into
        // into the delay line
        line_[length_ - 1] = sample;

        return output;
    }

private:

    index_t length_;

    double* line_;
};
```

Table 5.1: A C++ class that implements a very simple delay line of fixed size.

5.1.2 Flexible Delay Lines

While simple delay lines are easy to implement, they are not efficient if the delay time is not constant, as changing the size would require re-allocating memory to suit the new size and subsequently copying all items from the old delay line into the new one. Moreover, moving all samples by one index after retrieving the latest output sample is highly inefficient, especially for long delay times. A more reasonable approach is to implement a delay line as a circular buffer with a fixed maximum length and then to vary the distance between a read and a write iterator relative to the current delay time. The circularity is achieved by wrapping the read or the write index back to the front of the buffer if either reaches the end. Figure 5.2 shows such a circular buffer in an abstract visualization, where the write iterator, which points to the position where incoming samples are stored into the delay line, is currently at index 0. The Figure also shows two possible positions of the read iterator, from which delayed samples are retrieved, both resulting in a different delay time. It should be noted that read iterator A and B do not exist simultaneously, they just show possible indices for the read iterator. Read position A would cause a delay of 5 sample times while read position B would result in a 20 sample delay. Figure 5.3 shows the same buffer and iterators in a more realistic sequential layout, as it is stored in computer memory.

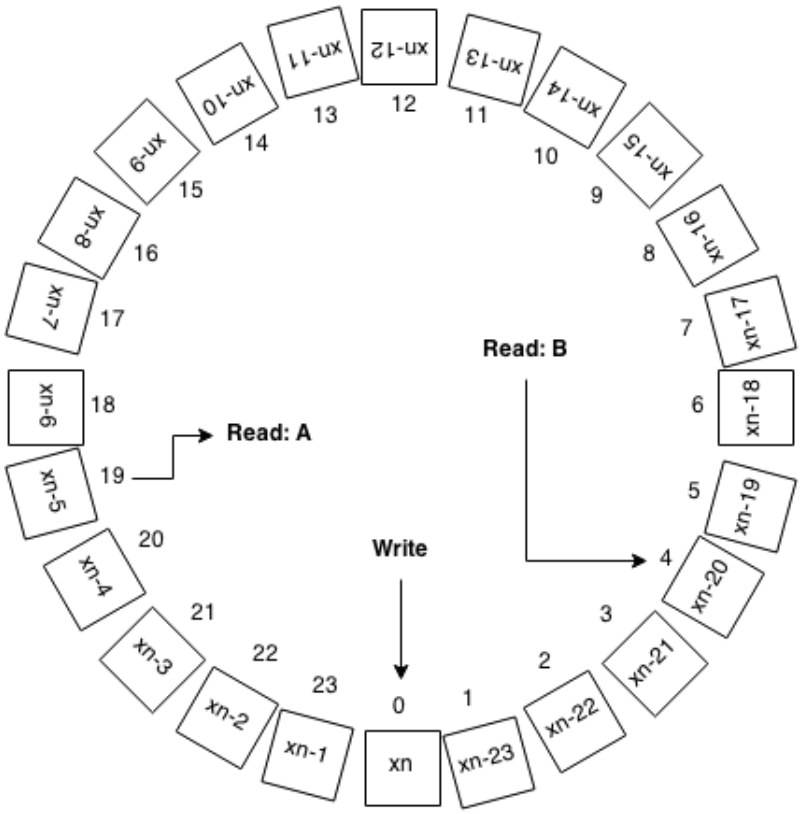


Figure 5.2: A circular delay buffer with two different possible read positions A and B. The labels inside the boxes are the relative sample delays and the labels outside the boxes are the sequential indices. The write iterator is where new samples are stored and the read iterators are where delayed samples are retrieved from.

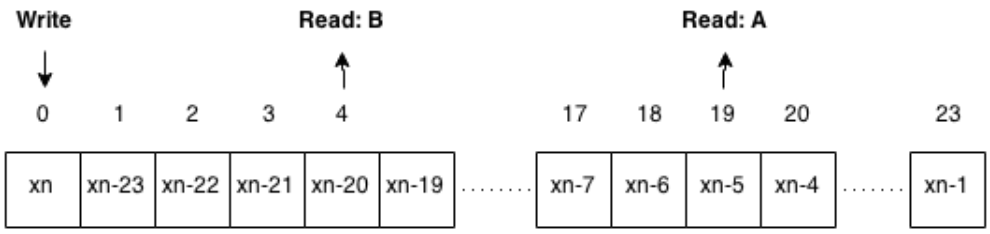


Figure 5.3: The buffer from Figure 5.2 in a sequential layout.

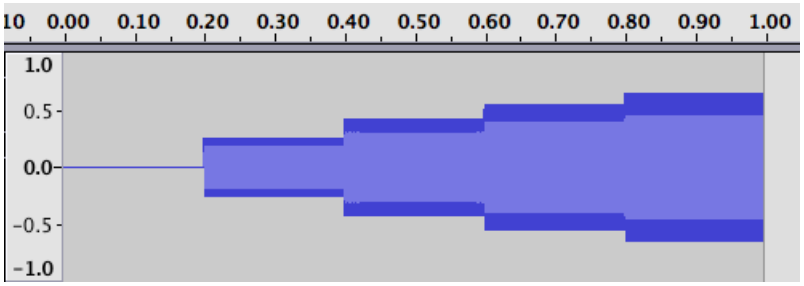


Figure 5.4: A sound wave resulting from a resonator. The initial silence is caused by the delay line filling up and delayed samples consequently still being equal to 0. The increase in amplitude per delay time (200 ms) is due to the feedback control. Because the decay is not 0, the amplitude does not double per delay time as it would if there were no decay and full feedback. Rather, the sound decays with time and would reach its maximum after 4 seconds.

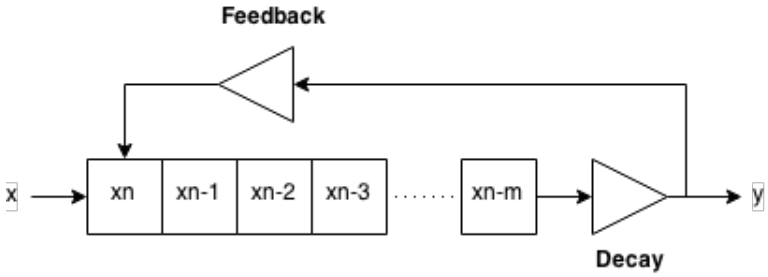


Figure 5.5: A resonating delay line.

5.1.3 Interpolation

It is also possible to have fractional delay values if samples are interpolated. Equation 5.1 shows how this interpolation process is performed, where i is the integral and f the fractional portion of the n-sample delay. Note that this is the same interpolation algorithm that is performed to retrieve sample values from fractional Wavetable indices.

$$y_n = x_i + ((x_{i-1} - x_i) \cdot f) \tag{5.1}$$

5.1.4 Feedback and Decay

Two parameters commonly associated with delay lines are *feedback* and *decay*. The feedback control of a delay line determines how much of the output signal is fed back into the delay line together with the input signal. The decay parameter determines the level of attenuation of the output signal as it leaves the delay line. It is "[...] applied each time the [fed-back] sample travels through the delay line and is equivalent to the dampening, or decay, of a signal over time". When these two parameters are used together, such a delay line is called a "Resonator". Figure 5.5 shows a block diagram for such a resonator and Figure 5.4 displays a sound wave resulting from a resonator. (Mitchell, 2008, p. 120)

5.1.5 Dry/Wet Control

Almost every effect, including the Delay effect, has what is called a "dry/wet" control. If the dry signal is the unchanged input signal and the wet signal the fully processed output signal,

the dry/wet control determines how much of an effect is applied to the dry signal. Equation 5.2 gives a mathematical definition for this principle, where dw is the dry/wet value between 0 (no effect) and 1 (full effect), x_n the unchanged, "dry", input sample and y_n the "wet" output signal from the effect.

$$z_n = (x_n \cdot (1 - dw)) + (y_n \cdot dw) \tag{5.2}$$

5.1.6 Implementation

In the synthesizer created for this thesis, a flexible delay line with all the properties and controls just mentioned is implemented in the `Delay` class. Relevant processing and updating methods of the `Delay` class are shown in Table 5.2.

```
1  double Delay::dryWet_(double originalSample, double processedSample)
2  {
3      return (originalSample * (1 - dw_)) + (processedSample * dw_);
4  }
5
6  void Delay::writeAndIncrement_(double sample)
7  {
8      *write_ = sample;
9
10     if (++write_ >= buffer_.end())
11     {
12         write_ -= buffer_.size();
13     }
14 }
15
16 double Delay::process(double sample)
17 {
18     const_iterator read = write_ - readIntegral_;
19
20     if (read < buffer_.begin())
21     {
22         read += buffer_.size();
23     }
24
25     if (! readIntegral_)
26     {
27         writeAndIncrement_(sample);
28     }
29
30     double output = *read;
31
32     if (--read < buffer_.begin())
33     {
34         read += buffer_.size();
35     }
36
37     output += (*read - output) * readFractional_;
38
39     output *= decayValue_;
40
41     if (readIntegral_)
42     {
43         writeAndIncrement_(sample + (output * feedback_));
44     }
45
46     return dryWet_(sample, output);
47 }
```

Table 5.2: Relevant member functions of the `Delay` class that implements a flexible delay line with feedback, decay, interpolation and dry/wet control.

5.2 Echo

The first effect that can be implemented very easily using just delay lines is the Echo effect. It is the result of summing an input sample with the output sample of a delay line. An implementation from the `Echo` class, derived from the `Delay` class, is shown in Table 5.3.

```
1 double Echo::process(double sample)
2 {
3     double output = sample + Delay::process(sample);
4
5     return dryWet_(sample, output);
6 }
```

Table 5.3: The `process` method of the `Echo` class. This shows that an Echo is just the input sample summed with the output from the delay line.

5.3 Flanger

A Flanger effect is created by mixing a signal with a delayed version of itself and varying the time by which it is delayed with a Low Frequency Oscillator. The middle delay time around which the LFO oscillates is called the "center" value. The term "depth" is used for the value that is added to or subtracted from the center value periodically. For example, if the center value is 10 ms and the depth 4 ms, the delay time will oscillate between 6 ms and 14 ms. The "rate" parameter controls how fast the LFO oscillates. A Flanger's "feedback" control is not the same as for delay lines, as it does not control how much of the output signal is fed back into the delay line. Rather, the feedback value determines how much of the sample at the center delay value is subtracted from the input sample. Varying the feedback controls the coloration and brightness of the sound produced. Lastly, a Flanger also has a dry/wet parameter, whose function was already described. Figure 5.6 depicts a block diagram for a Flanger. The sound produced by a Flanger effect can be described as a "swooshing" sound. (Mitchell, 2008, p. 136) Table 5.4 shows the `Flanger` class' `process` method.

5.4 Reverb

When a musician plucks a string on a guitar or bangs a percussion drum in a closed room, the sound that reaches a listener's ears does not stem solely from the sound's source. Rather, the sound emited from the musical instrument reflects off the room's walls and ceiling as well as any other object it meets on the way to the listener. This mixture of original and reflectd sound is called reverberation (Mitchell, 2008, p. 129). Naturally, the degree of reverberation depends, among other things, to a large part on the space of the room in which the sound is emitted. For example, an organ will sound differently if played in a large cathedral than it will in a small classroom. Therefore, any parameters that control the degree of reverberation essentially determine the size of the "virtual room" in which the sound is played. Most reverberators, reverbs for short, allow the user to control the reverb "time" and the reverb "rate". The reverb time determines how long it takes for the sound to reach inaudible levels, while the reverb rate controls at what rate this fading occurs. Finally, there is also a dry/wet control, as for all other effects as well.

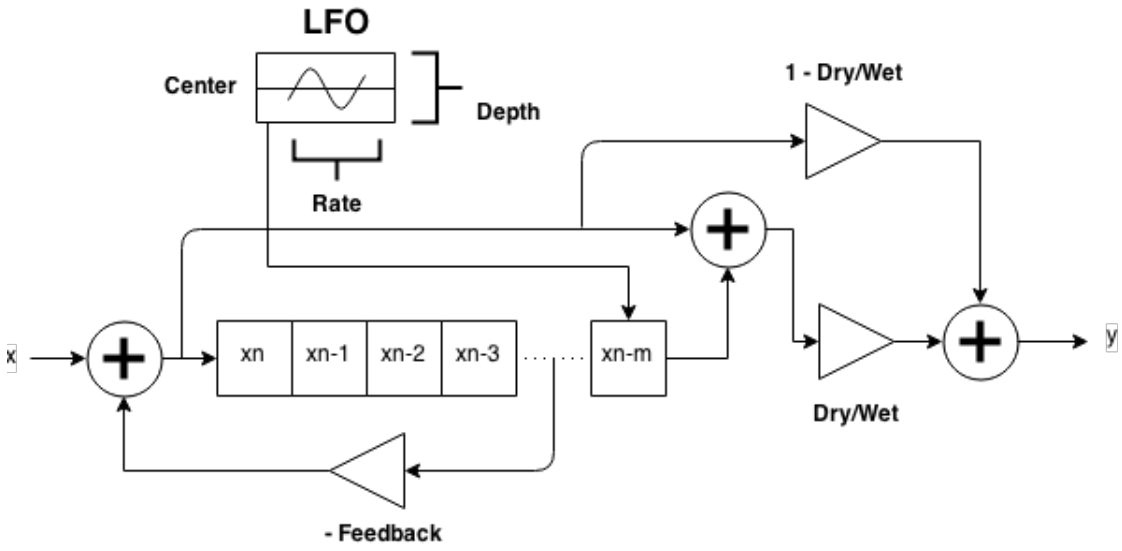


Figure 5.6: Block diagram for a Flanger effect. The LFO controls the delay length. The feedback is negative because it is subtracted from the input sample.

```
1  double Flanger::process(double sample)
2  {
3      double output = sample;
4
5      // Check for feedback
6      if (feedback_)
7      {
8          output -= delay_->offset(center_) * feedback_;
9      }
10
11     // Calculate new length by modulation. Modulation
12     // depth and maximum are 1 because the LFO's amplitude
13     // is the delay depth value
14     double length = lfo_->modulate(center_, 1, 1);
15
16     // Increment LFO
17     lfo_->update();
18
19     // Set the new length
20     delay_->setDelayTime(length);
21
22     // Retrieve new sample
23     output += delay_->process(output);
24
25     // Apply dry/wet
26     return dryWet_(sample, output);
27 }
```

Table 5.4: Member function of the `Flanger` class that implements flanging.

5.4.1 Schroeder Reverb

A relatively well-known and moderately popular reverb algorithm is the so-called "Schroeder Reverb", which consists of four parallel delay lines whose signals are fed into two all-pass filters connected in series. Figure 5.7 depicts a block diagram for a Schroeder Reverb. To simulate how sound waves reflect off different surfaces in a room, the delay lines are given different lengths. The decay parameter of these delay lines is determined by the reverb time and rate. The Schroeder Reverb's all-pass filters have fixed delay times and decay values. Table 5.5 gives these various delay line lengths and decay values. (Mitchell, 2008, p. 133)

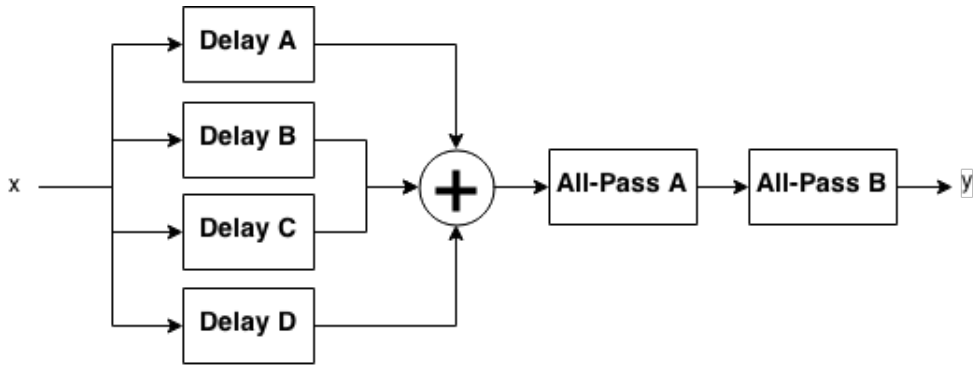


Figure 5.7: Block diagram for a Schroeder Reverb effect.

Component	Delay Time	Decay Value
Delay A	0.0297	Variable
Delay B	0.0371	Variable
Delay C	0.0411	Variable
Delay D	0.0437	Variable
All-Pass A	0.09638	0.0050
All-Pass B	0.03292	0.0017

Table 5.5: Delay times and decay values for a Schroeder Reverb.

5.4.2 Implementation

Table 5.6 shows relevant member of function of the `Reverb` class, which implements a Schroeder Reverb.

```
Reverb::Reverb(double reverbTime, double reverbRate, double dryWet)
: delays_(new Delay[4]),
  allPasses_(new AllPassDelay [2])
{

    delays_[0].setDelayTime(0.0437);
    delays_[1].setDelayTime(0.0411);
    delays_[2].setDelayTime(0.0371);
    delays_[3].setDelayTime(0.0297);

    allPasses_[0].setDecayTime(0.0050);
    allPasses_[0].setDelayTime(0.09638);

    allPasses_[1].setDecayTime(0.0017);
    allPasses_[1].setDelayTime(0.03292);

    setReverbTime(reverbTime);
    setReverbRate(reverbRate);
}

double Reverb::process(double sample)
{
    double output = 0;

    for (unsigned short i = 0; i < 4; ++i)
    {
        output += delays_[i].process(sample);
    }

    output = allPasses_[1].process(allPasses_[0].process(output));

    return dryWet_(sample, output);
}

void Reverb::setReverbRate(double reverbRate)
{
    for (unsigned short i = 0; i < 4; ++i)
    {
        delays_[i].setDecayRate(reverbRate);
    }
}

void Reverb::setReverbTime(double reverbTime)
{
    for (unsigned short i = 0; i < 4; ++i)
    {
        delays_[i].setDecayTime(reverbTime);
    }
}
```

Table 5.6: This code exerpt from the `Reverb` class shows how the Schroeder Reverb's delay lines and all-pass filters are initialized and then used to reveberate a signal.

Chapter 6

Synthesizing Sound

The most important feature of a synthesizer is its capability to synthesize sound. Synthesizing sound means to combine two or more (audio) signals to produce a new signal. The many possibilities to synthesize sound waves with variable parameters enable musicians to create an uncountable number of different sounds. A synthesizer can be configured to resemble natural sounds such as that of wind or water waves, can emulate other instruments like pianos, guitars or bass drums and, finally, a synthesizer can also be used to produce entirely new, electronic sounds. However, not all synthesis methods available to the creator of a digital synthesizer are equally suited to the various possible sounds just described. This chapter will examine four popular methods of synthesis — Additive, Subtractive, AM and FM synthesis — while focusing especially on the last technique, which was implemented in C++ for the purpose of this thesis.

6.1 Additive Synthesis

Additive Synthesis was already introduced in Chapter 2 as a method to produce complex waveforms. It involves the summation of a finite set of waveforms that can either be simple sinusoids or complex waves themselves — such as sawtooth or square waves. As it was already shown, this summation — formally called a Fourier Series — can produce a myriad of different waveforms. An example for an Additive Synthesizer is Native Instrument's *Razor*, which lets the user additively synthesize up to 320 partials. Also a simple church organ, whose characteristic sound is produced by the summation of the sounds emitted by its tubes, is an additive synthesizer. In terms of the natural world, Additive Synthesis occurs when any two sound waves meet and combine to produce a new sound.

6.2 Subtractive Synthesis

While Additive Synthesis creates sounds by summing many individual waveforms, Subtractive Synthesis starts out with a complex waveform very rich in harmonics, like a sawtooth wave, and then subtracts or "carves" away parts of that sound by filtering or attenuating selected frequencies. A very prominent example of Subtractive Synthesis is the human voice, which makes use of the larynx to shape air coming from the lungs in order to produce certain phonemes, which eventually make up words, sentences and our ability to communicate.

6.3 Amplitude Modulation Synthesis

During the discussion of Low Frequency Oscillators (LFOs) in Chapter 3, it was mentioned how a vibrato effect can be achieved by varying the amplitude of a sound using an LFO with a frequency f in the approximate range of $]0; 20]$ Hertz. When the frequency of modulation is in the audible range of 20 to 20000 Hertz, this modulation is termed "Amplitude Modulation" (AM). AM produces side-bands in the signal's frequency spectrum and thus changes the sound's timbre. In Amplitude Modulation Synthesis the original signal is termed the "carrier" signal, $c(t)$, and the modulation signal the "modulator", $m(t)$. Equation 6.1 shows a formula for Amplitude Modulation (Synthesis) with two signals, where A_c is the carrier amplitude and A_m that of the modulator. Given that already two oscillators can produce a great variety of interesting sounds, Amplitude Modulation Synthesis is a very popular synthesis method found in both the analog as well as the digital realm.

$$f(t) = (A_c + A_m \cdot \sin(\omega_m t + \phi_m)) \cdot \sin(\omega_c t + \phi_c) \tag{6.1}$$

6.4 Frequency Modulation Synthesis

The difference between Amplitude Modulation (AM) and Frequency Modulation (FM) Synthesis is that in the latter, the modulator varies the carrier's frequency as opposed to its amplitude. This produces very complex changes in the carrier's frequency spectrum and introduces a theoretically infinite set of side-bands, which contribute to the characteristic sound and timbre of FM Synthesis. The fact that Frequency Modulation can be used to synthesize audio signals was first discovered by John Chowning, who he described the mathematical principles and practical implications of FM Synthesis in his 1973 paper "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation". In 1974 his employer, Stanford University, licensed his invention to the Yamaha Corporation, a Japanese technology firm, which went on to create the first FM synthesizers, including the very popular DX-7 model. Many digital FM synthesizers, such as Native Instrument's *FM8*, try to emulate the DX-7. (Electronic Music Wiki, `http://electronicmusic.wikia.com/wiki/DX7`, accessed 4 January 2015) Equation 6.4 gives a full mathematical definition for Frequency Modulation (Synthesis). What Equation 6.4 shows is that FM Synthesis works by summing the carrier's instantaneous frequency ω_c, the carrier signal being $c(t)$, with the output of the modulator signal $m(t)$, thereby varying the carrier frequency periodically. The degree of frequency variation Δf_c depends on the modulator's amplitude A_m. Therefore, $\Delta f_c = A_m$. Figure 6.1 shows how frequency modulation effects a signal. Note that this Figure should only show how FM works. It is not a realistic example of FM *Synthesis*, as the modulator frequency is not in the audible range.

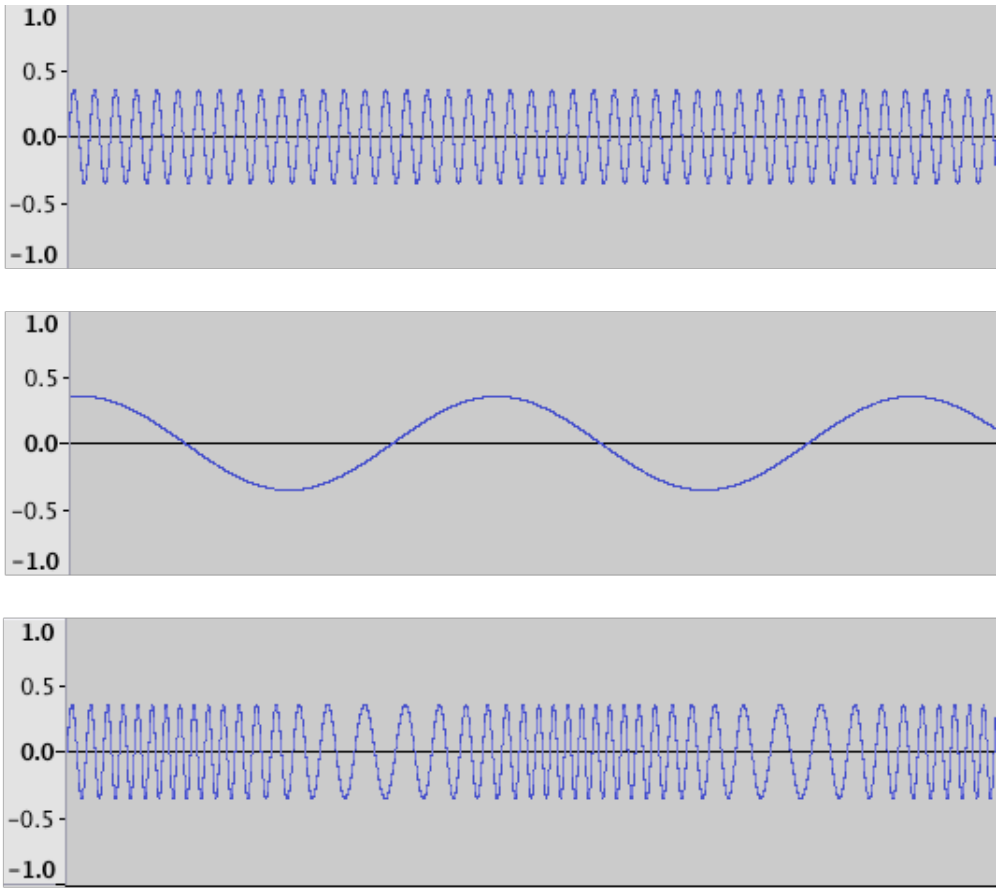


Figure 6.1: The first figure shows the carrier signal $c(t)$ with its frequency f_c equal to 100 Hz. The signal beneath the carrier is that of the modulator, $m(t)$, which has a frequency f_m of 5 Hz. When the modulator amplitude A_m is increased to 50 (instead of ~ 0.4, as shown here) and used to modulate the frequency of the carrier, the last signal $f(t)$ is produced. While these figures show how FM works, they are not a good example of FM synthesis, because the modulator frequncy is not in the audible range.

$$c(t) = A_c \cdot \sin(\omega_c t + \phi_c) \qquad (6.2) \qquad m(t) = A_m \cdot \sin(\omega_m t + \phi_m) \qquad (6.3)$$

$$f(t) = A_c \cdot \sin((\omega_c + m(t))t + \phi_c) = A_c \cdot \sin((\omega_c + A_m \cdot \sin(\omega_m t + \phi_m))t + \phi_c) \qquad (6.4)$$

6.4.1 Sidebands

One of the most noticeable effects of FM Synthesis is that it adds *sidebands* to a carrier signal's frequency spectrum. Sidebands are frequency components higher or lower than the carrier frequency, whose spectral position, amplitude as well as relative spacing depends on two factors: the ratio between the carrier and modulator frequency, referred to as the "C:M ratio", and the index of modulation β, which in turn depends on the modulator signal's amplitude and frequency.

6.4.2 C:M Ratio

The spacing and positions of sidebands on the carrier signal's frequency spectrum depends on the ratio between the frequency of the carrier signal, C, and that of the modulator, M. This $C : M$ ratio gives insight into a variety of properties of a frequency-modulated sound. Most importantly, when the $C : M$ ratio is known, Equation 6.5 gives all the relative frequency values of the theoretically infinite set of sidebands. Equation 6.6 describes how to calculate ω_{sb_n}, the angular frequency of the n-th sideband, absolutely. What these equations show is that for any given $C : M$ ratio, the sidebands are found at relative frequencies $C+M, C+2M, C+3M, ...$ and absolute frequencies $\omega_c + \omega_m, \omega_c + 2\omega_m, \omega_c + 3\omega_m, ...$ Hertz.

$$\omega_{sb_n} = C \pm n \cdot M, \text{where } n \in [0; \infty[\tag{6.5}$$

$$\omega_{sb_n} = \omega_c \pm n \cdot \omega_m, \text{where } n \in [0; \infty[\tag{6.6}$$

When examining these two equations one may notice that there also exist sidebands with negative frequencies, found relatively at $C - M, C - 2M, C - 3M$ and so on. Simply put, a signal with a negative frequency is equal to its positive-frequency counterpart but with inverted amplitude, which can also be seen as a 180° or π radian phase-shift (`http://www.sfu.ca/~truax/fmtut.html`, accessed 30 December 2014). Equation 6.7 defines this formally. What this also means is that if there is a sideband at a frequency f and another sideband at $-f$ Hertz, these two sidebands will phase-cancel completely if their amplitudes are the same. If not, the positive side-band will be reduced in amplitude proportionally. Consequently, it may happen that also the original carrier frequency is reduced in amplitude, for example at a $C : M$ ratio of 1:2, as the first "lower" sideband, meaning with a lower frequency than the carrier, is at $C - M = C - 2C = -C$ Hertz.

$$A \cdot \sin(-\omega t + \phi) = -A \cdot \sin(\omega t + \phi) = A \cdot \sin(\omega t + \phi - \pi) \tag{6.7}$$

Figure 6.2 shows the frequency spectrum of a carrier signal with a frequency f_c[1] of 200 Hz, modulated by a modulator signal with its frequency f_m equal to 100 Hz. The carrier frequency is seen on the spectrum as the peak at 200 Hz. The other peaks are the sidebands. Because the $C : M$ ratio here is $2 : 1$, the first two lower sidebands are found at relative positions $C - M = 2 - 1 = 1$ and $C - 2M = 2 - 2 = 0$, relative position 2 being the carrier. The first two upper sidebands have absolute frequency values of $\omega_c + \omega_m = 200 + 100 = 300$ and $\omega_c + 2 \cdot \omega_m = 200 + 200 = 400$ Hertz.

[1]Not to be confused with the cutoff frequency in the context of filters, which is also denoted by f_c

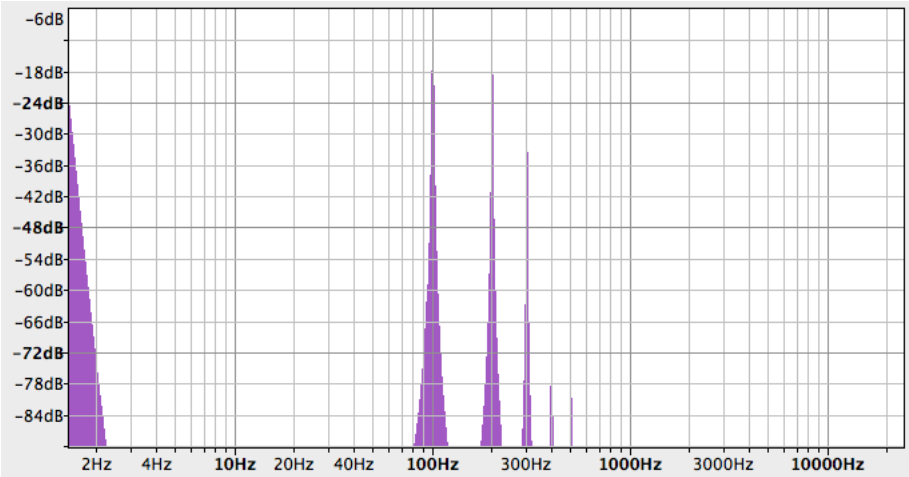


Figure 6.2: The frequency spectrum of a $C : M$ ratio of $2 : 1$, where carrier frequency f_c is 200 and the modulator frequency f_m 100 Hertz.

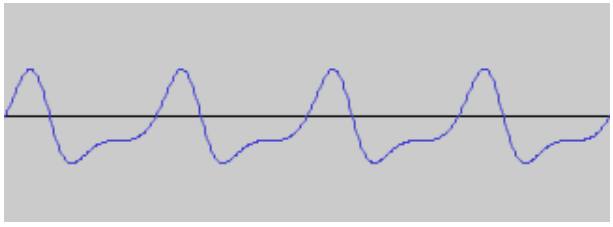


Figure 6.3: A sound wave produced by FM Synthesis that resembles a sawtooth wave with two partials visually as well as acoustically. The reason why is that the $C : M$ ratio was $1 : 1$ in this case, causing sidebands to be a perfect harmonic series.

The $C : M$ ratio can also help to understand how a frequency-modulated signal will sound. For example, a ratio of $1 : 2$ will sound similar to a square wave, since the harmonic series produced by this ratio will have sidebands at $3C, 5C, 7C$ etc., so, all odd partials. On the other hand, a $C : M$ ratio of $1 : 1$ will cause the resulting sound to resemble a sawtooth wave, as the sidebands are positoned at $2C, 3C, 4C, ...$, which fits the requirement of a sawtooth wave to be composed of *all* partials. Figure 6.3 shows how a ratio of $1 : 1$ causes the signal to look a lot like a sawtooth wave with two partials. Even though this waveform includes all partials, it does not look like a perfect sawtooth wave, as it would if it had been created by additive synthesis. The reason why is that the amplitude of each partial is not the inverse of the partial number, but some other value. The same applies to square waves.

There is also a way to find out if a $C : M$ ratio will result in a harmonic series or in an inharmonic series. To re-cap, a harmonic *sideband* is an integer multiple of the carrier frequency while an inharmonic sideband is not. A harmonic *series* with a harmonic *ratio*, such as $1 : 2$, includes only harmonic sidebands and has the carrier frequency as the fundamental pitch, while an inharmonic series with an inharmonic ratio, e.g. $2 : 5$, includes one or more inharmonic sidebands. For a general procedure to determine the *harmonicity* of a $C : M$ ratio, the ratio first has to be converted to *Normal Form*. In Normal Form, "M must be greater or equal to twice C or else be the ratio $1 : 1$". To *normalize* a ratio that is not in Normal Form, the operation $C = |C - M|$ is performed until the Normal Form criterion is met. For example, to normalize a ratio of $5 : 2$: $5 - 2 = 3 \rightarrow 3 - 2 = 1 \Rightarrow 1 : 2$. Once a ratio has been brought into Normal

Form, the rule to determine if a ratio will result in a harmonic series is the following: "harmonic [Normal Form] ratios are always of the form $1 : N$, and inharmonic ones [are not]". Therefore, normalized ratios like $1 : 2, 1 : 5, 1 : 10$ are inharmonic, while examples for inharmonic ratios are $2 : 9, 3 : 8$ or $4 : 9$. (`http://www.sfu.ca/~truax/fmtut.html`, accessed 30 December 2014)

6.4.3 Index of Modulation

Theoretically, Frequency Modulation Synthesis produces an infinite number of sidebands. However, sidebands reach inaudible levels very quickly, making the series practically finite. In general, the amplitude of individual sidebands depends on the "index of modulation" and the Bessel Function values that result from it. A definition for the index of modulation, denoted by β, is given in Equation 6.8. Because the variation in carrier frequency, Δf_c, depends directly on the amplitude of the modulator, Equation 6.8 can be re-written as Equation 6.9.

$$\beta = \frac{\Delta f_c}{f_m} \tag{6.8}$$

$$\beta = \frac{\Delta f_c}{f_m} = \frac{A_m}{f_m} \tag{6.9}$$

The index of modulation can be used to determine the amplitude of individual sidebands, if input into the Bessel Function, shown in Equation 6.10, where n is the sideband to calculate the amplitude for. Table 6.1 shows amplitude values for the n-th sideband, given an index of modulation β. These values were derived from the Bessel Function. Only values above 0.01 are shown.

$$J(n, \beta) = \sum_{k=0}^{\infty} \frac{(-1)^k \cdot (\frac{\beta}{2})^{n+2k}}{(n+k)! \cdot k!} \tag{6.10}$$

β	Sideband								
	Carrier	1	2	3	4	5	6	7	8
0	1								
0.25	0.98	0.12							
0.5	0.94	0.24	0.03						
1.0	0.77	0.44	0.11	0.02					
1.5	0.51	0.56	0.23	0.06	0.01				
2.0	0.22	0.58	0.35	0.13	0.03				
3.0	-0.26	0.34	0.49	0.31	0.13	0.04	0.01		
4.0	-0.40	-0.07	0.36	0.43	0.28	0.13	0.05	0.02	
5.0	-0.18	- 0.33	0.05	0.36	0.39	0.26	0.13	0.05	0.02

Table 6.1

Another useful property of the index of modulation is that it can be used to provide an intuitive interface to the user of a synthesizer. Instead of having to control the absolute modulator amplitude, which can be rather hard to tune, some synthesizers provide the user with a range of $[0; 10]$ to control β. Behind the scenes, the modulator amplitude A_m is then calculated as $\beta \cdot f_m$, where f_m is the modulator frequency. Moreover, also the modulator frequency is often not shown absolutely in a synthesizer's interface, but relative to the carrier's frequency, e.g. at a ratio of 2 times the carrier frequency f_c (the $C : M$ ratio is therefore

always $1 : M$). (JCPedroza, `http://sound.stackexchange.com/questions/31709/what-is-the-level-of-frequency-modulation-of-many-synthesizers`, accessed 3 January 2015)

6.4.4 Bandwidth

The bandwidth of a signal describes the range of frequencies that signal occupies on the frequency spectrum. The bandwidth of a frequency-modulated signal is theoretically infinite. However, it was already shown that the amplitude values of a carrier's sidebands quickly become inaudible and thus negligible. A rule of thumb for calculating the bandwidth B of an FM signal is the so-called "Carson Bandwidth Rule", given in Equation 6.11.

$$B \approx 2(\Delta f_c + f_m) = 2(A_m + f_m) = 2f_m(1 + \beta) \tag{6.11}$$

6.4.5 Algorithms and Operators

When speaking of FM Synthesis, it is common to refer to oscillators as "operators". This naming convention stems from the Yamaha DX-7 series. So far, FM Synthesis was only discussed for two operators, a carrier and a modulator. However, it is entirely possible to perform FM Synthesis with more than two operators, by modulating any number of operators either in series, or in parallel. When three operators A, B and C are connected in series, A modulates B, which in turn modulates C. If A and B are connected in parallel, but in series with C, C is first modulated by A and then by B. The same result is achieved if the sum of signals A and B is used to modulate C. In general, a configuration of operators is referred to as an "algorithm". The number of possible algorithms increases with an increasing number of operators. In the synthesizer created for this thesis, four operators A, B, C and D are used. The possible algorithms for these four operators are shown in Figure 6.4.

6.4.6 Implementation

The `Operator` class implements an FM operator and inherits from the `Oscillator` class. Its frequency is modulated by adding an offset to its Wavetable index increment, proportional to the frequency variation caused by the modulator. Table 6.2 shows how this offset is calculated and added to the Wavetable index whenever the Operator is updated. Listing A.6 shows how the `FM` class, which takes pointers to four Operators, implements the various algorithms shown in Figure 6.4.

```
1  void Operator::modulateFrequency(double value)
2  {
3      modOffset_ = Global::tableIncr * value;
4  }
5
6  void Operator::update()
7  {
8      // Normal frequency index increment +
9      // Index increment for frequency modulation value
10     increment_(indIncr_ + modOffset_);
11 }
```

Table 6.2: Two member functions from the `Operator` class that show how the frequency of an `Operator` object can be modulated.

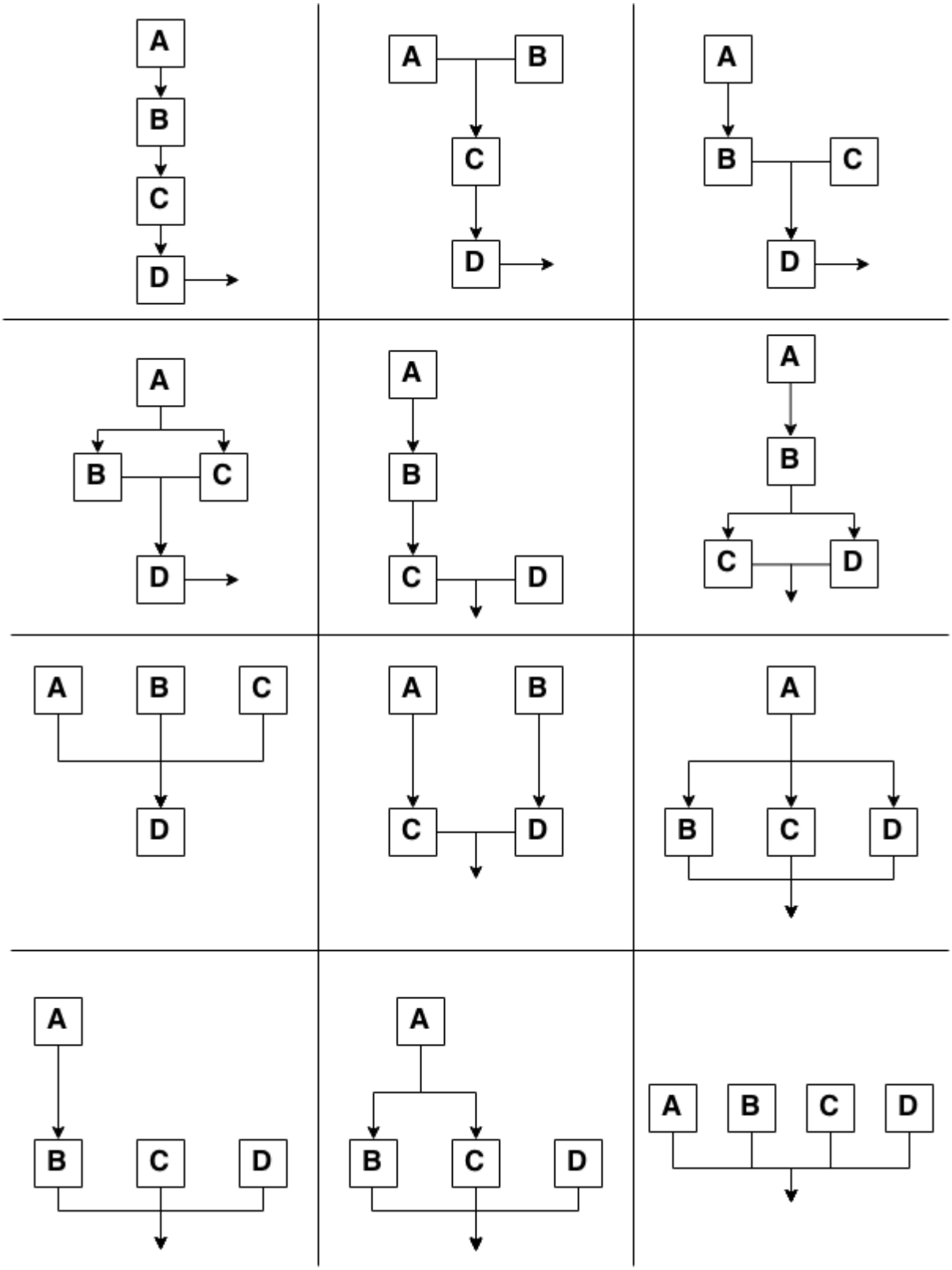


Figure 6.4: Signal flows for FM Synthesis algorithms with four operators, A, B, C and D. A direct connection between two operators means modulation of the bottom operator by the top operator. Signals of operators positioned side-by-side are summed.

Chapter 7

Making Sound Audible

There are two ways to make a computer-generated signal audible. It can either be sent directly to the sound card, often called the digital-to-analog-converter (DAC), or be recorded and stored to a file, for later playback. The following sections will examine these two possibilities and their implementation in C++.

7.1 Recording Sound

To record and store a sound for later playback, it must be written to a file. For digital audio recordings, there are a set of well-known and widely supported file formats with different benefits and detriments. Some are lossless, some lossy. Some are compressed, some uncompressed. Examples include MPEG-3 with its .mp3 file extension, Vorbis (.ogg), the Audio Interchange File Format (.aiff) or the Wave Audio File Format (.wav), usually referred to as WAVE. The latter is arguably the easiest to work with, as it requires relatively little configuration and stores raw, uncompressed sample data, making it very simple to write a sample buffer to a WAVE file. Consequently, the Wave Audio File Format will be further discussed.

7.1.1 WAVE files

WAVE files are part of the family of Resource Interchange File Format (RIFF) specifications, developed by Microsoft. (Wilson, 2003)

> *The RIFF specification defines the file as a set of nested 'chunks' of data. Each chunk begins with an 8-byte chunk header. The chunk header identifies the chunk type with a 4-character ID followed by the chunk size as a 4-byte binary value.* (Mitchell, 2008, p. 33)

The first of these chunks is the top-level RIFF chunk, which contains basic information about the type of the RIFF file, e.g. that the WAVE specification is used. Then follow two sub-chunks, the Format Chunk and the Data Chunk. The first is for information concerning the file and file data's format, such as the sample rate, the number of bytes used per sample or the number of channels. The second stores the sample data and its overall size. Figure 7.1 displays the structure of a WAVE file.

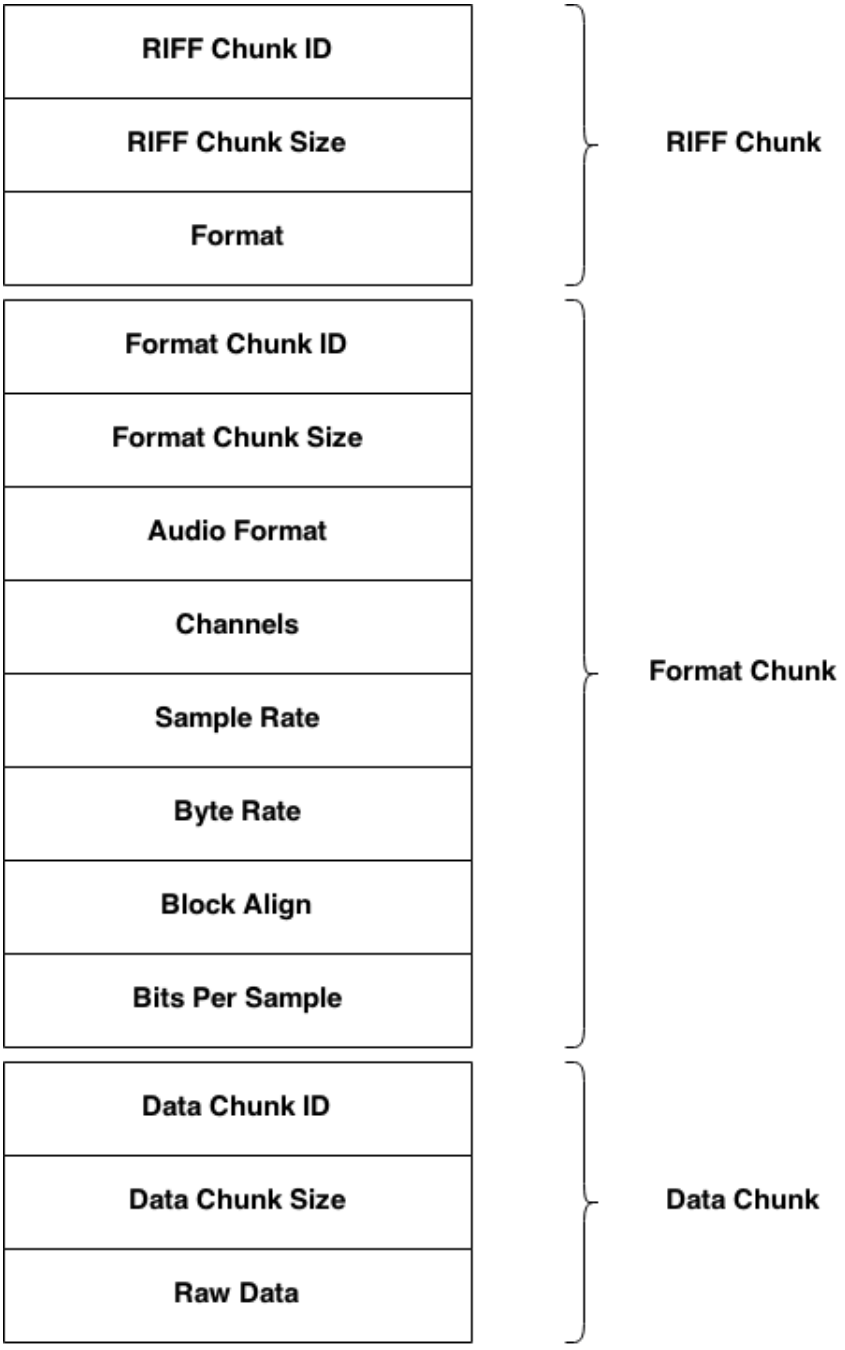


Figure 7.1: The WAVE file format specification.

Discussion of WAVE file chunks

The following list dicusses the individual chunks of a WAVE file in detail.

RIFF Chunk

RIFF Chunk ID
Identifier for the family of RIFF specifications, stored as a 4-byte string (`"RIFF"`).

RIFF Chunk Size
The size of everything to follow in the RIFF file. Determined after the sample data has been written to the file at the very end.

Format
The specific RIFF format used, in this case the WAVE audio format. Stored as a 4-byte string (`"WAVE"`).

Format Chunk

Format Chunk ID
The identifier of the sub-chunk, stored as a 4-byte string (`"fmt"`).

Format Chunk Size
The size of the sub-chunk. For WAVE files, the size of the Format Chunk is always 16 (bytes).

Audio Format
The type of quantization to be used for the audio data. The standard is Pulse-Code-Modulation (PCM), also called linear quantization. Setting the Audio Format to 1 means that the file uses PCM.

Channels
The number of channels for the audio data. Usually either one ("mono") or two ("stereo").

Sample Rate
The sample rate used for the audio data. Common values are 48000 or 44100.

Byte Rate
The number of bytes to stream per second, equal to the Sample Rate times the Block Align value.

Block Align
The number of bytes used per sample block, equal to the Bits Per Sample times the number of channels, divided by 8 to convert bits to bytes.

Bits Per Sample
The amount of bits used to represent a single sample value. Usually samples are stored as 16-bit signed integers, so this value can be set to 16.

Data Chunk

Data Chunk ID
4-byte identifier for the Data sub-chunk (`"data"`).

Data Chunk Size
The size of the sample data in bytes.

Raw Data
The raw, binary sample data.

Implementation

The `Wavefile` class implements a means to store sample data to a WAVE file. All the various chunks are stored in a simple `struct` and pre-initialized if possible. Samples can be written to an internal buffer and when the user wishes to write the recorded samples to a file, the relevant data members of the WAVE header are updated and written to disk together with the raw sample data. Listings A.7 and A.8 shows the full definition and implementation of the `Wavefile` class.

7.2 Direct Output

The second method of making sound audible is to send it to the sound card or digital-to-analog-converter (DAC). To do so, one has to interface with the Operating System's sound processing capabilities. The difficulty here is that each Operating System (OS) has its own sound API (Application Program Interface). Moreover, there may be different APIs for a single OS, as shown in Table 7.1. The pragmatic, cross-platform-oriented programmer does not interface with each of these APIs him- or herself, but makes use of one of the many well-documented and well-implemented Open-Source libraries, such as the RtAudio[1] library. This library eases the implementation of direct sound output in a digital synthesizer by adjusting to one of the many sound APIs, as needed. All the programmer needs to do is implement a simple call-back function, which the RtAudio library calls whenever the OS and the sound API request new audio data.

	Windows	OS X	Linux
ALSA	No	No	Yes
OSS	No	No	Yes
PulseAudio	No	No	Yes
Jack	No	Yes	Yes
CoreAudio	No	Yes	No
DirectSound	Yes	No	No
ASIO	Yes	No	No
WASAPI	Yes	No	No

Table 7.1: Sound APIs and what Operating Systems they are available for.

[1]Found at: `http://www.music.mcgill.ca/~gary/rtaudio/index.html`

Chapter 8

MIDI

While the previous chapter discussed how to output from a synthesizer, this chapter will examine how to input to it. The standard way to communicate with any digital instrument in real-time is the *Musical Instrument Digital Interface* (MIDI) protocol, which specifies the physical or virtual connections and the transmision format used between, for example, a keyboard and a music synthesizer. A MIDI message is composed of a sequence of bytes that have different meanings and can control a variety of parameters and settings in a digital synthesizer. For example, MIDI can be used to control the amplitude of an Operator or the Attack time of an Envelope. However, the most important use of MIDI is to send musical notes, which will be discussed in this chapter.

The first transmission byte of any MIDI message, no matter its purpose, is called the STATUS byte. The most signficant bit (MSB[1]) of a STATUS byte is set to 1, while all other MIDI transmission bytes have their MSB set to 0. The other three bits of a STATUS byte's high nibble[2] are reserved for specifying other information about the message. For example, when sending musical notes, having the least signficiant bit (LSB) of the high nibble set to 1 signifies a NOTE ON signal, meaning the musician pressed a key on his or her keyboard. Conversely, if this bit is set to 0, it means that the musician released the key. This is called a NOTE OFF signal. The low nibble of a STATUS byte, meaning the four least significant bits, specify the channel number. MIDI allows transmissions to be sent over any of 16 channels, numbered 0 to 15, making it possible to connect a digital synthesizer to up to 16 different physical or virtual instruments. Message 8.1 gives an example of a STATUS byte, as can be identified by the fact that the MSB is set to 1. Because the LSB of the high nibble is also set to 1, this message stands for a NOTE ON signal. In this example, the message is sent over channel number 4 (0100_2). (Vandenneucker, 2012)

$$10010100_2 \tag{8.1}$$

[1]The most signficiant bit (MSB) refers to the bit with the highest value in a byte. When numbering a bit from 7 to 0, the 7th bit has a value of 128 when set and is the most signficant bit, while the 0th bit has a value of 1 when set and is the least signficant bit (LSB).

[2]A nibble is one half of a byte. The "high" nibble refers to the four most significant bits of a byte (bits 7-4) and the "low" nibble to the four least significant bits (bits 3-0).

All subsequent bytes of a MIDI message, after the STATUS byte, are referred to as DATA bytes and specify any other information a digital musical instrument may need. Because the MSB of any MIDI byte is reserved to give information about its type (1 for STATUS, 0 for DATA), DATA bytes have 7 bits available to transmit actual data. For this discussion, only musical note messages are interesting. Therefore, the space of 128 values (0-127) available to a DATA byte means that a music keyboard can send up to 128 different musical notes to a digital synthesizer, that correspond and can be converted to certain frequencies. For example, piano key A4 is defined as MIDI note 69 and has a frequency of 440 Hz. Equation 8.2 gives a formula to calculate a frequency value f from a MIDI note n and Equation 8.3 shows the reverse process. Message 8.4 is a DATA byte that signifies that MIDI note 60 should be played, which stands for piano key C4 and a frequency of ca. 261.63 Hertz. (Vandenneucker, 2012)

$$f = \sqrt[12]{2^{n-69}} \cdot 440 \tag{8.2}$$

$$n = 12 \cdot log_2(\frac{f}{440}) + 69 \tag{8.3}$$

$$00111100_2 \tag{8.4}$$

MIDI messages for musical notes also include one more DATA byte, which specifies the note's velocity (how hard the musician hit the piano key), again in a range of 0 to 127. This information can be used by the creator of digital synthesizer, but it is not a must. The synthesizer created for this thesis only checks if the velocity is equal to 0, which is interpreted in the same way as a NOTE OFF signal.

Implementation

In *Anthem*, MIDI communication was implemented in the `MIDI` class, which makes use of the Open-Source RtMidi library[3]. Similar to how direct audio output was implemented, also the `MIDI` class has a call-back method, which is invoked whenever the RtMidi library detects a new MIDI message. Messages are then interpreted and Operators' frequencies updated accordingly.

[3]Found at: `http://www.music.mcgill.ca/~gary/rtmidi/index.html`

Conclusion

Digital music synthesis and audio processing are ongoing fields of study. New synthesis methods — such as physical modelling — as well as incumbent ones are actively being researched and perfected, to provide ever crisper and higher-quality sound. Reverberation algorithms implemented in professional audio software have become many times more complex than the Schroeder Reverb presented in chapter 6, making digital sound resemble its natural counterpart ever more strikingly. Filters are continuously being improved as well, ensuring that frequency responses have the fastest possible roll-off in the transition band, highest amount of attenuation in the stop band and least in the pass band.

This thesis clearly did not attempt to examine such state-of-the-art techniques or cutting-edge research. An entire thesis would have to be dedicated to every chapter — if not every section — presented here, if it intended to examine what current methods yield the best possible results. Rather, what this thesis aimed at was describing the most straight-forward and well-established practices, that can be ameliorated and refined by the interested reader in the future.

However, one may argue that attempting to build a complete synthesis system that implements not the simplest techniques, but those that produce best results, is an undertaking too large for a single person. For this reason, the synthesizer created for the purpose of this thesis, *Anthem*, will be open-sourced and made available to the online community, for free and with no restrictions. The goal is to make *Anthem* the go-to alternative to commercial music synthesizers, built by audio programming enthusiasts from all over the world, who will hopefully improve on the code base written for this thesis. The journey does not end here. It begins.

Appendices

Appendix A

Code Listings

```cpp
#include <cmath>

int main(int argc, char * argv[])
{
  // The sample rate, 48 kHz.
  const unsigned short samplerate = 48000;

  // The duration of the generated sine wave, in seconds.
  const unsigned long duration = 1;

  // The number of samples that will be generated, derived
  // from the number of samples per second (the sample rate)
  // and the number of seconds to be generated for.
  const unsigned long numberOfSamples = duration * samplerate;

  const double pi = 3.141592653589793;

  const double twoPi = 6.28318530717958;

  // The frequency of the sine wave
  double frequency = 1;

  // The phase counter. This variable can be seen as phi.
  double phase = 0;

  // The amount by which the phase is incremented for each
  // sample. Since one period of a sine wave has 2 pi radians,
  // dividing that value by the sample rate yields the amount
  // of radians by which the phase needs to be incremented to
  // reach a full 2 pi radians.
  double phaseIncrement = frequency * twoPi / samplerate;

  // The maximum amplitude of the signal, should not exceed 1.
  double maxAmplitude = 0.8;

  // The buffer in which the samples will be stored.
  double * buffer = new double[numberOfSamples];

  // For every sample.
  for (unsigned long n = 0; n < numberOfSamples; ++n)
  {
      // Calculate the sample.
      buffer[n] = maxAmplitude * sin(phase);

      // Increment the phase by the appropriate
      // amount of radians.
      phase += phaseIncrement;

      // Check if two pi have been reached and
      // reset if so.
      if (phase >= twoPi)
      {
          phase -= twoPi;
      }
  }

  // Further processing ...

  // Free the buffer memory.
  delete [] buffer;
}
```

Listing A.1: C++ implementation of a complete sine wave generator.

```
template <class PartItr>
double* additive(PartItr start,
                 PartItr end,
                 unsigned long length,
                 double masterAmp = 1,
                 bool sigmaAprox = false,
                 unsigned int bitWidth = 16)
{
    static const double pi = 3.141592653589793;

    static const double twoPi = 6.28318530717958;

    // calculate number of partials
    unsigned long partNum = end - start;

    double * buffer = new double [length];

    double * amp = new double [partNum];         // the amplitudes
    double * phase = new double [partNum];       // the current phase
    double * phaseIncr = new double [partNum];  // the phase increments

    // constant sigma constant part
    double sigmaK = pi / partNum;

    // variable part
    double sigmaV;

    // convert the bit number to decimal
    bitWidth = pow(2, bitWidth);

    // the fundamental increment of one period
    // in radians
    static double fundIncr = twoPi / length;

    // fill the arrays with the respective partial values
    for (unsigned long p = 0; start != end; ++p, ++start)
    {
        // initial phase
        phase[p] = start->phaseOffs;

        // fundIncr is two ÏĂ / tablelength
        phaseIncr[p] = fundIncr * start->num;

        // reduce amplitude if necessary
        amp[p] = start->amp * masterAmp;

        // apply sigma approximation conditionally
        if (sigmaAprox)
        {
            // following the formula
            sigmaV = sigmaK * start->num;

            amp[p] *= sin(sigmaV) / sigmaV;
        }
    }

    // fill the wavetable
    for (unsigned int n = 0; n < length; n++)
    {
        double value = 0;

        // do additive magic
        for (unsigned short p = 0; p < partNum; p++)
        {
            value += sin(phase[p]) * amp[p];

            phase[p] += phaseIncr[p];

            if (phase[p] >= twoPi)
            { phase[p] -= twoPi; }

            // round if necessary
```

```
73              if (bitWidth < 65536)
74              {
75                  Util::round(value, bitWidth);
76              }
77
78              buffer[n] = value;
79          }
80      }
81
82      delete [] phase;
83      delete [] phaseIncr;
84      delete [] amp;
85
86      return buffer;
87  }
```

Listing A.2: C++ program to produce one period of an additively synthesized complex waveform, given a start and end iterator to a container of partials, a buffer length, a maximum, "master", amplitude, a boolean whether or not to apply sigma approximation and lastly a maximum bit width parameter.

```
#include <cmath>
#include <stdexcept>

class EnvSeg
{

public:

    typedef unsigned long len_t;

    EnvSeg(double startLevel = 0,
           double endLevel = 0,
           len_t len = 0,
           double rate = 1)

    : startLevel_(startLevel), endLevel_(endLevel),
    rate_(rate), curr_(0),len_(len)

    {
        calcRange_();
        calcIncr_();
    }

    double tick()
    {
        // If the segment is still supposed to
        // tick after reaching the end amplitude
        // just return the end amplitude

        if (curr_ >= 1 || ! len_) return endLevel_;

        return range_ * pow(curr_,rate_) + startLevel_;
    }

    void update()
    {
        // Increment curr_
        curr_ += incr_;
    }

    void setLen(len_t sampleLen)
    {
        len_ = sampleLen;

        calcIncr_();
    }

    len_t getLen() const
    {
        return len_;
    }

    void setRate(double rate)
    {
        if (rate > 10 || rate < 0)
        { throw std::invalid_argument("Rate must be between 0 and 10"); }

        rate_ = rate;
    }

    double getRate() const
    {
        return rate_;
    }

    void setEndLevel(double lv)
    {
        if (lv > 1 || lv < 0)
        { throw std::invalid_argument("Level must be between 0 and 1"); }

        endLevel_ = lv;

```

```
73          calcRange_();
74      }
75
76      double getEndLevel() const
77      {
78          return endLevel_;
79      }
80
81      void setStartLevel(double lv)
82      {
83          if (lv > 1 || lv < 0)
84          { throw std::invalid_argument("Level must be between 0 and 1"); }
85
86          startLevel_ = lv;
87
88          calcRange_();
89      }
90
91      double getStartLevel() const
92      {
93          return startLevel_;
94      }
95
96      void reset()
97      {
98          curr_ = 0;
99      }
100
101 private:
102
103     /*! Calculates the amplitude range and assigns it to range_ */
104     void calcRange_()
105     {
106         // The range between start and end
107         range_ = endLevel_ - startLevel_;
108     }
109
110     /*! Calculates the increment for curr_ and assigns it to incr_ */
111     void calcIncr_()
112     {
113         incr_ = (len_) ? 1.0/len_ : 0;
114     }
115
116     /*! The rate determining the type (lin,log,exp) */
117     double rate_;
118
119     /*! Starting amplitude */
120     double startLevel_;
121
122     /*! End amplitude */
123     double endLevel_;
124
125     /*! Difference between end and start amplitude */
126     double range_;
127
128     /*! Current segment value */
129     double curr_;
130
131     /*! Increment value for curr_ */
132     double incr_;
133
134     /*! Length of segment in samples */
135     len_t len_;
136 };
```

Listing A.3: C++ implementation of single Envelope segments.

```
1  private:
2
3      struct ModItem;
4
5      typedef std::vector<ModItem> modVec;
6
7      // Not using iterators because they're invalidated when
8      // pushing back/erasing from modItems_ and not using
9      // pointers to ModItems because then it's difficult
10     // to interact with modItems_
11     typedef std::vector<modVec::size_type> indexVec;
12
13     typedef indexVec::iterator indexVecItr;
14
15     typedef indexVec::const_iterator indexVecItr_const;
16
17     /*! A ModItem contains a ModUnit*, its depth value as well as */
18     struct ModItem
19     {
20         ModItem(ModUnit* modUnit, double dpth = 1)
21         : mod(modUnit), depth(dpth), baseDepth(dpth)
22         { }
23
24         /*! The actual ModUnit pointer. */
25         ModUnit* mod;
26
27         /*! The current modulation depth. */
28         double depth;
29
30         /*! For sidechaining */
31         double baseDepth;
32
33         /*! Vector of indices of all masters in modItems_*/
34         indexVec masters;
35
36         /* Vector of indices of slaves of this ModItem in modItems_ */
37         indexVec slaves;
38     };
39
40     /*! Indices of all ModItems that are masters for sidechaining
41      and thus don't contribute to the ModDocks modulation value */
42     indexVec masterItems_;
43
44     /*! Pointer to all ModItems excluding sidechaining masters */
45     indexVec nonMasterItems_;
46
47     /*! All ModItems */
48     modVec modItems_;
49
50     /*! This is the base value that the modulation happens around */
51     double baseValue_;
52
53     /*! Lower boundary value to scale to when modulation trespasses it */
54     double lowerBoundary_;
55
56     /*! Higher boundary value to scale to when modulation trespasses it */
57     double higherBoundary_;
```

Listing A.4: Private members of the `ModDock` class.

```
double ModDock::modulate(double sample)
{
    // If ModDock is not in use, return original sample immediately
    if (! inUse()) return sample;

    // Sidechaining

    // For every non-master
    for (indexVecItr nonMasterItr = nonMasterItems_.begin(), nonMasterEnd =
    nonMasterItems_.end();
         nonMasterItr != nonMasterEnd;
         ++nonMasterItr)
    {
        // If it isn't a slave, nothing to do
        if (! isSlave(*nonMasterItr)) continue;

        ModItem& slave = modItems_[*nonMasterItr];

        // Set to zero initially
        slave.depth = 0;

        // Then sum up the depth from all masters
        for (indexVecItr_const masterItr = slave.masters.begin(), masterEnd =
    slave.masters.end();
             masterItr != masterEnd;
             ++masterItr)
        {
            ModItem& master = modItems_[*masterItr];

            // Using the baseDepth as the base value and the master's depth as
            // the depth for modulation and 1 as the maximum boundary
            slave.depth += master.mod->modulate(slave.baseDepth, master.depth, 1);
        }

        // Average the depth
        slave.depth /= slave.masters.size();
    }

    // Modulation

    double temp = 0;

    // Get modulation from all non-master items
    for(indexVecItr_const itr = nonMasterItems_.begin(), end = nonMasterItems_.end();
        itr != end;
        ++itr)
    {
        // Add to result so we can average later
        // Use the sample as base, the modItem's depth as depth and the
        // higherBoundary as maximum
        temp += modItems_[*itr].mod->modulate(sample,
                                              modItems_[*itr].depth,
                                              higherBoundary_);
    }

    // Average
    sample = temp / nonMasterItems_.size();

    // Boundary checking
    if (sample > higherBoundary_) { sample = higherBoundary_; }

    else if (sample < lowerBoundary_) { sample = lowerBoundary_; }

    return sample;
}
```

Listing A.5: Definition of the `modulate` method in the `ModDock` class.

```
double FM::modulate_(index_t carrier, double value)
{
    ops_[carrier]->modulateFrequency(value);

    return ops_[carrier]->tick();
}

double FM::add_(index_t carrier, double value)
{
    return ops_[carrier]->tick() + value;
}

double FM::tick()
{
    const double aTick = tickIfActive_(A);

    switch (alg_)
    {
        case 0:
            return modulate_(D, modulate_(C, modulate_(B, aTick)));

        case 1:
            return modulate_(D, modulate_(C, add_(B, aTick)));

        case 2:
            return modulate_(D, add_(C, modulate_(B, aTick)));

        case 3:
            return modulate_(D, modulate_(B, aTick) + modulate_(C, aTick));

        case 4:
        {
            double temp = modulate_(B, aTick);

            return modulate_(D, temp) + modulate_(C, temp);
        }

        case 5:
            return add_(D, modulate_(C, modulate_(B, aTick)));

        case 6:
        {
            double bTick = tickIfActive_(B);

            return modulate_(D, add_(C, aTick + bTick));
        }

        case 7:
        {
            double bTick = tickIfActive_(B);

            return modulate_(C, aTick) + modulate_(D, bTick);
        }

        case 8:
            return modulate_(D, aTick) + modulate_(C, aTick) + modulate_(B, aTick);

        case 9:
            return add_(D, add_(C, modulate_(B, aTick)));

        case 10:
            return add_(D, modulate_(C, aTick) + modulate_(B, aTick));

        case 11:
        default:
            return add_(D, add_(C, add_(B, aTick)));
    }
}
```

Listing A.6: Implementations of the various FM algorithms shown in Figure 6.4. This code excerpt is from the `FM` class.

```
#ifndef __Anthem__Wavefile__
#define __Anthem__Wavefile__

#include <fstream>
#include <string>
#include <deque>
#include <memory>

class Sample;

class Wavefile
{

public:

    Wavefile(const std::string& fname = std::string(),
             unsigned short channels = 2);

    Wavefile(const Wavefile& other);

    Wavefile& operator= (const Wavefile& other);

    void setChannels(unsigned short channels);

    void open(const std::string& fname);

    void close();

    void process(const Sample& sample);

    void write();

    void flush();

private:

    /*! Wavefile header */
    struct
    {
        uint8_t riffId[4]; // 'R' 'I' 'F' 'F'

        uint32_t riffSize; // chunks size in bytes

        uint8_t wavetype[4]; // 'W' 'A' 'V' 'E'

        uint8_t fmtId[4]; // 'f' 'm' 't' ''

        uint32_t fmtSize;

        uint16_t fmtCode; // 1 = pulse code modulation

        uint16_t channels;

        uint32_t samplerate;

        uint32_t byterate; // bytes per second

        uint16_t align; // bytes per sample * channel

        uint16_t bits; // one byte per channel, so 16 bits per sample

        uint8_t waveId[4]; // 'd' 'a' 't' 'a'

        uint32_t waveSize; // byte total

    } header_;

    /*! The sample buffer */
    std::deque<std::unique_ptr<Sample>> buffer_;

    /*! The file name */
    std::string fname_;
```

```
73
74      /*! Wavefile stream object */
75      std::ofstream file_;
76  };
77
78  #endif
```

Listing A.7: `Wavefile` header file.

```
1   #include "Wavefile.hpp"
2   #include "Global.hpp"
3   #include "Util.hpp"
4   #include "Sample.hpp"
5
6   #include <stdint.h>
7   #include <stdexcept>
8
9   Wavefile::Wavefile(const std::string& fname, unsigned short channels)
10  {
11      memcpy(header_.riffId, "RIFF", 4*sizeof(char));
12
13      memcpy(header_.wavetype, "WAVE", 4*sizeof(char));
14
15      memcpy(header_.fmtId, "fmt ", 4*sizeof(char));
16
17      header_.fmtSize = 16;
18
19      header_.fmtCode = 1;    // 1 = PCM
20
21      header_.channels = channels;    // 1 = mono, 2 = stereo
22
23      header_.samplerate = Global::samplerate;
24
25      header_.bits = 16;
26
27      header_.align = (channels * header_.bits) / 8;
28
29      header_.byterate = header_.samplerate * header_.align;
30
31      memcpy(header_.waveId, "data", 4*sizeof(char));
32
33      open(fname);
34  }
35
36  Wavefile::Wavefile(const Wavefile& other)
37  : fname_(other.fname_),
38    file_(other.fname_, std::ios::out | std::ios::binary | std::ios::trunc),
39    header_(other.header_)
40  {
41      // Copy buffer data
42      for(std::deque<std::unique_ptr<Sample>>::const_iterator itr = other.buffer_.begin(),
43          end = other.buffer_.begin();
44          itr != end;
45          ++itr)
46      {
47          buffer_.push_back(std::unique_ptr<Sample>(new Sample(*(*itr))));
48      }
49  }
50
51  Wavefile& Wavefile::operator= (const Wavefile& other)
52  {
53      if (this != &other)
54      {
55          fname_ = other.fname_;
56
57          file_.close();
58
59          file_.open(fname_, std::ios::out | std::ios::binary | std::ios::trunc);
60
61          header_ = other.header_;
62
63          // Copy buffer data
64          for(std::deque<std::unique_ptr<Sample>>::const_iterator itr = other.buffer_.begin(),
```

```
65              end = other.buffer_.begin();
66              itr != end;
67              ++itr)
68          {
69              buffer_.push_back(std::unique_ptr<Sample>(new Sample(*(*itr))));
70          }
71      }
72
73      return *this;
74  }
75
76  void Wavefile::setChannels(unsigned short channels)
77  { header_.channels = channels; }
78
79  void Wavefile::open(const std::string& fname)
80  {
81      fname_ = Util::checkFileName(fname, ".wav");
82
83      file_.open(fname_, std::ios::out | std::ios::binary | std::ios::trunc);
84
85      if (! file_)
86      { throw std::invalid_argument("Error opening file!"); }
87  }
88
89  void Wavefile::close()
90  {
91      file_.close();
92  }
93
94  void Wavefile::process(const Sample &sample)
95  {
96      buffer_.push_back(std::unique_ptr<Sample>(new Sample(sample)));
97  }
98
99  void Wavefile::flush()
100 {
101     buffer_.clear();
102 }
103
104 void Wavefile::write()
105 {
106     unsigned int totalSamples = static_cast<unsigned int>(buffer_.size());
107
108     header_.waveSize = totalSamples * header_.align;
109
110     header_.riffSize = header_.waveSize + sizeof(header_) - 8;
111
112     unsigned long twoTotalSamples = totalSamples * 2;
113
114     // because the current buffer is of type double
115     int16_t* outBuffer = new int16_t [twoTotalSamples];
116
117     for (unsigned long n = 0; n < twoTotalSamples;)
118     {
119         // Convert to 16 bit integer
120         (*buffer_.front()) *= 32767;
121
122         // Write to both channels (first channel 1, then channel 2)
123         outBuffer[n++] = buffer_.front()->left;
124         outBuffer[n++] = buffer_.front()->right;
125
126         buffer_.pop_front();
127     }
128
129     // Try writing the header_ and the data to file
130     if (! file_.write(reinterpret_cast<char*>(&header_), sizeof(header_)) ||
131         ! file_.write(reinterpret_cast<char*>(outBuffer), header_.waveSize))
132     { throw std::runtime_error("Error writing to file"); }
133 }
```

Listing A.8: `Wavefile` implementation file.

Bibliography

[1] Daniel R. Mitchell,
BasicSynth: Creating a Music Synthesizer in Software.
Publisher: Author.
1st Edition,
2008.

[2] Steven W. Smith,
The Scientist and Engineer's Guide to Digital Signal Processing.
California Technical Publishing,
San Diego, California,
2nd Edition,
1999.

[3] Ed: Catherine Soanes and Angus Stevenson,
Oxford Dictionary of English.
Oxford University Press,
Oxford,
2003.

[4] Robert Bristow-Johnson,
Cookbook formulae for audio EQ biquad filter coefficients.
`http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt`
Accessed: 4 January 2015.

[5] Phil Burk, Larry Polansky, Douglas Repetto, Mary Roberts and Dan Rockmore,
Music and Computers: A Historical and Theoretical Approach.
2011
`http://music.columbia.edu/cmc/MusicAndComputers/`
Accessed: 22 December 2014.

[6] Gordon Reid,
Synth Secrets, Part 12: An Introduction To Frequency Modulation.
`http://www.soundonsound.com/sos/apr00/articles/synthsecrets.htm`
Accessed: 30 December 2014.

[7] Gordon Reid,

Synth Secrets, Part 13: More On Frequency Modulation.

http://www.soundonsound.com/sos/may00/articles/synth.htm

Accessed: 30 December 2014.

[8] *Tutorial for Frequency Modulation Synthesis*.

http://www.sfu.ca/~truax/fmtut.html

Accessed: 30 December 2014.

[9] Justin Colletti,

The Science of Sample Rates (When Higher Is Better âĂŤ– And When It Isn't).

2013.
http://www.trustmeimascientist.com/2013/02/04/
the-science-of-sample-rates-when-higher-is-better-and-when-it-isnt/

Accessed: 17 December 2014.

[10] John D. Cutnell and Kenneth W. Johnson,

Physics.

Wiley,

New York,

4th Edition,

1998.

[11] Scott Wilson,

WAVE PCM soundfile format.

2003.

https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

Accessed: 2 January 2015.

[12] Dominique Vandenneucker,

MIDI Tutorial.

2012.

http://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%
20programmers.html

Accessed: 2 January 2015.

[13] User *JCPedroza*,

What is the level of frequency modulation of many synthesizers?.

http://sound.stackexchange.com/questions/31709/
what-is-the-level-of-frequency-modulation-of-many-synthesizers

Accessed: 3 January 2015.

[14] Electronic Music Wiki,
DX7.
http://electronicmusic.wikia.com/wiki/DX7
Accessed: 4 January 2015.

[15] Propellerhead,
Malström Graintable Synthesizer.
https://www.propellerheads.se/products/reason/instruments/malstrom/
Accessed: 4 January 2015.

List of Figures

Declaration Of Authorship

I, Peter Goldsborough, declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research. I confirm that all sources I consulted are listed in the bibliography.

Villach, January 12, 2015,

Peter Goldsborough