Wavetables From Recorded Sounds

The wavetable oscillator shown above uses a wavetable containing exactly one period of the waveform. However, a wavetable can contain multiple periods of the waveform. When the wavetable contains a recorded sound rather than a calculated waveform it always has more than one period. In addition, the wavetable may have been recorded at a sample rate different from the playback sample rate.

We can change the pitch of the recorded sound if we use a phase increment other than 1, just as with single period wavetables. In order to calculate the phase increment for a desired frequency, we must know the frequency of the recorded waveform and the sample rate at which it was recorded. With those two values we can easily calculate one period of the waveform.

The frequency of a recorded sound could be specified directly, but, for most wavetables based on recorded sound, the frequency is specified as combination of pitch class (or keyboard key number) and frequency deviation in cents (1/100 of a semitone). Using the pitch number 48 as middle C and A4=440Hz as the reference frequency, we can calculate the frequency of the wavetable (f_w) as follows.

$$f_w = 440 \cdot 2^{\frac{key-57}{12}} \cdot 2^{\frac{cents}{1200}} = 440 \cdot 2^{\frac{(key-57)+(cents \cdot 0.01)}{12}} \qquad (8.2)$$

Note that we can use any frequency as the reference frequency so long as we adjust the pitch number offset appropriately. For example, if the wavetable uses MIDI key 60 for middle C, we simply change the offset to 69. Likewise, if we use the frequency for pitch number zero in place of 440Hz, we eliminate the subtraction of the pitch number offset.

Applying the rules for exponents we can combine the two exponential terms in (8.2) together, as shown in the second form of the equation. However, since the two values for key number and cents are specified individually, we can just as easily use the first form. In order to improve performance, we use a table lookup to convert key to frequency and cents to a multiplier, avoiding the time consuming exponential calculations at runtime.

Once we know the frequency we can calculate the period of the waveform in samples. The period length in samples (P) is the time of

one period (t_w) divided by the time of one sample (t_s). The time of one sample is the inverse of the wavetable sample rate (f_{sw}) and the time of one period is the inverse of the wavetable frequency (f_w).

$$t_w = \frac{1}{f_w}$$

$$t_s = \frac{1}{f_{sw}}$$

$$P = \frac{t_w}{t_s} = \frac{1/f_w}{1/f_{sw}} = \frac{f_{sw}}{f_w} \tag{8.3}$$

In words, the period of the waveform is simply the sample rate of the wavetable divided by the frequency of the wavetable. We can also see this by using the phase increment for playback at the original frequency. In order to play the wavetable at its original frequency, the phase increment must be equal to 1. Rearranging the terms gives us the same result as before.

$$1 = \frac{P \cdot f_w}{f_{sw}}$$

$$P = \frac{f_{sw}}{f_w}$$

Once we know the period of the wavetable we can calculate the phase increment as before using the period (P) for table length (L). Substituting and then refactoring gives us the final equation for pitch shifting a sampled recording.

$$i = \frac{\left(f_{sw}/f_w\right) \cdot f_o}{f_s}$$

$$i = f_{sw} \cdot \frac{1}{f_w} \cdot f_o \cdot \frac{1}{f_s}$$

$$i = \frac{f_{sw}}{f_s} \cdot \frac{f_o}{f_w} \tag{8.4}$$

Intuitively, this is correct as it shows the phase increment is the ratio of the desired pitch to the recorded pitch. When the recorded sample rate is the same as the playback sample rate, the first factor is 1. Likewise, when the recorded frequency is the same as the playback frequency, the second factor is also 1. Thus the phase increment is 1 and the result is to playback the wavetable as recorded. An increase in pitch results in a phase increment greater than 1, while a decrease in pitch results in a phase increment less than 1.

When writing the code to implement this oscillator, we should pre-calculate portions of (8.4). The ratio of sample rates and the recorded frequency will remain invariant for a note and the three associated factors can be calculated once and stored as a multiplier for the desired frequency.

```
frqMult = wavetable.sampleRate;
frqMult /= synthParams.sampleRate;
frqMult /= wavetable.frequency;
phaseIncr = frequency * frqMult;
```

In addition to multiple periods of the sound, a typical recorded wavetable will be divided into two or three segments. The first segment represents the attack of the sound and is played straight through. The middle segment represents the sustain portion of the table and is repeated as long as the note is sustained. When a third section is present, it is played during the note release. If a third section is not present, the looped portion is played through the release. By creating a looping section in the wavetable, the length of the sound is not limited to the recording length. We can extend the sound indefinitely by looping while still retaining a close approximation of the original recording.

Because the wavetable loops somewhere in the middle of the wavetable we cannot simply wrap the phase between the beginning and end of the table. Instead, we must keep track of the segment we are playing and only wrap the phase during looped portions of the sound.

Some sampled sounds are not intended to be looped and will contain only one segment in the wavetable. Percussion sounds,

plucked strings, and sound effects are typically played straight from beginning to end. To handle those sounds, we need a flag to indicate we should not loop, but continue until we reach the end of the table, returning zeros after that point. We can do this easily by using a state variable to indicate how and when to wrap the phase.

The following code shows phase wrapping in the oscillator for this type of wavetable.

```
// state:
//   0 = in attack portion
//   1 = in loop portion
//   2 = play to end without looping.
   if (phase < 0)
      phase += period;
   if (state == 0) {
      if (phase >= loopStart)
         state = 1;
   }
   else if (state == 1) {
      if (phase >= loopEnd)
         phase -= loopLength;
      else if (phase < loopStart)
         phase += loopLength;
   }
   else if (state == 2) {
      if (phase >= tableEnd)
         return 0;
   }
```

The check for phase less than 0 and less than loop start in state 1 is needed in case the oscillator is modulated and the modulation amplitude is negative, causing the phase to move backwards. If no modulation is applied, these tests can be removed.

When the wavetable contains a release section, we will need to add some kind of event that causes the state to shift to 2 when the sound is released. This will cancel any looping and cause the sound to play to the end.

Generating samples is simply a matter of indexing into the wavetable as before. However, for this kind of wavetable it is good idea to always use some form of interpolation. Simple rounding of the index can work for long wavetables, but that would be the equivalent of requiring samples recorded at a significant multiple of the

synthesizer sample rate (e.g., 96K or 192K). Several forms of interpolation are possible, including convolution with a *sinc* function (See Chapter 10, below). However, the linear interpolation shown above works well enough for most synthesis systems.

Recorded sounds may also contain two channels (stereo). Several options are available in that case. We could simply discard one channel, we could average the two channels, or produce separate values for left and right. When writing the program for the oscillator we should calculate both left and right values and then allow the caller to decide which values to use. The following code shows the calculation of the sample from the current phase and calculation of both left and right channel values. The average of the two is returned, but the caller may retrieve the left and right values separately if desired. We assume the left channel wavetable always exists and is used as a monophonic channel if the right channel wavetable is missing.

```
int ii = (int) phase;
float fract = phase - (float) ii;
v1 = wtLeft[ii];
v2 = wtLeft[ii+1];
left = v1 + ((v2 - v1) * fract);
if (wtRight) {
   v1 = wtRight[ii];
   v2 = wtRight[ii+1];
   right = v1 + ((v2 - v1) * fract);
   out = (right + left) / 2.0;
} else {
   out = right = left;
}
return out;
```