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 25, 2015

Abstract

The digital revolution has not only left its traces in the world of medicine, aeronautics or telecommunications, but also in the realm of electronic music production. While physical music synthesizers once produced sound through analog circuitry, they can now be implemented virtually — in software — using computer programming languages such as C++. This thesis aims to describe how the creation of such a software synthesizer can be achieved. The first chapter introduces rudimentary concepts of digital audio processing and explains how digital sound differs from its analog counterpart. Subsequently, chapter two discusses the generation of sound in software — using either mathematical calculation or Additive Synthesis, outlines how digital waveforms can be efficiently stored in a Wavetable and, lastly, touches upon the concept of Noise. Chapter three investigates how modulation sources such as Audio Envelopes or Low Frequency Oscillators can be used to modify amplitude, timbre and other properties of an audio signal. It also proposes a means of flexible modulation of a variable number of parameters in a synthesis system by an equally unfixed number of modulation sources. The last chapter explains how Frequency Modulation Synthesis can be used to synthesize audio signals and thereby produce entirely new, electronic sounds. Embedded into all chapters are numerous C++ computer programs as well as images, diagrams and charts to practically illustrate the theoretical concepts of digital audio processing at hand.

Acknowledgements

I want to thank Professor Martin Kastner for supervising my thesis. His advice on how to structure and plan this paper 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 25, 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*[1] series 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 for 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 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 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)

[1]The Moog series of synthesizers were some of the first commercial analog synthesizers, designed by Dr. Robert Moog, founder of the *Moog Music* company, in the mid 1960s. Source: `http://en.wikipedia.org/wiki/Moog_synthesizer`, accessed 23 January 2015. Homepage: `http://www.moogmusic.com`, accessed 23 January 2015.

This thesis is intended to discuss the most essential 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[2], 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 relatively trivial task.

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 how computer music is generated, modulated and finally synthesized. 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.

The tools used to create the aforementioned images and diagrams include the *GeoGebra*[3] function plotting program for any figures depicting mathematical coordinate systems; *Audacity*[4], a software program for sound manipulation and visualization, for time or frequency domain representations of sound and, lastly, the web application *draw.io*[5] for any box diagrams or flow charts. Moreover, it may be of interest that this thesis was written with the LaTeX document preparation system [6].

Finally, 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.

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

[3] GeoGebra homepage: `http://www.geogebra.org`, accessed 23 January 2015.

[4] Audacity homepage: `http://audacity.sourceforge.net`, accessed 23 January 2015.

[5] draw.io homepage: `https://www.draw.io`, accessed 23 January 2015.

[6] LaTeX homepage: `http://www.latex-project.org`, accessed 23 January 2015.

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.

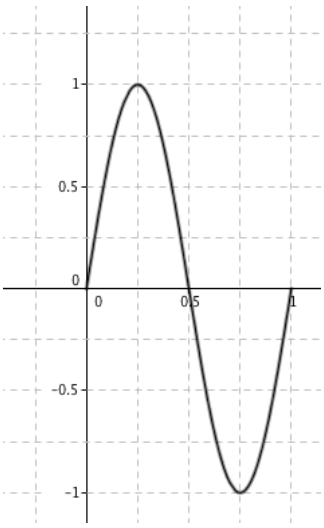

Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

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)

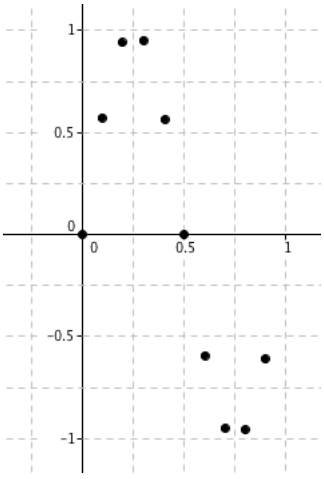

Figure 1.2: The discrete representation of a typical sine wave.

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). Common values for audio sample rates are 44.1 kHz, a frequency originally chosen by Sony in 1979 that is still used for Compact Discs (CDs), and 48 kHz, the standard audio sample rate used today. (Mitchell, 2008, p. 18) (Colletti, 2013) Equation 1.1 shows that 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 periodically taken from a continuous signal. The inverse of the sampling interval again yields the sample rate, as can be seen in Equation 1.2.

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

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

Nyquist Limit

The sample rate also determines the range of frequencies representable by a digital audio system. The reason for this is that only frequencies that are less than or equal to one half of the sample 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 higher than 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, 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, in this case ca. 20 Khz. (Cutnell & Johnson, 1998, p. 466)

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)

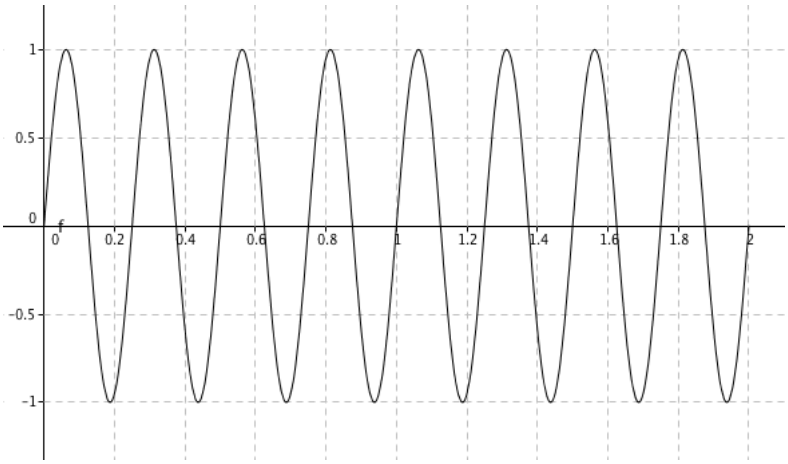

Figure 1.3: A sinusoid with a frequency of 4 Hz.

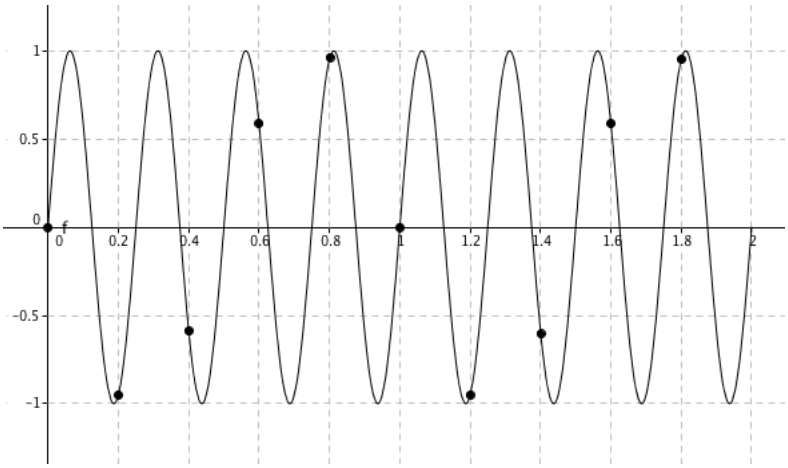

Figure 1.4: A sinusoid with a frequency of 4 Hz, sampled at a sample rate of 5 Hz. According to the Nyquist Theorem, this signal is sampled improperly, as the frequency of the continuous signal is not less than or equal to one half of the sample rate, in this case 2.5 Hz.

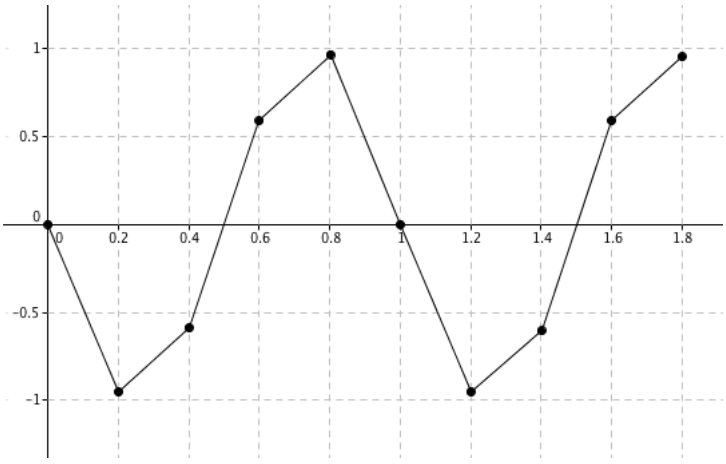

Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper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

Chapter 2

Generating Sound

This chapter will outline how sound waveforms can be generated in theory and implemented in practice, using the C++ programming language. It will also examine how *Wavetables* can be used to efficiently store and process these waveforms in a computer program. Lastly, a very special form of sound — *Noise* — will be discussed.

2.1 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. For future reference, the value of $2\pi f$ is called the *angular frequency* and is usually assigned the variable ω. 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. The program from Listing A.1 could be modified to compute a cosine wave by changing line 22 from `double phase = 0;` to `double phase = pi/2.0;`.

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

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

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. 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 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.

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. (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 2.1 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.

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

Equation 2.14: Simplification 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 2.1: C++ code to represent a single partial in a Fourier Series.

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 Additive Synthesis.

Square Waves

When speaking of Additive Synthesis, a square wave is the result of summing all odd-numbered partials (3^{rd}, 5^{th}, 7^{th} etc.) at a respective amplitude equal to the reciprocal of their partial number (3^{-1}, 5^{-1}, 7^{-1} 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 2.3, 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 2.2 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.

$$f(t) = \sum_{n=1}^{N} \frac{1}{2n-1} \sin(\omega(2n-1)t) \tag{2.3}$$

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 2.4 gives a mathematical definition for a sawtooth wave, Figure 2.2 displays sawtooth functions with various partial numbers and Table 2.3 shows C++ code to generate such functions.

$$f(t) = \sum_{n=1}^{N} \frac{1}{n} \sin(\omega n t) \tag{2.4}$$

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, n^{-1}, but now of the partial number squared: 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 2.5 or, more concisely, in Equation 2.6. Figure 2.3 displays such a triangle wave with various partial numbers and Table 2.4 implements C++ code to compute a triangle wave.

$$f(t) = \sum_{n=1}^{\frac{N}{2}} \frac{1}{(4n-3)^2} \sin(\omega(4n-3)t) - \frac{1}{(4n-1)^2} \sin(\omega(4n-1)t) \tag{2.5}$$

$$f(t) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)^2} \sin(\omega(2n+1)t) \tag{2.6}$$

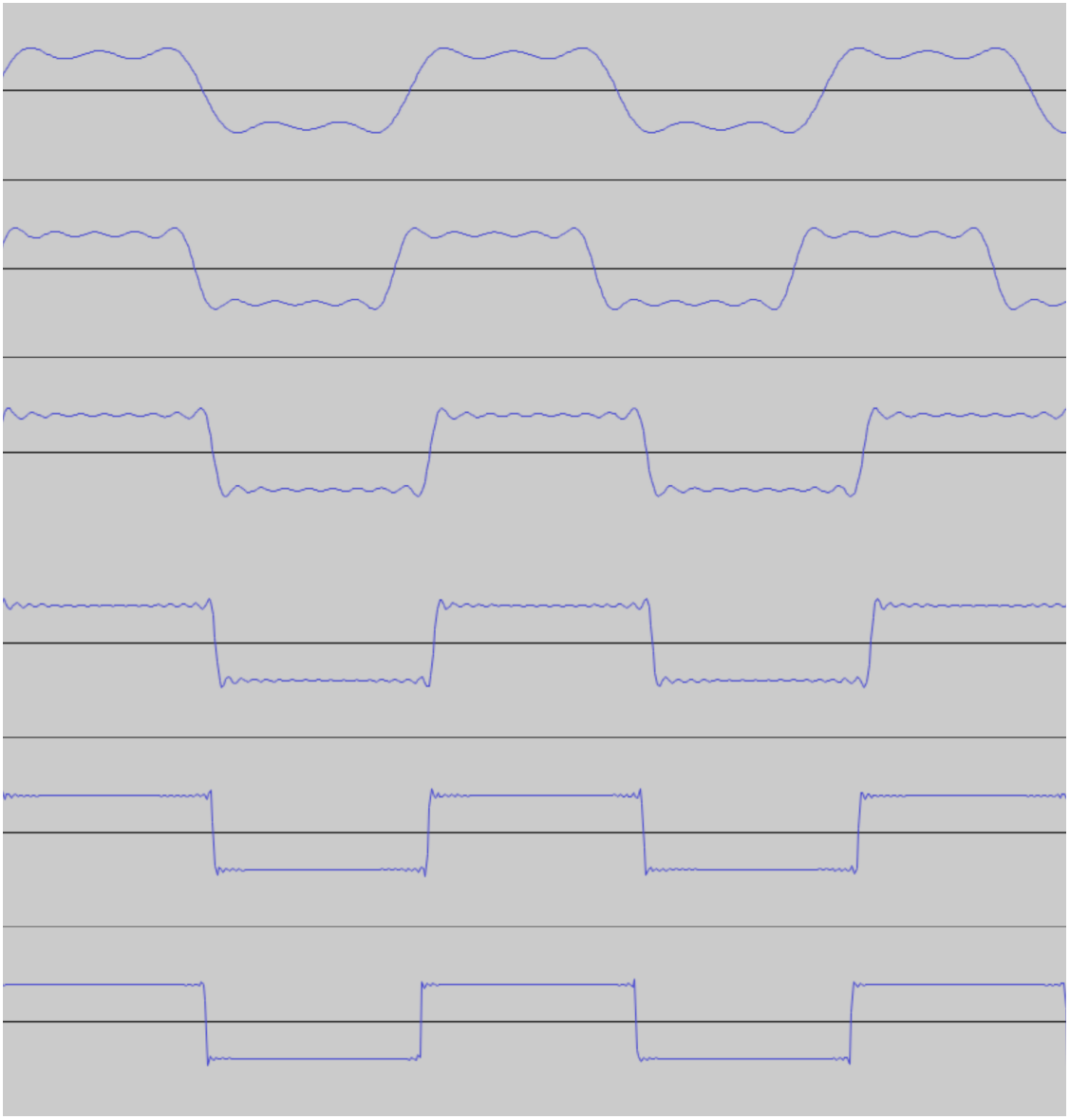

Figure 2.1: Square waves with 2, 4, 8, 16, 32 and 64 partials.

```
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 2.2: C++ code for a square wave with 64 partials.

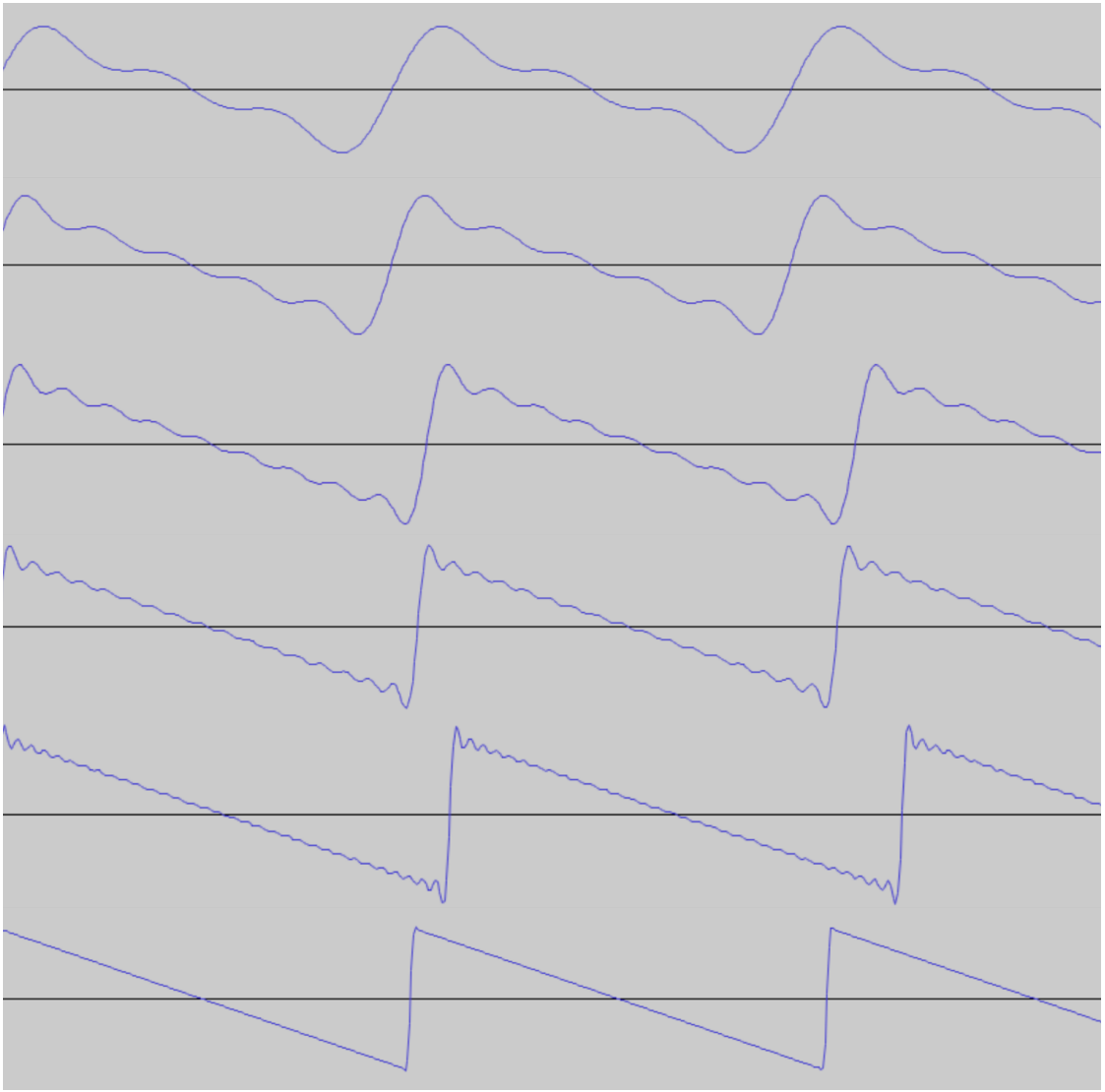

Figure 2.2: Sawtooth waves with 2, 4, 8, 16, 32 and 64 partials.

```
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 2.3: C++ code for a sawtooth wave with 64 partials.

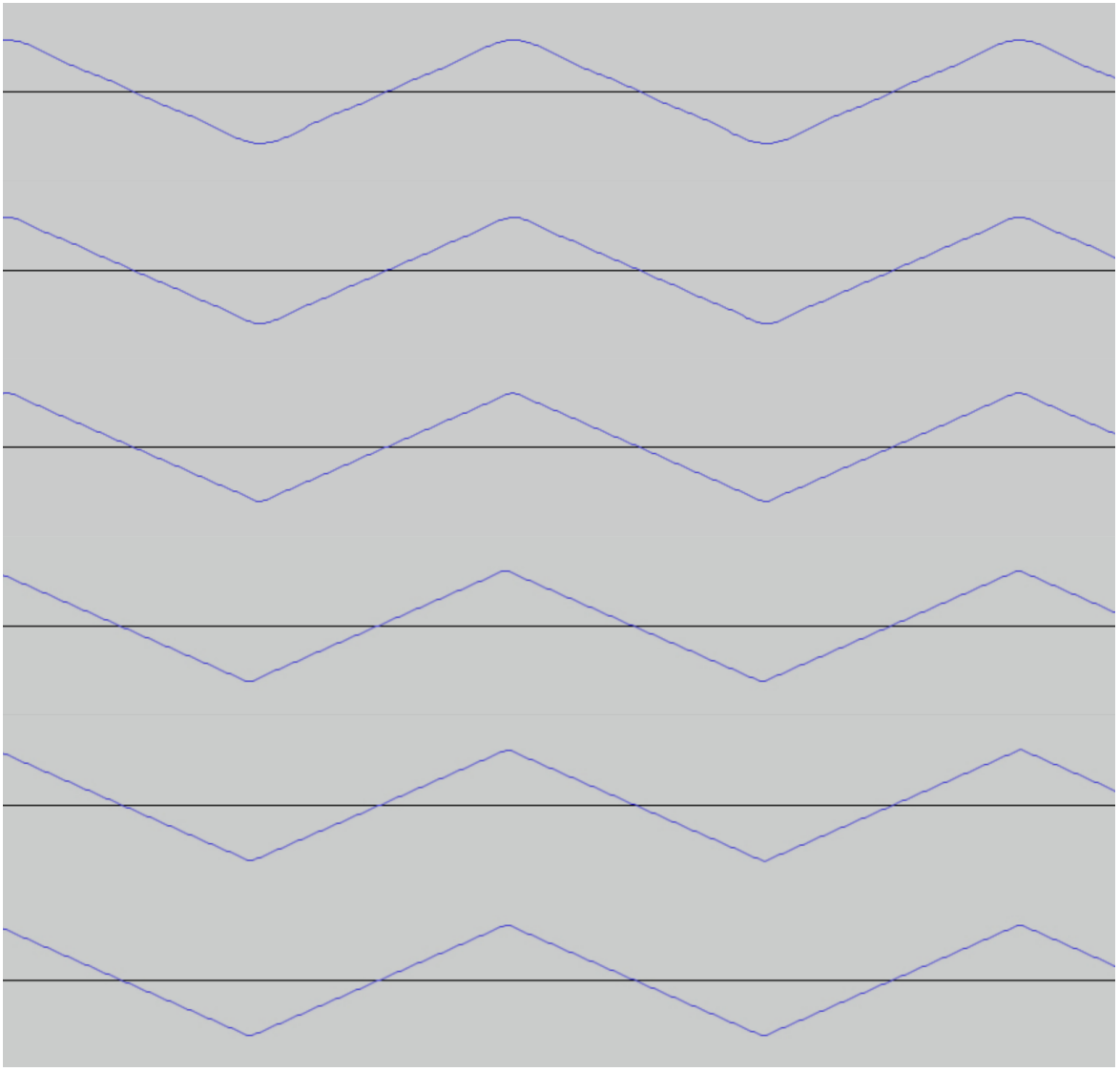

Figure 2.3: Triangle waves with 2, 4, 8, 16, 32 and 64 partials. Note that already 2 partials produce a very good approximation of a triangle wave.

```
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 2.4: C++ code for a triangle wave with 64 partials.

2.2 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 is calculated once and then written into memory — a *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.

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 2.7, where L is the Wavetable length and the f_s the sample rate. To alter the frequency f of the played-back waveform, Equation 2.8 can be used to calculate the appropriate increment value i.

$$i_{fund} = \frac{L}{f_s} \tag{2.7}$$

$$i = i_{fund} \cdot f = \frac{L}{f_s} \cdot f \tag{2.8}$$

Interpolation

For almost all configurations of frequencies, table lengths and sample rates, the table index i produced by Equation 2.8 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 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 2.5 displays the calculation of a sample by means of interpolation in pseudo-code and Table 2.6 in C++. (based on pseudo-code, Mitchell, 2008, p. 85)

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

Table 2.5: An interpolation 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 + ((difference between value2 and value1) * fractional part)
    return value1 + ((value2 - value1) * fractional);
}
```

Table 2.6: Full C++ template function to interpolate a value from a table, given a fractional index.

Table Length

The length of the Wavetable must be chosen carefully and should consider both memory efficiency and waveform resolution. An equation to calculate the size of a single wavetable in Kilobytes is given by Equation 2.9, 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`.

$$Size = \frac{L \cdot N}{1024} \tag{2.9}$$

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 2.4. 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.

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

One important fact to mention, also discussed by Jari Kleimola, is that because the interpolation algorithm from Table 2.6 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 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.

File Format

For maximum efficiency, the Wavetables are not created at program startup but read from a binary file. To prevent the synthesizer program from accidentally reading faulty files, some simple identification string must be added to the file. Therefore, Wavetable files contain a 6-byte ID string equal to the name of the synthesizer, *Anthem*, after which the 32 KB of Wavetable data follow. Additionally, Wavetable files end with a `.wavetable` file extension. Table 2.7 displays a function to read such a Wavetable file and Figure 2.5 shows the first few bytes of a Wavetable file when opened as plain-text. The total size of a Wavetable file is exactly 32774 bytes, 32768 bytes (32KB) of Wavetable data and 6 bytes for the ID string.

```
#include <fstream>
#include <stdexcept>

double* readWavetable(const std::string &fname)
{
  std::ifstream file(fname);

  if (! file.is_open())
  { throw std::runtime_error("Could not find wavetable file: " + fname); }

  if (! file.good())
  { throw std::runtime_error("Error opening wavetable: " + fname); }

  char signature[6];

  file.read(signature, 6);

  if (strncmp(signature, "ANTHEM", 6))
  { throw std::runtime_error("Invalid signature for Anthem file!"); }

  int len = 4096;
  int size = len * sizeof(double);

  double * wt = new double [len];

  file.read(reinterpret_cast<char*>(wt), size);

  return wt;
}
```

Table 2.7: C++ code to read a Wavetable file.

```
ANTHEM{†Åå#Y?ö`)¯ä#i?'œ Ã•/r?·zO5É#y?÷?Ëº\l?À°≠µò/Ç?ìµ7$˘Ö?±@ˆ-d#â?õÆÕVƒGå?¯£" lè?‰¯≠ä;Hë?
BtoYd/í?®ïâ?älî?áqœ¸¨¸ï?ÀòòYÃêó?g"wË"ô?Î¥ÅË˘¥ö?êÆ†Gú?cñÙ°"Ÿù?®VK º-kü?Œä'–ô~†?µk:öG°?¶4µ¯ó
¢?ã5fiÎíŸ¢?b,Ç¯ä¢£?
```

Figure 2.5: The first few bytes of a Wavetable file.

2.3 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, while others find noise relaxing and even listen to it while studying. (Mitchell, 2008, p. 76) Unlike all audio signals presented so far, noise cannot[1] 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* [2] of the sound produced, as well as, scientifically speaking, the spectral power density and frequency content of the signal.

White noise is the most frequently encountered 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 2.6 shows a typical white noise signal in the time domain, Figure 2.7 gives a close-up view of Figure 2.6, Figure 2.8 displays the frequency spectrum of a white noise signal and Table 2.8 shows the implementation of a simple C++ class to produce white noise.

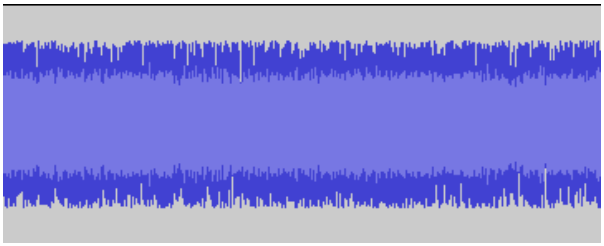

Figure 2.6: A typical white noise signal.

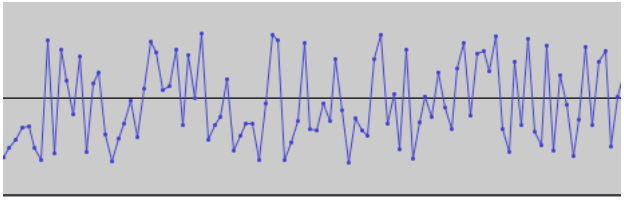

Figure 2.7: A close-up view of Figure 2.6. This Figure shows nicely how individual sample values are completely random and independent from each other.

[1]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.

[2]Timbre is a rather vague term used by musicians and audio engineers to describe the properties, such as pitch, tone or 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".

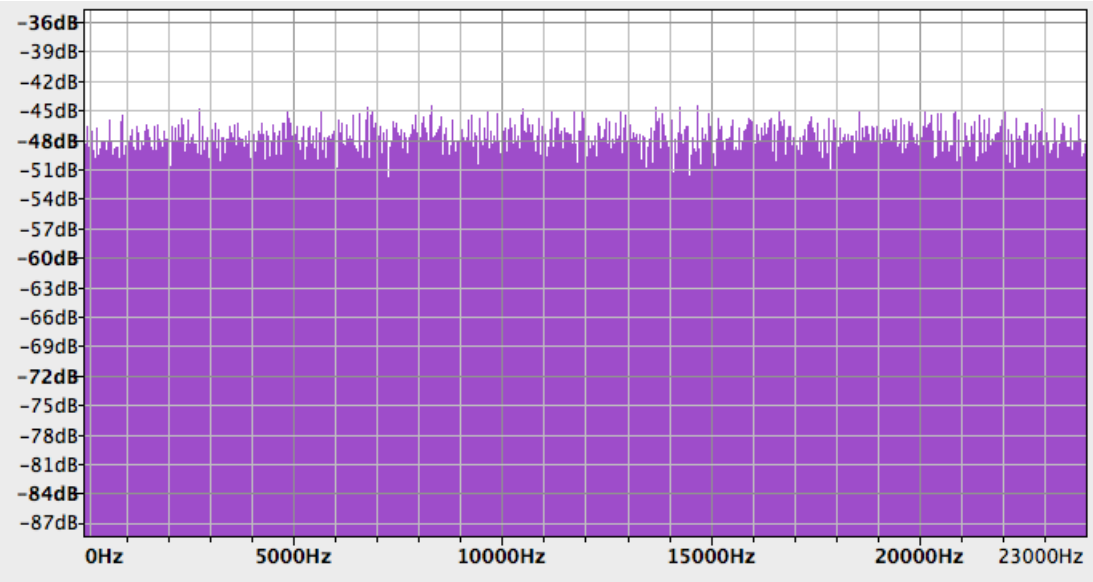

Figure 2.8: The signal from Figures 2.6 and 2.7 in the frequency domain. This frequency spectrum analysis proves the fact that white noise has a "flat" frequency spectrum, meaning that all frequencies are distributed uniformly and at (approximately) equal intensity.

```
#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 2.8: 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 ability to *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 or the influence of air friction. 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 referred to as an *Envelope*.

ADSR

A concept commonly 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 while the key of a synthesizer is still being pressed, the last segment type, a *Release* segment, refers to the change in loudness once the key has been 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.

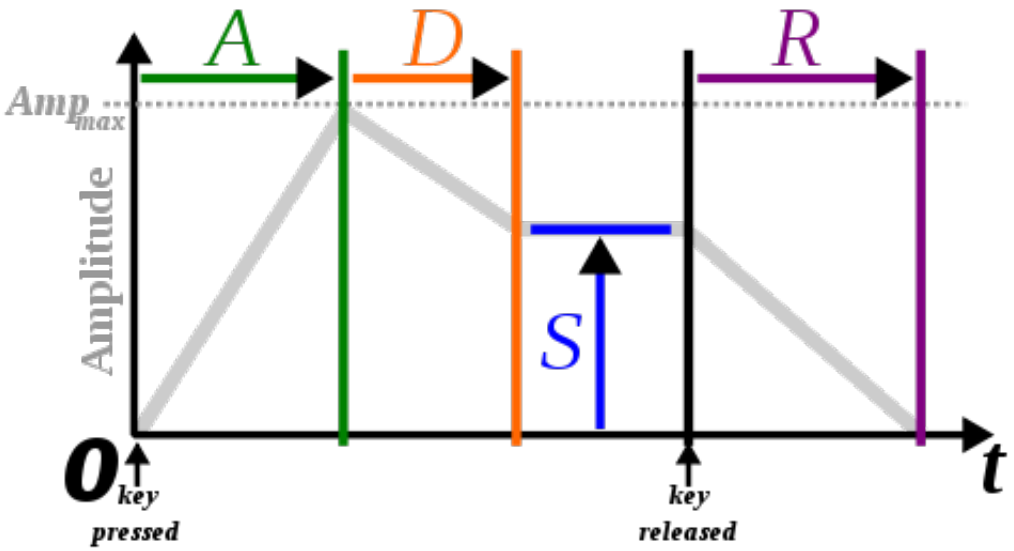

Figure 3.1: An Envelope with an Attack, a Decay, a Sustain and finally a Release segment. Source: `http://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/ADSR_parameter.svg/500px-ADSR_parameter.svg.png`, accessed 23 January 2015.

Figure 3.2: A 440 Hz sine wave.

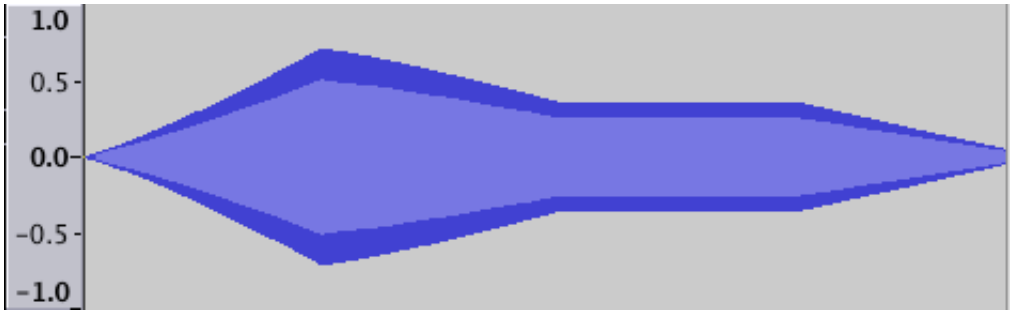

Figure 3.3: The same signal from Figure 3.2 with an ADSR Envelope overlayed on it.

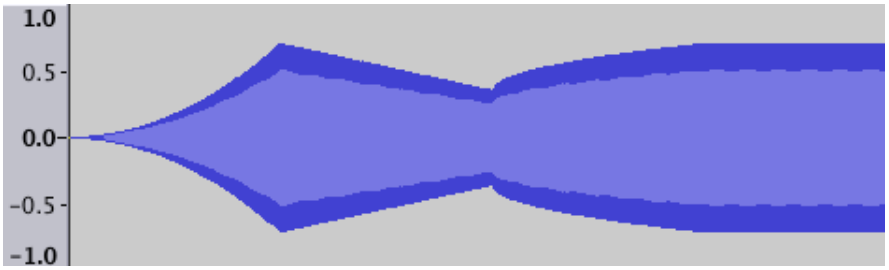

Figure 3.4: An envelope where r is equal to 2, then to 1, then to 0.5.

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 0.5 (a square root function). A C++ class for single Envelope segments is shown in Listing A.3.

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

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).

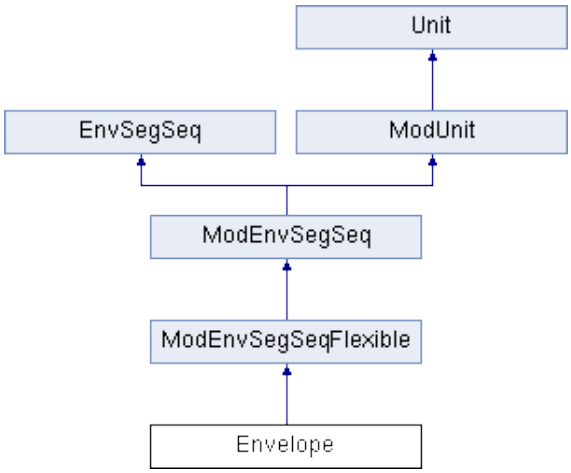

Figure 3.5: The inheritance diagram for the `Envelope` class. The `Unit` and `ModUnit` classes are two abstract classes that will be discussed in later parts of this thesis.

3.2 Low Frequency Oscillators

A Low Frequency Oscillator (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 3.7 shows a 440 Hz sine wave, Figure 3.8 displays the same sine wave now modulated by a 2 Hz LFO and in Figure 3.9 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.

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

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: `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 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). 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 3.6.

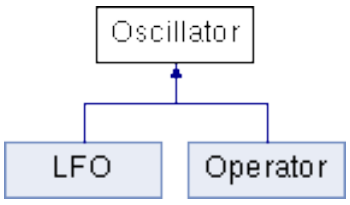

Figure 3.6: Inheritance diagram showing the relationship between an `LFO`, an `Operator` and their base class, the `Oscillator` class.

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

Figure 3.7: A 440 Hz sine wave.

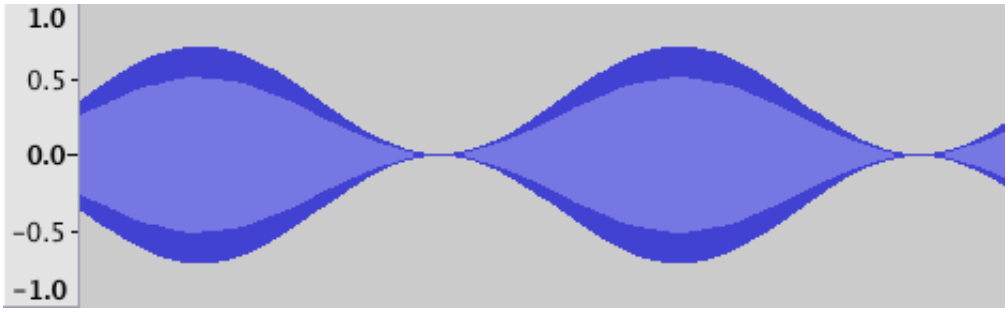

Figure 3.8: The sine wave from Figure 3.7 modulated by an LFO with a frequency of 2 Hertz. Note how the maximum amplitude of the underlying signal now follows the waveform of a sine wave. Because the LFO's amplitude is the same as that of the oscillator (the "carrier" wave), the loudness is twice as high at its maximum, and zero at its minimum.

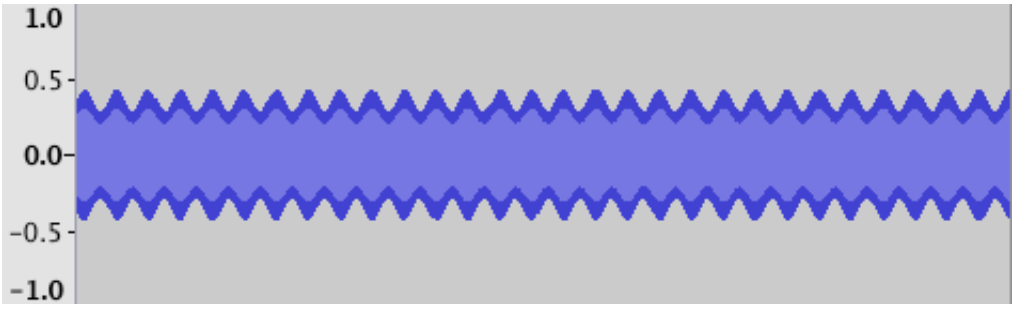

Figure 3.9: The sine wave from Figure 3.7 modulated by an LFO with a frequency of 20 Hertz. This signal is said to have a *Vibrato* sound to it.

3.3 The ModDock System

The majority of digital synthesizers, such as Propellerhead's[2] *Thor* or Native Instruments'[3] *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.

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 *sidechaining* 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 sidechain 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.

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 3.10 displays the inheritance diagram for the Unit class.

[2] "Propellerhead Software is a music software company, based in Stockholm, Sweden, and founded in 1994." Source: `http://en.wikipedia.org/wiki/Propellerhead_Software`, accessed 23 January 2015. Homepage: `https://www.propellerheads.se`, accessed 23 January 2015.

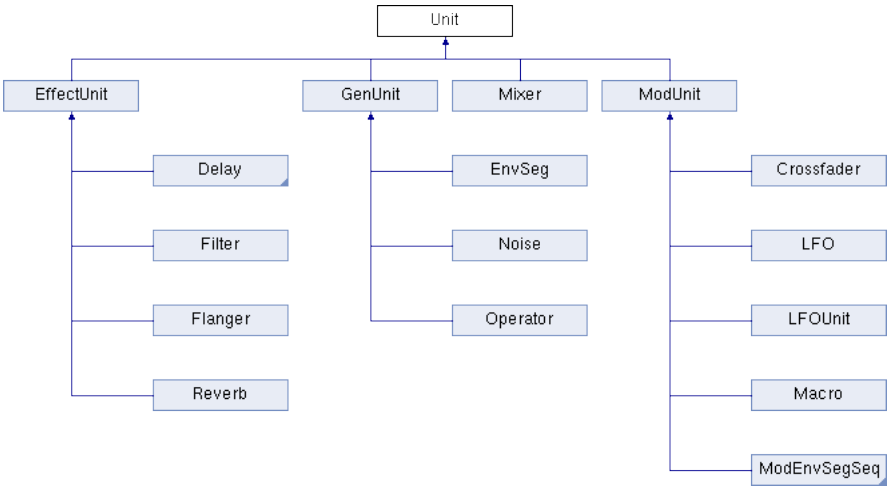

Figure 3.10: The inheritance diagram of the Unit class. GenUnits are Units that generate samples. EffectUnits apply some effect to a sample, e.g. a sample delay.

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 3.2 shows the definition of the `modulate` method in the LFO and Table 3.3 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 3.2: The implementation of the `modulate` method for LFOs.

[3]"Native Instruments is a technology company that develops software and hardware for music production and DJ-ing", based out of Berlin, Germany. Source: `http://en.wikipedia.org/wiki/Native_Instruments`, accessed 23 January 2015.. Homepage: `http://www.native-instruments.com/en/`, accessed 23 January 2015.

[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 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 3.3: 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 3.4 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 3.4

Sidechaining

One of the most interesting and equally complicated tasks encountered when creating the ModDock system was the implementation of sidechaining. Sidechaining 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 sidechains another ModUnit is termed a *Master* and the ModUnit being sidechained 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 sidechaining 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 before, 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 3.11 depicts these relationships in a flow-chart. Figure 3.12 shows a scan of early sketches created while implementing sidechaining.

What Figure 3.12 also displays is that there are two possible ways to implement sidechaining. The first method, which was ultimately not chosen, is to give each ModUnit in the ModDock its own ModDock where Masters can 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 sidechaining 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 sidechain 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-sidechain" 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 base value of the ModDock, in the sense that it 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.

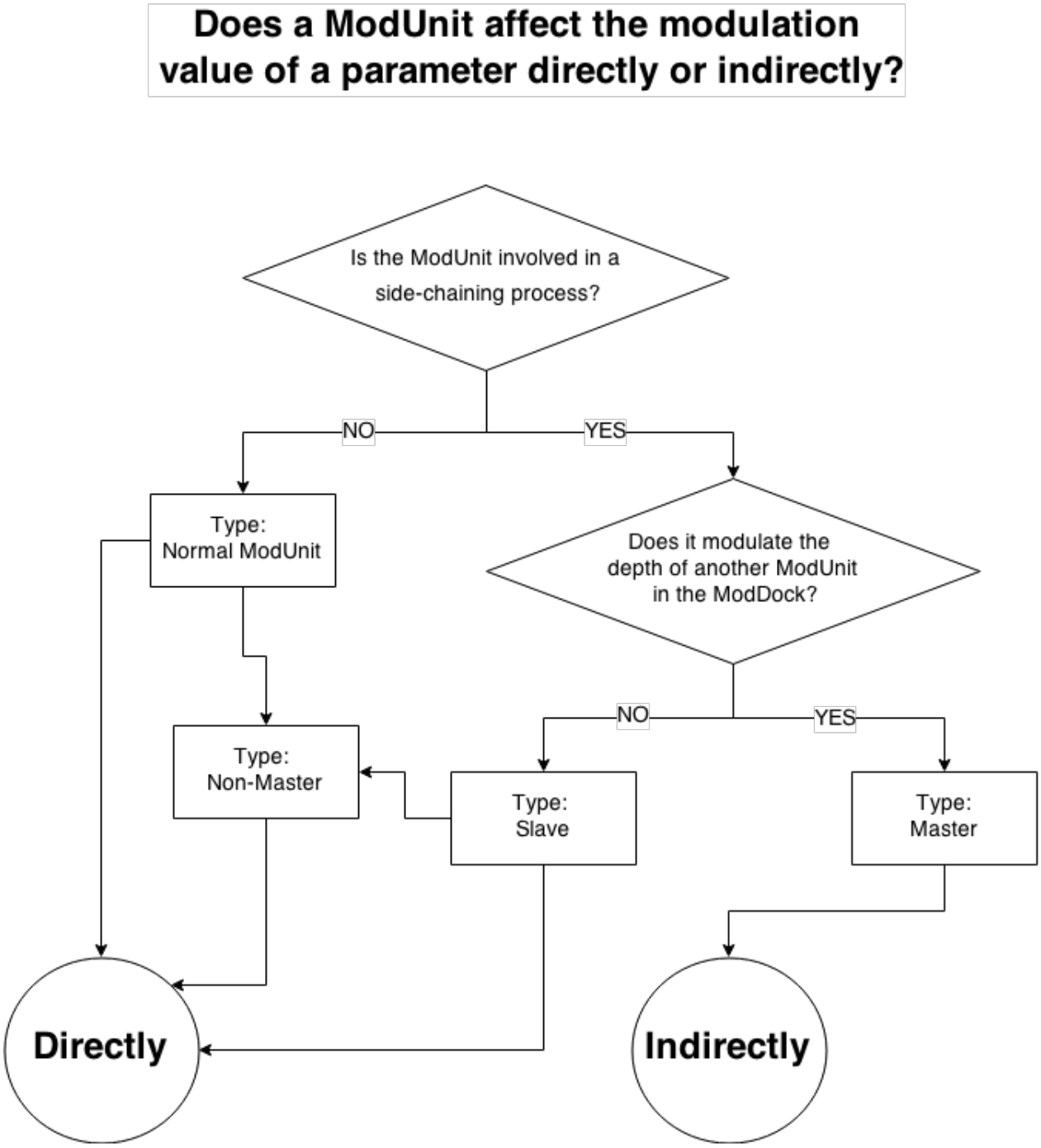
Does a ModUnit affect the modulation value of a parameter directly or indirectly?
Is the ModUnit involved in a side-chaining process?
NO
YES
Type: Normal ModUnit
Does it modulate the depth of another ModUnit in the ModDock?
NO
YES
Type: Non-Master
Type: Slave
Type: Master
Directly
Indirectly

Figure 3.11

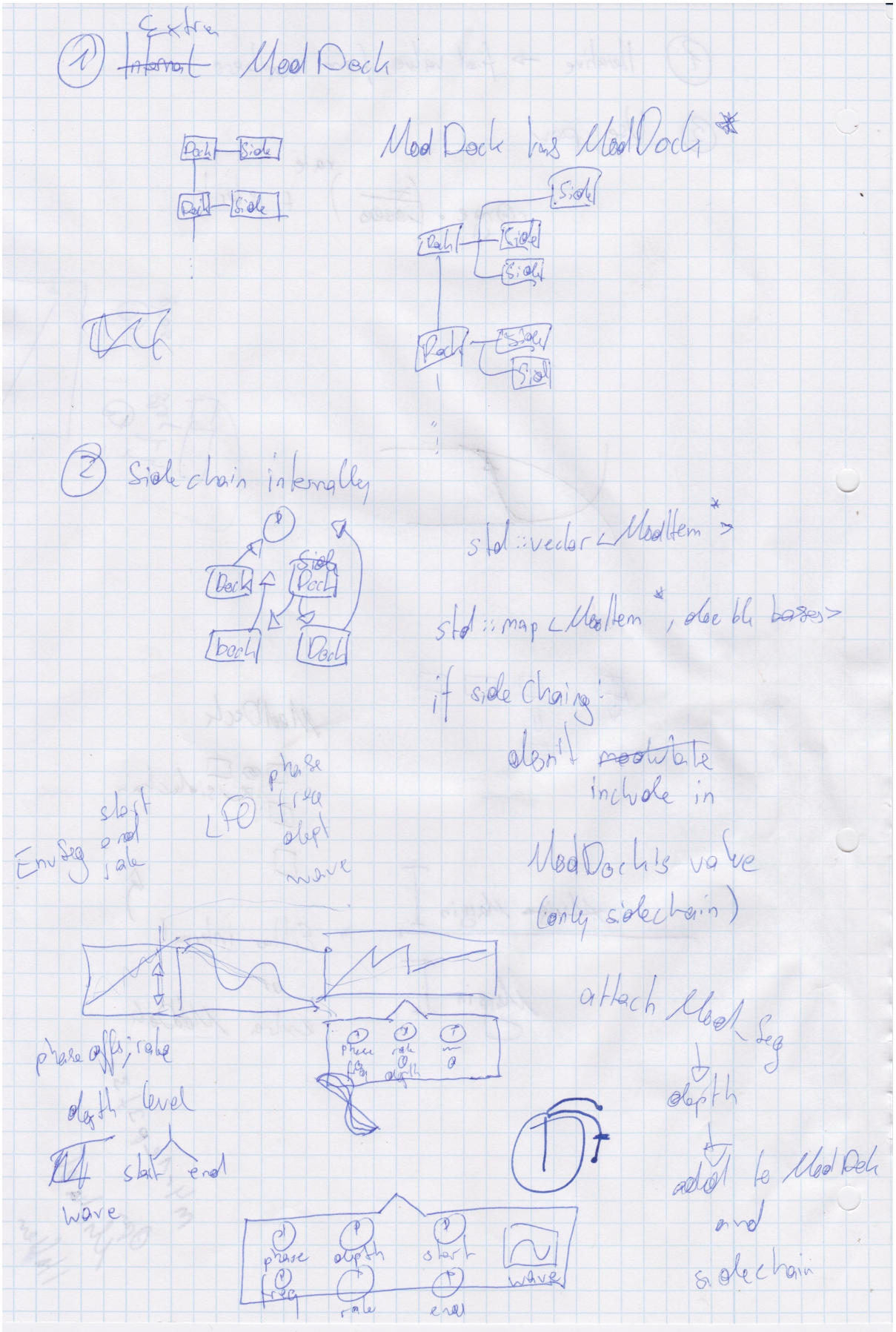
ModDock has ModDock *
Side chain internally
std::vector<ModItem *>
std::map<ModItem *, double> bases>
if side chain:
don't modulate
include in
ModDock's value
(only sidechain)
attach ModLfo
depth
add to ModDock
and
sidechain
EnvSeg
start
end
rate
LFO
phase
freq
depth
wave
phase offs; rate
depth level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

Figure 3.12

Chapter 4

Synthesizing Sound

The most important feature of a synthesizer is its ability 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 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. Common methods of synthesis are Additive Synthesis, Granular Synthesis, Amplitude Modulation Synthesis and Frequency Modulation Synthesis. This chapter aims to examine and describe the latter.

Frequency Modulation Synthesis

In Frequency Modulation (FM) Synthesis, one signal, termed the *modulator*, is used to modulate the frequency of another signal — the *carrier*. 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 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 4.3 gives a full mathematical definition for Frequency Modulation (Synthesis). What Equation 4.3 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 4.1 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.

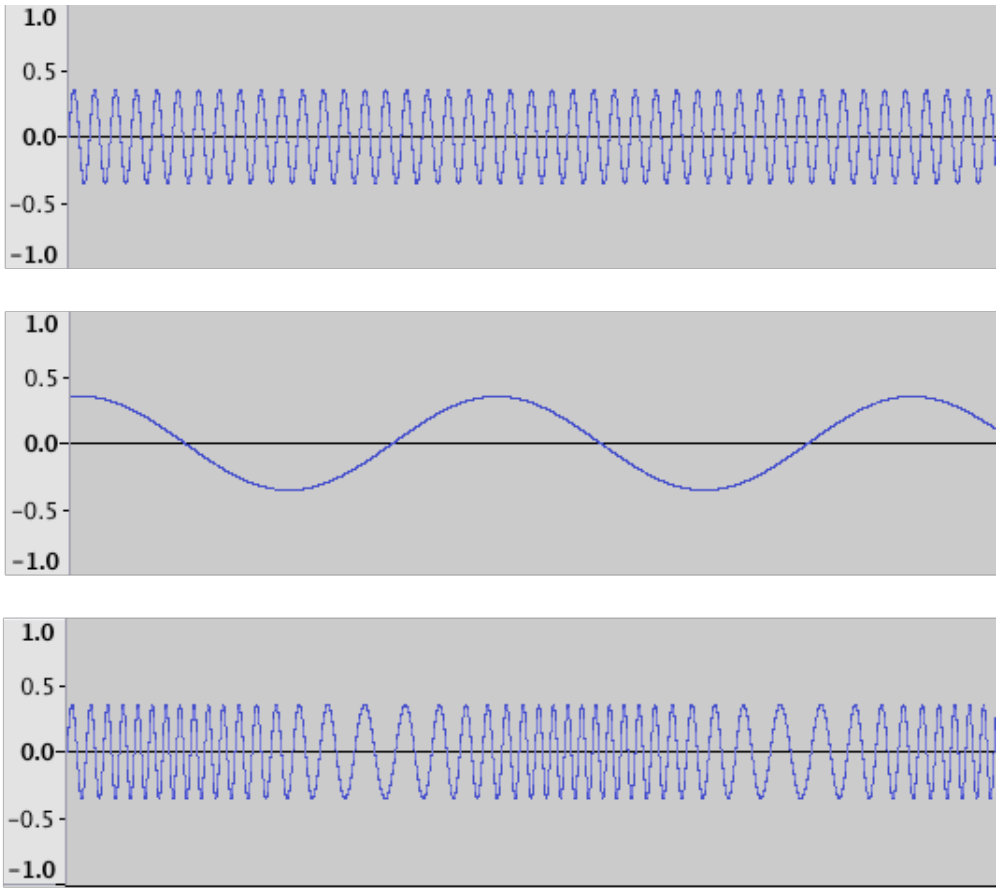

Figure 4.1: The first figure shows the carrier signal $c(t)$ with its frequency f_c equal to 100 Hz. The signal beneath the carrier is that of the modulator, $m(t)$, which has a frequency f_m of 5 Hz. When the modulator amplitude A_m is increased to 50 (instead of ~ 0.4, as shown here) and is used to modulate the frequency of the carrier, $f(t)$, shown in the last figure, is produced.

$$c(t) = A_c \cdot \sin(\omega_c t + \phi_c) \qquad (4.1) \qquad m(t) = A_m \cdot \sin(\omega_m t + \phi_m) \qquad (4.2)$$

$$f(t) = A_c \cdot \sin((\omega_c + m(t))t + \phi_c) = A_c \cdot \sin((\omega_c + A_m \cdot \sin(\omega_m t + \phi_m))t + \phi_c) \qquad (4.3)$$

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.

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 4.4 gives all the relative frequency values of the theoretically infinite set of sidebands. Equation 4.5 describes how to calculate ω_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.

$$\omega_n = C \pm n \cdot M, \text{where } n \in [0; \infty[\tag{4.4}$$

$$\omega_n = \omega_c \pm n \cdot \omega_m, \text{where } n \in [0; \infty[\tag{4.5}$$

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 4.6 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.

$$A \cdot \sin(-\omega t + \phi) = -A \cdot \sin(\omega t + \phi) = A \cdot \sin(\omega t + \phi - \pi) \tag{4.6}$$

Figure 4.2 shows the frequency spectrum of a carrier signal with a frequency f_c 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.

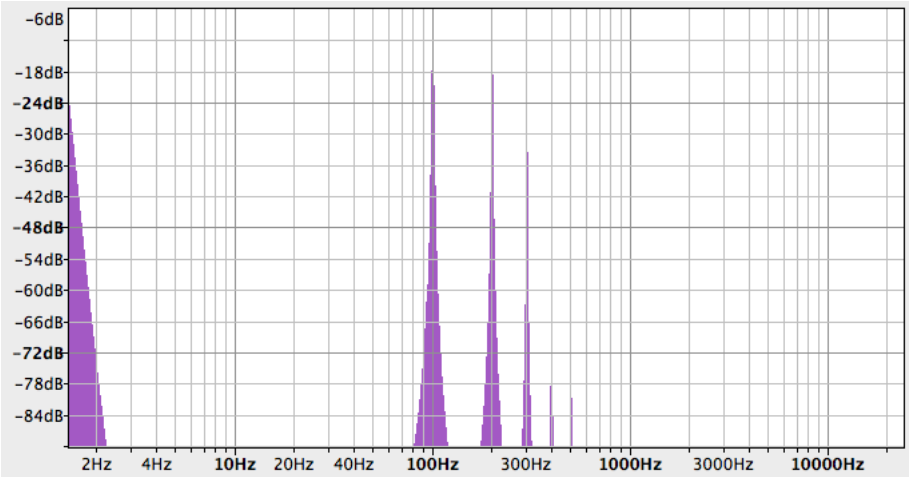

Figure 4.2: The frequency spectrum of a $C : M$ ratio of $2 : 1$, where the carrier frequency f_c is equal to 200 and the modulator frequency f_m to 100 Hertz.

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 4.7. Because the variation in carrier frequency, Δf_c, depends directly on the amplitude of the modulator, A_m, Equation 4.7 can be re-written as Equation 4.8.

$$\beta = \frac{\Delta f_c}{f_m} \tag{4.7}$$

$$\beta = \frac{\Delta f_c}{f_m} = \frac{A_m}{f_m} \tag{4.8}$$

The index of modulation can be used to determine the amplitude of individual sidebands, if input into the Bessel Function, shown in Equation 4.9, where n is the sideband to calculate the amplitude for. Table 4.1 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.

$$J(n, \beta) = \sum_{k=0}^{\infty} \frac{(-1)^k \cdot (\frac{\beta}{2})^{n+2k}}{(n+k)! \cdot k!} \tag{4.9}$$

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 *Carson Bandwidth Rule*, given in Equation 4.10.

$$B \approx 2(\Delta f_c + f_m) = 2(A_m + f_m) = 2f_m(1 + \beta) \tag{4.10}$$

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 4.1

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 *FM Algorithm*. The number of possible FM 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 FM Algorithms for these four Operators are shown in Figure 4.3.

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 4.2 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 FM Algorithms shown in Figure 4.3.

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

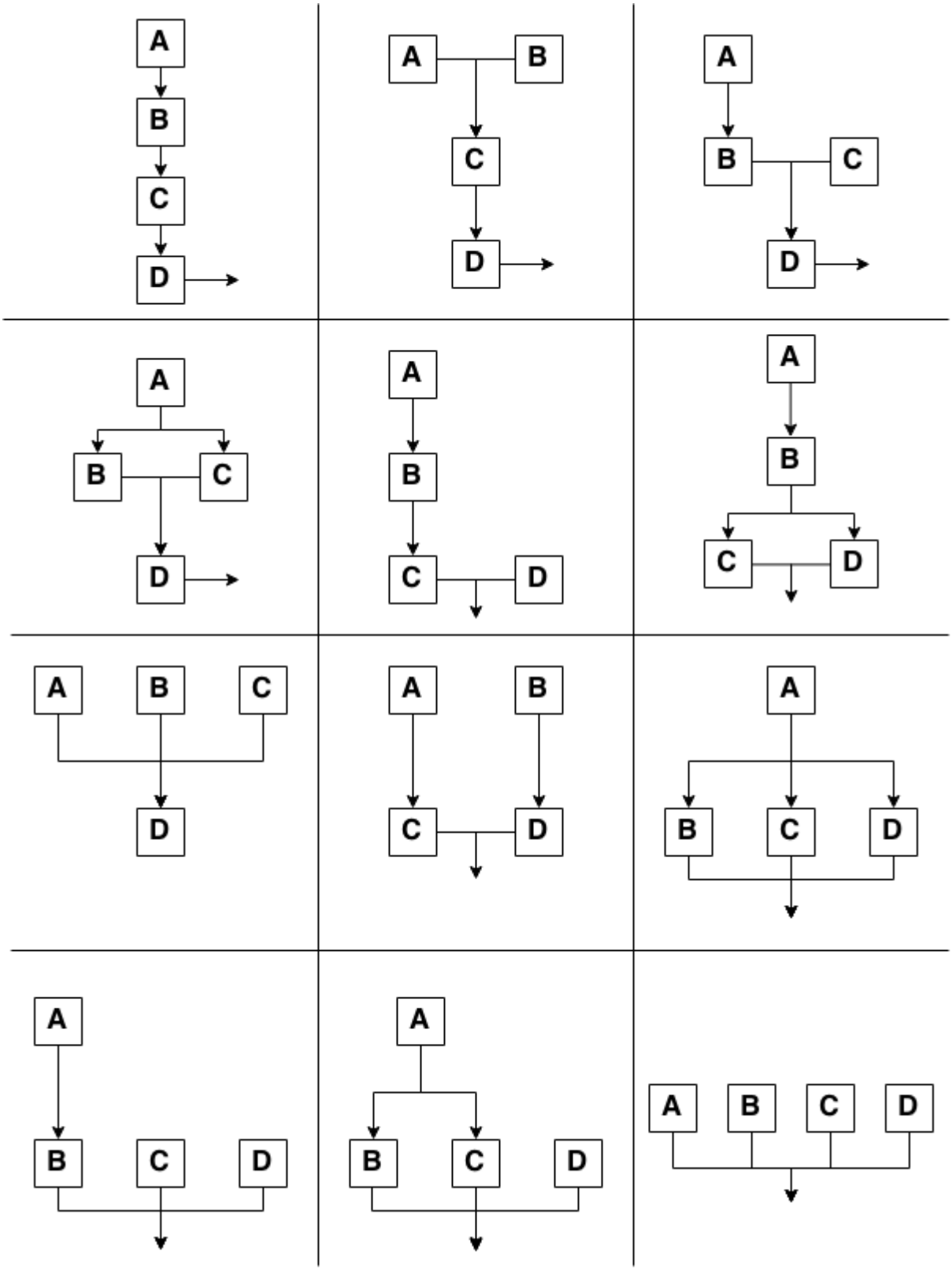

Figure 4.3: Signal flows for FM Synthesis FM Algorithms with four Operators, A, B, C and D. A direct connection between two Operators means modulation of the bottom Operator by the top Operator. Signals of Operators positioned side-by-side are summed.

Conclusion

This thesis has shown the most important steps of how the C++ programming language can be used to create a digital synthesis system. It has used available knowledge sources off and online to discuss the theoretical concepts behind digital audio processing and then gave practical, tested examples of how these concepts can be implemented in C++.

The first chapter outlined the basic knowledge required to discuss digital sound, and hinted at the differences between sound in the analog and the digital realm. Firstly, it was postulated that in the digital world, sound is not a continuous, but a discrete signal, composed of many individual *samples*. Chapter 1 also explained that the rate at which these samples are recorded from a continuous signal is called the *sample rate*. Moreover, it was discussed that the maximum frequency of a signal to be sampled must be less than or equal to one half of the sample rate — the *Nyquist Limit* — to ensure *proper sampling*. Lastly, a common phenomenon associated with sample rates, *Aliasing*, which occurs when a signal is sampled improperly and which results in a misrepresentation of the signal when *quantized*, was also examined.

The subsequent chapter dealt with the generation of digital sound. It explained that audio waveforms could be created either mathematically, yielding *exact* waveforms, or via *Additive Synthesis*, producing more natural sounds. It was then shown how three popular waveforms — square, sawtooth and triangle — can be created with Additive Synthesis. Moreover, the concept of a *Wavetable*, to efficiently store and play-back such waveforms, was also investigated. Lastly, a special form of sound, *Noise*, was studied and its implementation discussed.

Chapter 3 concerned itself with the modulation of sound. Two common means of modulation, *Audio Envelopes* and *Low Frequency Oscillators* (LFOs), were examined in detail. This chapter also gave a concrete example of how a modular dock of modulation sources, a so-called *ModDock*, is a very efficient and intuitive method of making it possible to modulate a variable number of parameters in a synthesis system by a variable number of modulation sources.

The last chapter analyzed how many different sound signals can be *synthesized* to give an entirely new signal. Popular methods of synthesis were mentioned, while one — *Frequency Modulation* (FM) *Synthesis* — was discussed in particular. Some important concepts concerning FM Synthesis, such as the $C : M$ ratio, *FM Algorithms* or the phenomenon of *sidebands*, were examined. Subsequently, the process of implementing FM Synthesis in C++ was outlined.

In summary, it can be said that this thesis has shown the most essential steps of creating a digital synthesizer in C++. Naturally, not all steps of the complete process were given. As a matter of fact, this thesis initially included chapters on a multitude of other subjects associated with the creation of a digital synthesizer — such as digital filters, effects or sound output — which ultimately had to be excluded from the final version due to a limit on its length. However, these topics could be handled in a future paper. Additionally, it should be mentioned that this thesis did not discuss cutting-edge research or recent findings that may be used in commercial synthesizers today, but tried to explain approved and already implemented methods. However, enough theory and practical examples for anyone interested in creating his or her own digital synthesizer in C++, to begin the journey, were given.

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 4.3. This code excerpt is from the `FM` class.

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] 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.

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

[6] Gordon Reid,
Synth Secrets, Part 13: More On Frequency Modulation.
`http://www.soundonsound.com/sos/may00/articles/synth.htm`
Accessed: 30 December 2014.

[7] *Tutorial for Frequency Modulation Synthesis*.
http://www.sfu.ca/~truax/fmtut.html
Accessed: 30 December 2014.

[8] 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.

[9] John D. Cutnell and Kenneth W. Johnson,
Physics.
Wiley,
New York,
4th Edition,
1998.

[10] Electronic Music Wiki,
DX7.
http://electronicmusic.wikia.com/wiki/DX7
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 25, 2015,

Peter Goldsborough