Developing a Digital Synthesizer in C++

Peter Goldsborough
petergoldsborough@hotmail.com

December 19, 2014

Contents

Chapter 1

Sound in the Digital Realm

1.1 What is a Music Synthesizer?

1.2 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.

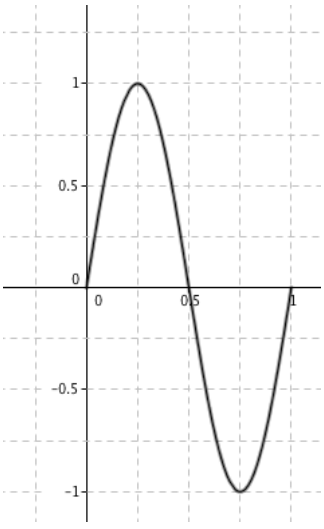

Figure 1.1: The continuous representation of a typical sine wave. In this case, both the signal's frequency f as well as the maximum elongation from the equilibrium a are equal to 1.

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 (Mitchell, 2008, p. 16). 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) (Mitchell, 2008, p. 16). The reverse process of converting discrete samples to a continuous signal is called *digital-to-analog-conversion* (Mitchell, 2008, p. 17). 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.

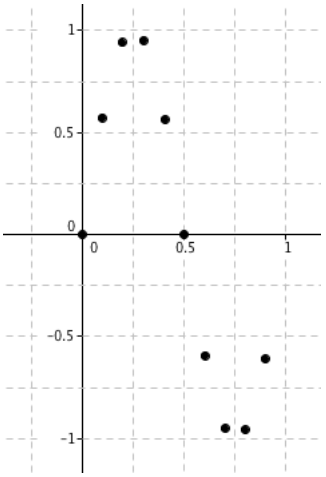

Figure 1.2: The discrete representation of a typical sine wave.

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

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

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

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

1.4 Nyquist limit

The sample rate determines the range of frequencies that can be represented by a digital sound system, as only frequencies that are less than or equal to one half of the sampling rate, where

it is possible to take at least one sample above the equilibrium and at least one sample below the equilibrium for every cycle (Mitchell, 2008, p. 18), 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*. 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" (Smith, 1999, p. 40). Any frequencies above the Nyquist limit lead to *aliasing*, which is discussed in the next section.

Given the definition of the Nyquist limit and considering the fact that the limit of human hearing is approximately 20 kHz (Cutnell & Johnson, 1998, p. 466), the reason for which the two most common audio sample rates are 40 kHz and above is clear: they were chosen to allow the "proper" representation of the entire audio frequency range, since a sample rate of 40 kHz meets the Nyquist requirement of a sample rate at least twice the maximum frequency component of the signal to be sampled (the Nyquist limit), in this case ca. 20 Khz.

1.5 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, which, when reconstructed, 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 samples per second, 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).

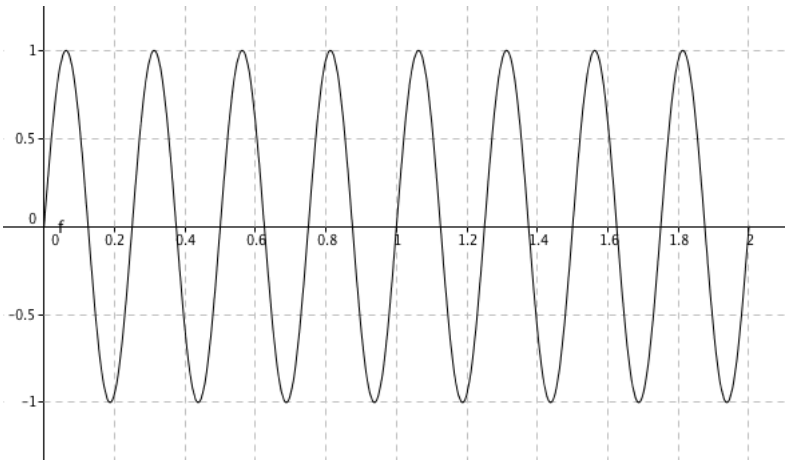

Figure 1.3: A sinusoid with a frequency of 4 Hz.

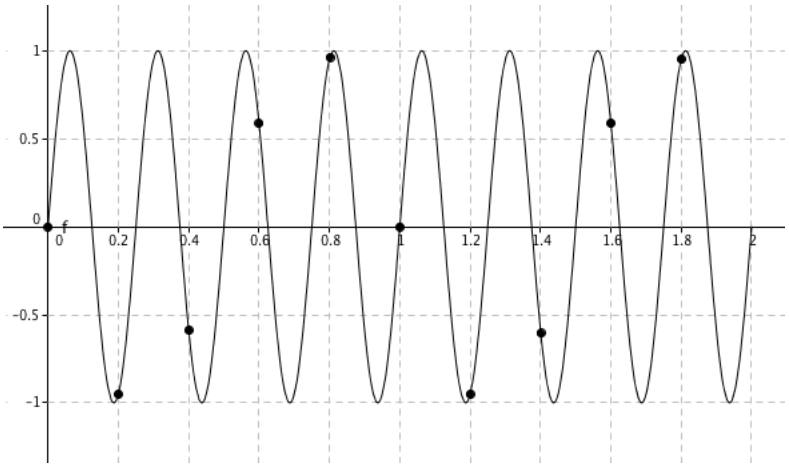

Figure 1.4: A sinusoid with a frequency of 4 Hz, sampled at a sample rate of 5 Hz. According to the Nyquist Theorem, this signal is sampled improperly, as the frequency of the continuous signal is not less than or equal to one half of the sample rate, in this case 2.5 Hz.

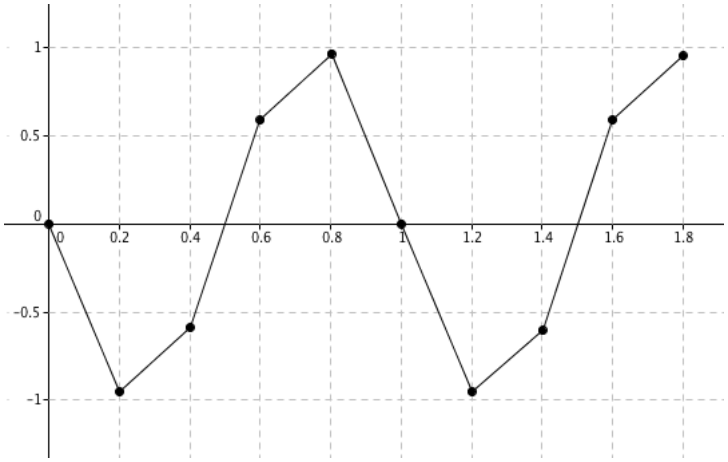

Figure 1.5: An approximated representation of the signal created from the sampling process shown in Figure 1.4. Visually as well as acoustically, the new signal is completely different from the original signal. However, in terms of sampling, they are indistinguishable from each other due to improper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

1.6 Overflow

Another interesting property of digital sound, which is not encountered in the analog world, is that it can overflow. When we attempt to increase the loudness of something in the analog world, e.g. by hitting a drum more intensely, the expected result is a louder sound. In the digital realm however, it may occur that attempting to increase a signal's amplitude does not result in an increased loudness, but in distortion. The cause of this phenomenon lies in the way digital audio is stored. Since computer memory is a finite resource, each sample has a dedicated portion of computer memory allocated to it. For example, the Waveform Audio File Format (WAVE), a common computer file format for audio data, stores each sample of an audio track as a 16-bit signed integer. A 16-bit signed integer gives a possible range of -2^{16-1} to 2^{16-1} ($16-1$ because the most significant bit is used as the sign bit in two's complement representation). This means that a signal with an amplitude of 1 will be stored as 32767, an amplitude of 0.5 as 16384, an amplitude of -1 as -32768 and so on. If one tries to increase the amplitude of a signal whose value has already saturated the available range and space allocated to it, in this case 32767 on the positive end and -32768 on the negative end, the result is that the integer with which the sample is stored *overflows*. Because WAVE files (and many other storage media) store samples as *signed* integers, overflow always results in a change of sign:

$$32767_{10} = 0111111111111111_2$$

$$0111111111111111_2 + 1 = 1000000000000000_2 = -32768_{10}$$

$$1000000000000000_2 - 1 = 0111111111111111_2 = 32767_{10}$$

A visualization of the result of increasing the amplitude of a signal with a saturated value (an amplitude of 1), shown in Figure 1.6, is given in Figure 1.7.

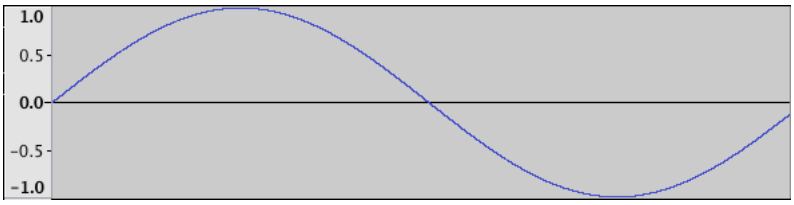

Figure 1.6: A typical sinusoidal signal with an amplitude of 1. The integer range provided by the allocated memory for the top-most sample is saturated, meaning it is equal to 32767_{10} or 0111111111111111_2.

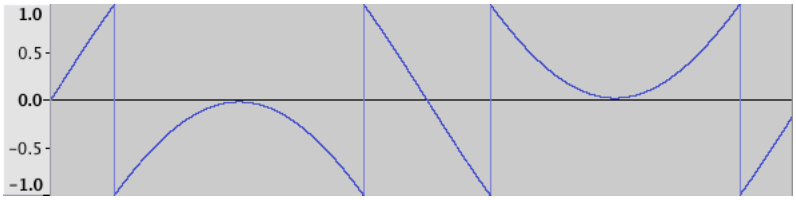

Figure 1.7: What happens when the amplitude of the signal from Figure 1.6 is increased by a factor of 2. Because the top-most sample had a saturated value, increasing it results in overflow into the negative range. Also other samples have overflowed and thus changed their sign. Because of the way two's-complement representation is implemented, the signal continues it's path as if no overflow had ever occurred. The only difference being, of course, that the sign has changed mid-way.

Chapter 2

Generating Sound

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

2.1 Simple Waveforms

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

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

A computer program to compute the values of a sine wave with a variable duration, implemented in C++, is shown in Table 2.1. 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:

$$f_c(t) = A\cos(2\pi ft + \phi) = A\sin(2\pi ft + \phi + 90) \tag{2.2}$$

Therefore, the program from Table 2.1 could be modified to compute a cosine wave by changing line 22 from:

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

to

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

2.2 Complex Waveforms

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

```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;
}
```

Table 2.1: C++ implementation of a sine wave generator.

2.2.1 Mathematical Calculation of Complex Waveforms

In the first case — mathematical calculation, waveforms are computed according to certain mathematical formulae and thus yield *perfect* or *exact* waveforms, such as a square wave that is equal to the maximum amplitude exactly one half of a period and equal to the minimum

amplitude for the rest of the period. While these waveforms produce a very crisp and clear sound, they are rarely found in nature due to their degree of perfection and are consequently rather undesirable for a music synthesizer. Nevertheless, they are considerably useful for modulating other signals, as tiny acoustical imperfections such as those found in additively synthesized waveforms can result in unwanted distortion which is not encountered when using mathematically calculated waveforms. Therefore, exact waveforms are the best choice for modulation sources such as Low Frequency Oscillators (LFOs), which are discussed in later chapters. (Mitchell, 2008, p. 71)

The following paragraphs will analyze how four of the most common waveforms found in digital synthesizers, the square, the sawtooth, the ramp and the triangle wave, can be generated via mathematical calculation.

Note: There have found to be disparities in literature over which waveform is a sawtooth and which a ramp wave. This thesis will consider a sawtooth wave as descending from maximum to minimum amplitude with time and a ramp wave as ascending from minimum to maximum amplitude with time.

Square Waves

Ideally, a square wave is equal to its maximum amplitude for exactly one half of a period and equal to its minimum amplitude for the other half of the same period. A single period of a square wave can be calculated as shown in Equation 2.3. A mathematical equation for a full, periodic square wave function is given by Equation 2.4 and an equivalent computer program is shown in Table 2.2.

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

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

Sawtooth Waves

2.2.2 Additive Synthesis

The second method, Additive Synthesis, produces waveforms that, despite not being mathematically perfect, are closer to the waveforms found naturally. This method involves the summation of a finite set of sine waves with varying parameters and 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.

2.3 Real-time calculation versus Wavetable lookup

Before the generation of more complex waveforms than the sine and cosine waves from section 2.1 can be discussed, the two options for playing back waveforms in a digital synthesis system must be examined: continuous real-time calculation of waveform samples or lookup from a table that has been calculated once and then written to disk.

```
double* directSquare_(const unsigned int length) const
{
    // the sample buffer
    double * wt = new double [length + 1];

    // time for one sample
    double sampleTime = 1.0 / length;

    // the midpoint of the period
    double mid = 0.5;

    double ind = 0;

    // fill the sample buffer
    for (int n = 0; n < length; n++)
    {
        wt[n] = (ind < mid) ? -1 : 1;

        if ( (ind += sampleTime) >= length)
        { ind -= length; }
    }

    return wt;
}
```

Table 2.2: C++ code to generate and return one period of a square wave, where `length` is the period duration in samples.

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] Gordon Reid,
Synth Secrets, Part 12: An Introduction To Frequency Modulation.
http://www.soundonsound.com/sos/apr00/articles/synthsecrets.htm
Accessed: 8 October 2014.

[4] 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.

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