Audio Synthesis Basics

Analog Synthesis
Intro to Digital Oscillators

Analog Synthesis Overview

- Sound is created by controlling electrical current within synthesizer, and amplifying result.
- Basic components:
  - Oscillators
  - Filters
  - Envelope generators
  - Noise generators
- Voltage control

Oscillators

- Creates periodic fluctuations in current, usually with selectable waveform.
- Different waveforms have different harmonic content, or *frequency spectra*.

Filters

- Given an input signal, attenuate or boost a frequency range to produce an output signal
- Basic Types:
  - Low pass
  - High pass
  - Band pass
  - Band reject (notch)

Envelope Generators

- Generate a control function that can be applied to various synthesis parameters, including amplitude, pitch, and filter controls.

Noise Generators

- Generate a random, or semi-random fluctuation in current that produces a signal with all frequencies present.

Digital Synthesis Overview

- Sound is created by manipulating numbers, converting those numbers to an electrical current, and amplifying result.
- Numerical manipulations are the same whether they are done with software or hardware.
- Same capabilities (components) as analog synthesis, plus significant new abilities

Digital Oscillators

- Everything is a Table
  - A table is an indexed list of elements (or values)
  - The index is the address used to find a value

Generate a Sine Tone Digitally (1)

- Compute the sine in real time, every time it is needed.
  - equation:

$$signal(t) = r\sin(\omega t)$$

  - t = a point in time; r = the radius, or amplitude of the signal; w (omega) = 2pi*f the frequency
  - Advantages: It's the perfect sine tone. Every value that you need will be the exact value from the unit circle.
  - Disadvantages: must generate every sample of every oscillator present in a synthesis patch from an algorithm. This is very expensive computationally, and most of the calculation is redundant.

Generate a Sine Tone Digitally (2)

- Compute the sine tone once, store it in a table, and have all oscillators look in the table for needed values.
  - Advantages: Much more efficient, hence faster, for the computer. You are not, literally, re-inventing the wheel every time.
  - Disadvantages: Table values are discrete points in time. Most times you will need a value that falls somewhere in between two already computed values.

Table Lookup Synthesis

- Sound waves are very repetitive.
- For an oscillator, compute and store one cycle (period) of a waveform.
- Read through the wavetable repeatedly to generate a periodic sound.

Changing Frequency

- The Sample Rate doesn't change within a synthesis algorithm.
- You can change the speed that the table is scanned by skipping samples.
- skip size is the <u>increment</u>, better known as the <u>phase increment</u>.

  phase increment is a very important concept

Algorithm for a Digital Oscillator

- Basic, two-step program:
  - $phase\_index = \text{mod}_L(previous\_phase + increment)$
  - $output = amplitude \times wavetable[phase\_index]$
- $increment = \dfrac{(TableLength \times DesiredFrequency)}{SampleRate}$

If You're Wrong, it's Noise

- What happens when the phase increment doesn't land exactly at an index location in the table?
  - It simply looks at the last index location passed for a value.
    In other words, the phase increment is truncated to the integer.
- Quantization
- Noise
- The greater the error, the more the noise.

Interpolation

- Rather than truncate the phase location…
  - look at the values stored before and after the calculated phase location
  - calculate what the value would have been at the calculated phase location if it had been generated and stored.

  Interpolate
- More calculations, but a much cleaner signal.

Sample Playback

- Oscillator concept can be used to explain sample playback, with one important caveat:
  - Table length is variable among different soundfiles, so
  - Playback rate is usually expressed in terms of a ratio: *desired_speed* : *root_speed*

Delay

- Delay is a fundamental operator!
  - Also easy to do in digital
  - Long delays – echos, reverb
  - Short delays - filtering
- How do we delay sound?
  - Queues
  - Consider using circular queues

Circular queue implementation

- Initialization
  - Mono queue, 1 second long

```
// We'll delay one second
int DELAY = int(SampleRate());

short *queue = new int[DELAY + 1];
int rdloc = 1;
int wrloc = 0;

// Initially zero the queue
for(int j=0; j<DELAY + 1; j++)
  queue[j] = 0;
```


Accessing the queue

```
// For each sample…

    // Queue it
    queue[wrloc] = sample;

    // Add in the delayed version
    sample += queue[rdloc];

    // Update queue locations
    wrloc++;   wrloc %= DELAY + 1;
    rdloc++;   rdloc %= DELAY + 1;

// And write the samples
```


What about a multi-tap queue?

- Make queue 1 larger than longest delay
- Write at wrloc each step
  - Increment wrloc: wrloc++; wrloc %= QSIZE;
- Read at:
  - (wrloc – delay + QSIZE) % QSIZE;

Samples or Wave Tables

- A *sample* or *wave table* is a short digital sound recording
  - We play it back to make the sound
- Examples:
  - Digital piano – recorded sound for each key
  - Speech synthesis – recorded sound for each phoneme
  - Computer games – samples for gunshots, crashes, etc.

Samples, the easiest way

Current sample

Start with first audio frame in sample

Each request for a sample, advance
return wave[curr++];

At end, we are done

More advanced ideas

- What if I want to play at a different speed?
  - Playing faster or slower changes the pitch

Music and the scale

- Music is based on an exponential scale
  - To move up one octave, we double the frequency
  - To move down one octave, we halve the frequency
- There are 12 “semitones” in a scale
  - Sometimes called “half-steps”
  - C, C#, D, D#, E, F, F#, G, G#, A, A#, B
  - To move up one semitone, multiply playout rate by 1.05946 (1.05946 ^ 12 = 2.0)
  - To move down one semitone, divide playout rate by 1.05946

Example: Playing a violin note

- Recording of violin playing C, we want to play E (4 half-steps up)
- Playout rate is 1.05946^4 = 1.2599
- So, how do we play at that rate?

Fractional sample positions

```
// Initialization:
sample = 0.0;        // double
rate = 1.2599;       // double

…

// After each sample acquisition
sample += rate;
```


How to select the sample

- Important: Desired sample is between real samples!
- We can:
  - 1: Select the nearest sample
  - 2: Linearly interpolate between samples
  - 3: Resample

Selecting the nearest sample

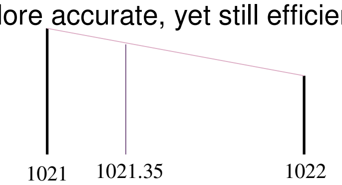
- Simply round and access your wave table
  - return wave(int(sample + 0.5));
- Works, but is somewhat noisy

Linear interpolation

- Interpolate between two audio samples

```
double inbetween = fmod(sample, 1);
return (1. – inbetween) * wave[int(sample)] +
        inbetween * wave[int(sample) + 1];
```

- More accurate, yet still efficient

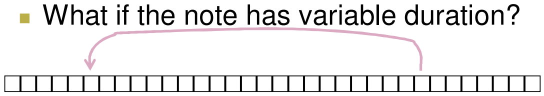


Sample playback class members

- Constructor – Loads file and initializes for playback
- Pitch – Sets the pitch to play back at
- Frame – Returns an audio frame and advances
- Rewind – Resets to play again
- Done – Returns true if playback is done

Looping

- What if the note has variable duration?

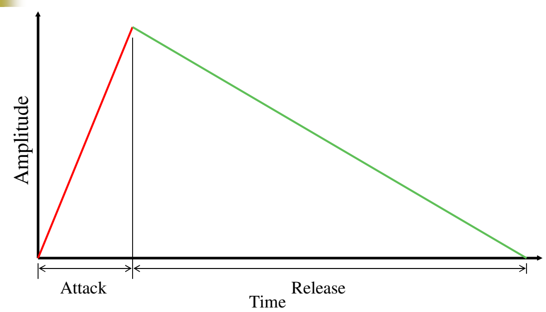


- Associate with the sample
  - Loop from location
  - Loop to location
- How might we select these points?

Envelopes

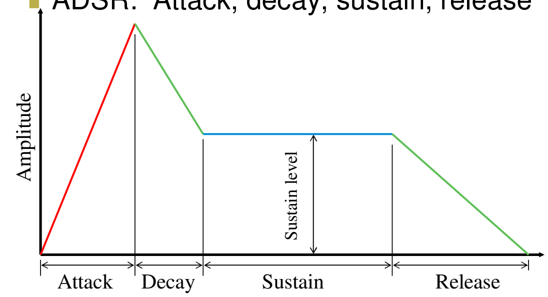
- What if we use looping to make an efficient piano sound?
  - Looping does not decay, but a piano sound does
- We commonly will make samples with fixed amplitudes, then make a synthetic *envelope* for the sound event.

Attack and Release



ADSR

- ADSR: Attack, decay, sustain, release



Where do samples come from?

- Pure recordings of instruments
- Artificially generated sounds
- Modifications of existing sounds