ES3 Lecture 12

Realtime audio on mobile devices

Recommended reading

- **Real sound synthesis for interactive applications** by *Perry Cook* [2002]
  - short, but complete and well written introduction to audio synthesis

- Julius O. Smith has three very good (but technical) online books on audio processing
  - **Introduction to Digital Filters with Audio Applications** by *Julius O Smith* [2009]
    - www.dsprelated.com/dspbooks/filters
  - **Mathematics of the DFT with Audio Applications** by *Julius O Smith*
    - www.dsprelated.com/dspbooks/mdft
  - **Physical Audio Signal Processing** by *Julius O Smith*
    - www.dsprelated.com/dspbooks/pasp
  - **Spectral Audio Signal Processing** by *Julius O Smith*
    - www.dsprelated.com/dspbooks/sasp

  - **Computer Music Tutorial** by *Curtis Roads*
    - Very complete introduction to ditgal audio

  - Lots of very useful code snippets at **musicdsp.org**

Digital Audio

- Sounds consist of pressure waves
  - variations in air pressure levels are picked up by the ears
- Sounds are by their nature *analog*
  - They vary continuously in both time and value
- In order to deal with them on a computer, a digital representation is required
  - Discrete time, and discrete value
- There is a very important result that shows that if you *sample* (measure) an analog value frequently enough, and with enough resolution, it can be reproduced nearly perfectly
  - The speed of sampling determines the maximum frequency which can be represented
  - Maximum frequency is 1/2 the sample rate -- the **Nyquist** rate
  - The number of levels used determine the accuracy of the signal
    - Fewer levels mean noisier signals

Sampling

- To represent a sound, regularly spaced samples are taken
  - Samples have a *rate* and a *resolution*

- Humans can hear at the most up to about 20000Hz
  - So a sampling rate of around 40000Hz can represent all audible frequencies
    - e.g. CD audio is sampled at 44100Hz, SACD
  - Lower sample rates occupy less space (obviously) but lose high frequency components

- The resolution specifies the number of possible levels used. Common values are:
  - 8 bit: 256 level, sounds crude and noisy, but was often used in old hardware
  - 12 bit: 4096 levels, used on many old digital musical instruments
  - 16 bit: 65536 levels, the most common digital standard. Resolution above this are not audible.
  - 24 bit: 16777216 levels. Used in professional audio. This resolution is used because certain processing can reduce the levels available -- this would result in noticeable degradation at 16 bit.
  - 32 bit or 64 bit: floating point. Used for ease and speed of computation

PCM Data

- The canonical form for audio data is PCM (pulse code modulation)
  - Just a sequence of integer values representing sound levels
  - Assumes a constant sample rate
- All (well, almost all) digital audio hardware uses this internally at some stage
  - A/D convertors convert analog signals (e.g. from a microphone) to PCM
  - D/A convertors convert it back into electrical signals (to go to speakers)
- It is very easy to manipulate audio data in PCM format
  - e.g. to mix two sounds, their PCM representations can just be added

Formats

- Raw PCM data can have several forms
  - When working with PCM data, you need to know the format!
  - It has a **sample rate**
    - e.g. 44100Hz
  - It has a **resolution** or **bit depth**
    - e.g. 16 bit
  - It has a **signedness**
    - PCM can either be unsigned (0-65535, for example) or signed (-32768--32767, for example)
    - Signed data is generally easier to work with
  - It has an **endianness**
    - order of bytes in machine representations of words
  - It has a **number of channels**
    - e.g. 1 for mono, 2 for stereo, 6 for surround

WAV files

- WAV files are commonly used to store PCM data
  - (although they can store compressed data as well)

- Just has a header specifying the features listed on previous page
  - and the length of the data
  - followed by a block of binary data with the PCM data

- Lots of standard routines for reading/writing WAV files
  - e.g. using the **AudioToolbox** library on the iPhone

Compressed Formats

- Raw PCM audio data is often very large
  - e.g. 1 minute at 44100Hz, 16 bit = 5.2Mb
- Lossy compressed formats remove data which are less perceptually important
- Simple mulaw coding reduces the dynamic range of a signal using an exponential signal
  - changes in small values are more important than changes in large values
- MP3 coding splits up sound files into chunks, and splits those chunks into frequency bands
  - throws away those that are not "perceptually important" according to a fairly complex model
  - results in huge filesize reduction but often very similar sounding sounds
- Compressed formats are always converted to raw PCM before playback!

Buffers

- Almost all digital audio hardware (and audio APIs) use **buffers**
- Data is passed to the hardware in blocks
  - e.g. of 2048 samples

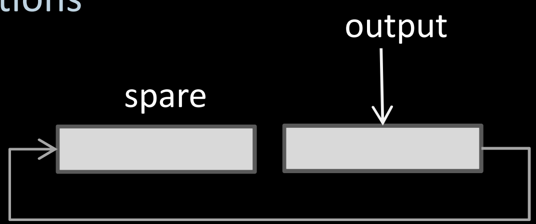
- APIs never have methods like **outputNextSample**()
  - **Instead, you fill a whole buffer of data and pass that in**

- Audio data is expensive to process and is **absolutely** time critical
  - a variation of a few microseconds will corrupt the sound
  - hardware takes care of streaming data to the D/A from the buffer
  - buffering eliminates any errors in timing
    - so long as the buffer is longer than any timing variation

- You must be able to fill the buffers fast enough
  - otherwise the audio hardware will glitch, usually with sonically devastating results

Buffering

- The disadvantage of buffering is *latency*
  - The longer the buffer is, the longer between an event being detected (e.g. a tap) and a sound being output
    - 2048 sample buffer is 46ms at 44100Hz (reasonable)
    - 65536 sample buffer is 1.49 seconds (not reasonable!)
  - In very sensitive tasks (like drumming) humans can detect latency down to around 5ms
  - 2ms latency is usually desirable in professional musical applications
    - only 88 sample buffer at 44100Hz!



- Most APIs have a callback system
  - You register a function to fill buffers
  - Each time the audio API runs out of data, it automatically calls your function to fill the buffer

- If there were only one buffer, this would lead to glitches between buffers!
  - Usually have at least two buffers
  - The API asks you to fill a buffer which is not currently being output

Simple Example

- Using an imaginary Objective-C API:

```
// in init...
[soundDriver registerCallbackTarget:self action:fillBuffer];


- (void) fillBuffer(int length, SInt16 *buffer)
{
  for(int i=0;i<length;i++)
  {
    // produces pure tone at high A (440Hz) (assuming 44100Hz sampling rate)
    double v = sin((440*i*2*M_PI)/44100.0);

    // Buffer is signed 16 bit integers
    // multiply floating point value -1 .. 1 by 32767 to fit to range
    buffer[i] = v * 32767;
  }
}
```

- Every time the hardware needs more data, it calls **fillBuffer**; and gets some more data

Floating-point and integer

- PCM data is usually integer
- On many devices, integer operations are much much faster than floating point operations
  - but **not** on modern desktop processors -- floating point is faster!

- Unfortunately, it's much easier to work with sampled data in floating point
  - Either have to do processing in floating point and convert at the end...
  - Or use integer versions of routines

- Large literature exists on efficiently implementing audio synthesis and processing effects using only integer instructions
  - Problems often resolve around loss of precision
  - e.g. sum together 64 16 bit integers and divide by 64 to get the average...
  - result has only 10 bits of resolution!

Playing a sample back

- The simplest thing to do is to play back pre-recorded sound
- We will assume the pre-recorded sound is PCM, with the same format as the output API (i.e. same sample rate, bit depth, same number of channels)
  - Otherwise will have to convert!
  - Converting between sample rates accurately is very hard....
  - Although converting between signedness, endianness and bit-depth is very easy
- All that needs be done is to copy the data into the buffers

Simple Playback

```
SInt16 *PCMSample;
int sampleLength;
int samplePointer = 0;

// Assume this loads a sample into PCM sample
loadSample(PCMSample, &sampleLength);


- (void) fillBuffer(int length, SInt16 *buffer)
{
  for(int i=0;i<length;i++)
  {
          if(samplePointer<sampleLength)
                  buffer[i] = PCMSample[samplePointer++];
          else
                  buffer[i] = 0;

  }
}
```

Mixing samples

- Having one sample playing is useful, but often multiple layers needed
  - e.g. in a musical instrument, many notes can be playing at once

- Can just add together samples (possibly scaling them down to reduce volume) to mix them together

- Often need to exactly specify starting point of sample
  - since we are dealing with buffers, we can't just start the sample at the **fillBuffer** function call
  - timing of samples will be limited to multiples of the buffer length
    - sounds bad, gives a staccato machine gun effect when many samples are triggered
    - sound playback should never depend on buffer length!

- The solution to this is to maintain a queue of currently active samples
  - Each with a starting offset, representing the number of samples from now to start the sample

Event Queues

- Each element of the queue is of the form (time, sampleData)
  - (-210, <SampleData 0x45AD>)
  - (51, <SampleData 0x5010>)
  - (1813, <SampleData 0x5014>)
  - (4003, <SampleData 0x5018>)

- Queue is maintained in sorted order
  - A negative time indicates a currently playing sample

- On each **fillBuffer**, decrement the time by the length of the buffer
  - if it is or becomes negative, will need to be mixed into the buffer
  - if -time > sample length, remove the sample from the queue (because it finished)

Better sample player

```
- (void) fillBuffer(int length, SInt16 *buffer)
{
  //it's faster to do the loops in the other order, but this is clearer
  for(int i=0;i<length;i++)
  {
          int v = 0;
          for(SoundEvent *event in queue)
          {
                    // add in currently playing samples
                    if(event.time-i<0 && -(event.time-i) < event.length)
                       v = v + event[-(event.time-i)];

                    // remove old samples
                    // in practice it is dangerous to remove an element from
                    // a queue we are iterating over...
                    if(-(event.time-i)>=event.length)
                       [queue removeElement:event];
          }

          buffer[i] = v;
  }
  // move the buffer on
 for(SoundEvent *event in queue)
          event.time -= length;
}
```

Frequency adjustment

- Frequency of samples can be adjusted by reading out samples either faster or slower than their original rate
  - e.g. by reading out at 1/2 speed, pitch is lowered by half
  - this is a naive way to adjust pitch and results in significant artifacts, but is cheap to implement

- Volume modulation is just multiplication by a constant
  - multiply by 0.5 to half level

- Adding two field, **event.rate** and **event.volume** it is easy to create a sample player with adjustable frequency and volume

Frequency shifting sample player

```
- (void) fillBuffer(int length, SInt16 *buffer)
{
for(int i=0;i<length;i++)
  {
          int v = 0;
          for(SoundEvent *event in queue)
          {
                    // add in currently playing samples
                    int position = event.time - i * event.rate;
                    if(position <0 && -position < event.length)
                       v = v + event[-position] * event.volume;

                    // remove old samples
                    if(position>=event.length)
                       [queue removeElement:event];
          }
          buffer[i] = v;
  }
  // move the buffer on
 for(SoundEvent *event in queue)
          event.time -= length*event.rate;
}
```

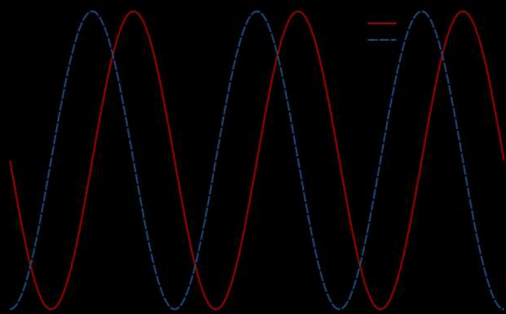
Generating tones

- Often we want to do something more interesting than just playing back pre-recorded data
  - Synthesizing audio in realtime for example
  - Signals can be generated directly as needed
- Tones can be generated with signals who have a basic period of 1/frequency of the desired tone
  - i.e. repeat (in some sense) after 1/(frequency/sample rate) samples
  - a tone is different from a noise in that it has a *harmonic* structure
  - it appears to have a clear pitch when listened to
- A tone at 261Hz (middle C on a piano) has a period of ~167 samples at 44100Hz
- Lots and lots of functions and techniques can be used to generated sounds!

Sound basics

- Most sounds have three important properties
  - pitch
    - the fundamental pitch which the sound appears to have
    - obviously some sounds are unpitched entirely
  - amplitude
    - the level (and variation in level) of a sound
  - timbre
    - the "other quality" of sound
    - woodwind vs piano, steel vs carpet

Sine wave



- The simplest, purest tone is a sine wave
  - just a single frequency
  - very easy to generate (as in the first example)
    - (computing sin(x) is quite expensive, normally precomputed tables are used)
  - v = sin((frequency*phase*2*pi)/(samplerate)
    - ranges from -1 to 1, must be scaled to fit the bit depth
    - phase is a variable that increases by 1 for each sample produced
- Lots of synthesis techniques use the idea of a *phasor*
  - Just a value which increments at each sample
  - The increment is by frequency/(sample rate)
  - Increases by 1.0 every period
    - v = sin(phasor)
    - phasor += (2*pi*frequency)/samplerate

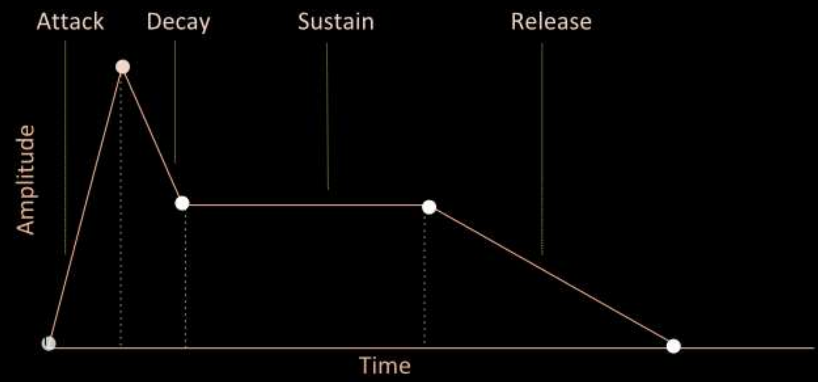
Envelopes

- One of the key aspects of a sound is the way the amplitude changes over time

- Most sounds become rapidly loud, then become quieter
  - The characteristic shape is very important

- The **envelope** of a sound is its amplitude profile

- Often described in terms of
  - **attack** time (increase at start)
  - **decay** time (decay to steady state)
  - **sustain** level (volume while sustaining)
  - **release** time (time to go back to silent)



- A flute has a slow attack and high sustain

- A drum has a very fast attack, no sustain and slow release

Use of envelopes

- Often envelopes are used to modify the amplitude of sounds
  - an envelope can be multiplied by a sample for example, to impose that envelope on to it
- Envelope generators just produce slowly varying sample patterns according to an envelope definition
- Often used for other parameters in synthesis
  - for example, the "brightness" of a sound can be defined by an envelope
  - lots of sounds are very bright in their attack and then become less bright
  - brass instruments have the opposite envelope (brighter after attack)
- Envelopes vary slowly over a range of seconds, rather than the fast oscillations of tone generators

Synthesis types

- There are many common synthesis types, including:
  - **Wavetable synthesis**
    - sample playback, usually with sample layering and pitch shifting
    - widely used in electronic instruments (e.g. for acoustic instruments)
  - **Subtractive synthesis**
    - generates tones with very basic tones and then *filters* them
    - most explicitly electronic-sounding instruments use this principle
  - **FM synthesis**
    - generates tones by modulating phase of a sine wave by another sine wave
    - flexible and powerful, widely used in the 80's...

Synthesis types

- **Physical modelling synthesis**
  - simple physical models of real systems (airflows in tubes, vibrating strings)
  - realistic and expressive, but computationally intensive
- **Granular synthesis**
  - uses large numbers of very short fragments of sampled sound
  - sound is defined by probability distributions over parameters
- **Distortion Synthesis**
  - generalisation of FM, includes things like wave shaping, phase distortion and DSF synthesis
  - excellent for generating new, artificial timbres and can be expressive
  - computationally efficient

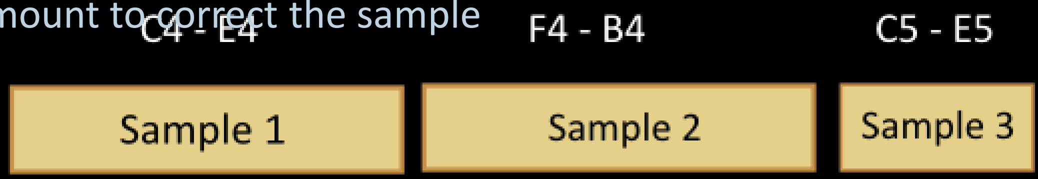
Wavetable

- Wavetable synthesis is just playing back samples
  - Recordings from a real instrument or object are played back
  - Pitches are matched to desired pitches by pitch shifting
- The code given previously is sufficient to implement a wavetable synthesizer
- The amplitude of the waveform can be adjusted, often with an envelope, so that different amplitude patterns can be achieved
- Wavetable synthesizers sound very realistic (because they are samples)
  - But not very expressive, because there are very few ways to modulate them
- Usually the pitch, an amplitude envelope and a simple filter are used to modulate the raw samples

Sample layering

- Pitch shifting samples sounds bad if the shift is more than a few percent
  - length of sound changes, and fixed resonances shift unnaturally

- Many wavetable synthesizers use many samples of an instrument, at different pitches
  - choose the sample nearest to the desired pitch
  - then pitch shift a small amount to correct the sample



- This is quite memory intensive though
  - some piano synthesizers use multiple *gigabytes* of samples!
  - every key sampled at many levels of velocity

- Other variations might be recorded
  - playing hard vs playing gently
  - again, closest sample is selected, and then amplitude adjustment is used to fill in levels

Subtractive

- Subtractive synthesis is the (digital emulation of) the techinques used in early electronic instruments such as Moog

- Use a few simple signal generators to create basic tones
  - Sine waves, saw waves, square waves...
  - Frequency and ampltiude of tones can be enveloped

- These signals are then filtered using digital filters
  - e.g. lowpass filters to remove high frequency content

- Most "electronic" sounding instruments use subtractive synthesis
  - e.g. extensively used in dance music

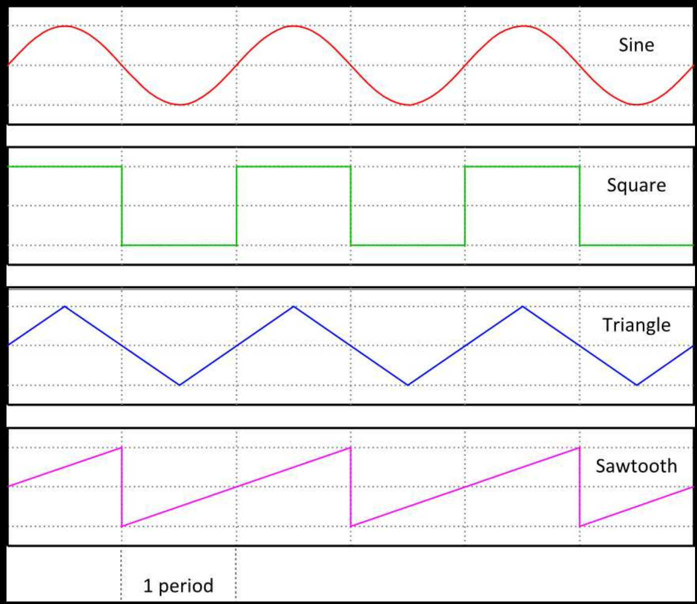
- Making good sounding subtractive synthesizers is actually really hard in the digital domain, because the analog techniques are tricky to emulate without artifacts

Waveform generation

- Simple "classic" waveforms are used
  - Originally used because they are easy to generate in analog hardware

- Traditional waveforms are sine, square, saw and triangle

- Square, saw and triangle are very rich in harmonics
  - i.e. lots of high frequency content



- Other waveform types, such as white noise, are also used
  - computationally simple but frequency rich
- These harmonics can be filtered to produce interesting sounds

Digital Filters

- Filters are used to "sculpt" the sound by removing frequency
  - Lowpass filters remove high frequencies
  - Highpass remove low
  - Bandpass just keep frequencies in a particular band

- The filter cutoff frequency can be adjusted throughout the sound
  - e.g. letting through lots of high frequency at the start of a sound and then cutting it down
  - usually modulated with an envelope

- Interesting filters are usually *resonant*
  - enhance frequencies near the cutoff frequency
  - resonant filters are the characteristic "analog synthesizer" sound
  - filters often resonate so much they go into oscillation

- Although simple digital filters are easy to implement, making good sounding filters is hard
  - especially since analog versions often have significant non-linearities...

Simple lowpass/highpass filter

- A very simple "one-pole" lowpass filter is given by
  - y(t) = alpha*y(t-1) + (1-alpha)*x(t)
- A corresponding highpass filter is just
  - z(t) = x(t) - y(t)
- *alpha* can be set to produce a given cutoff frequency
  - alpha = exp(-2*pi*frequency / sampleRate)
- This can't resonate though...
  - One that can is the State Variable Filter, which also sounds pretty good (few digital artifacts) (see next page)
- Filters can be cascaded or run in parallel for richer modulations
  - e.g. a bank of bandpass filters can be used to simulate a set of resonances

State Variable Filter

- From musicdsp.com, originally from "Effect Design Pt. 1", J. Dattorro, J. Audio Eng. Soc., 45:9 1997

```
cutoff = cutoff freq in Hz
fs = sampling frequency //(e.g. 44100Hz)
f = 2 * sin (pi * cutoff / fs) //[approximately]
q = resonance/bandwidth [0 < q <= 1]  most res: q=1, less: q=0
low = lowpass output
high = highpass output
band = bandpass output
notch = notch output

scale = q
low=high=band=0;


//--beginloop
low = low + f * band;
high = scale * input - low - q*band;
band = f * high + band;
notch = high + low;
//--endloop
```

FM synthesis

- **Frequency modulation synthesis** is a simple technique for generating complex waveforms with minimal computation
  - It is also the sound of the 80's due to the popularity of the Yamaha DX7!

- Idea: take a sine wave, and modulate its frequency with another sine wave
  - When done slowly sounds like vibrato (frequency wobble)
  - When done quickly, changes character (timbre) of the sound

- In practice, true frequency modulation can run into nasty problems
  - **phase modulation** is used instead

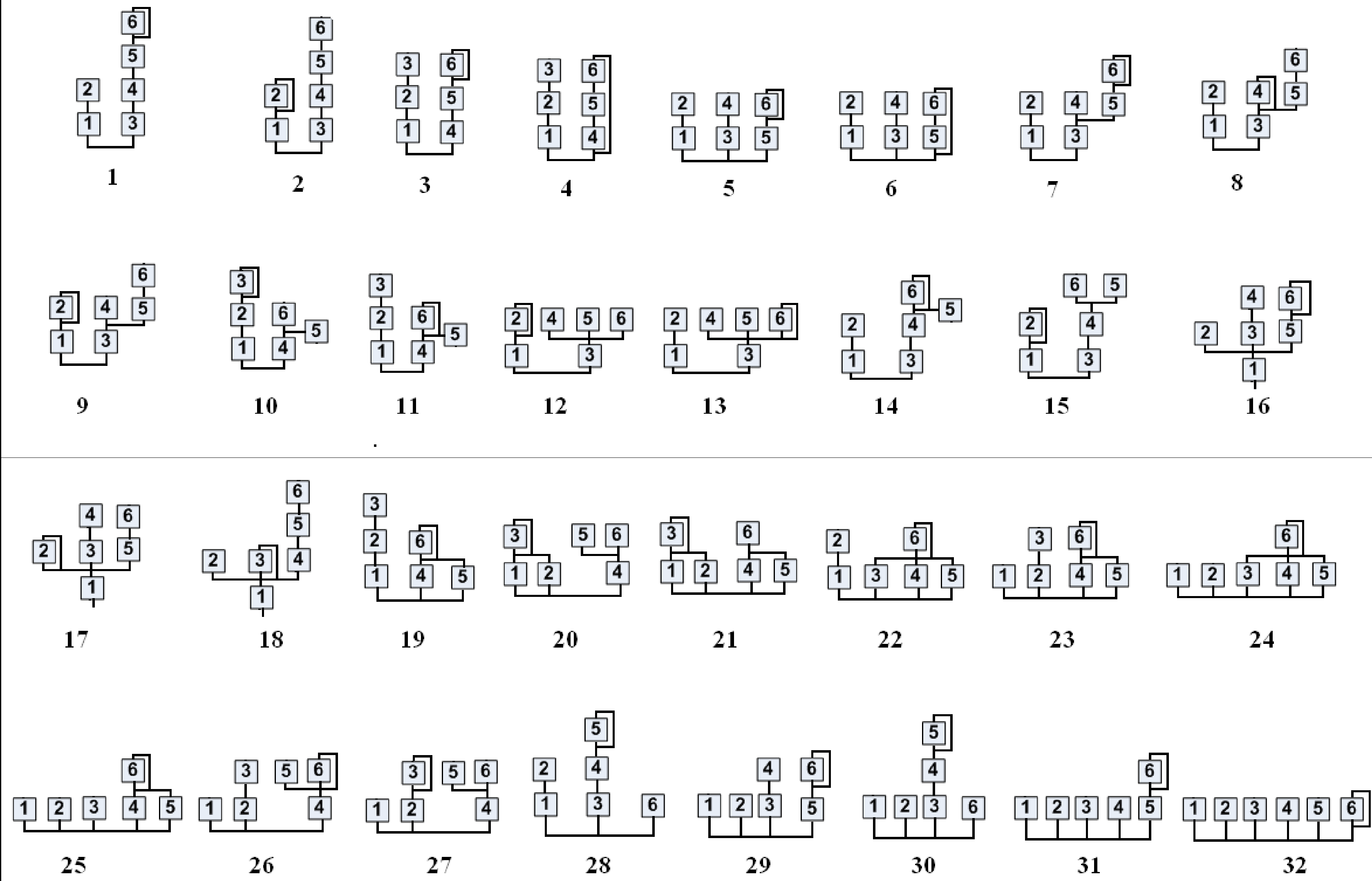
- Simple formula:
  - v = sin((phasor1 + sin (phasor2) * modulation))
  - **phasor1** and **phasor2** run at different frequencies
  - **modulation** specifies how much the second waveform distorts the first

FM synthesis (II)

- As the modulation of the sine wave increases, the spectral richness of the signal increases
  - more high frequency components
  - if the modulator:carrier frequency is integer, the resulting sound is harmonic
  - if it's not, the result is *inharmonic*
    - this is hard to achieve with other methods
    - excellent for bell sounds, where inharmonicity is important

- More complex sounds can be created by combining units together
  - one FM unit can be the modulator of another unit, replacing the basic sine wave
  - multiple FM units can be cascaded or run in parallel

- Classic instruments like the DX7 had 6 "operators" (sine wave synthesizers) which could be arranged in different patterns
  - other synthesizers have used 4 or 8 operators


1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32

FM Synthesis (III)

- FM can produce a wide variety of sounds
  - very "sharp" compared to traditional analog synthesis
    - lots of high frequency components
  - sometimes said to have a "plasticy" tone
- Using envelopes to modulate the frequency and modulation index of the different "operators", rich changes in timbre can be created
- Extremely efficient
  - Just needs a sine table lookup
  - No need for any floating point computations
  - Earlier synthesizers used log/exp tables so that envelope modulation could be carried out without even using multiplies!

Physical models

- Physical modelling synthesis tries to model the actual physics of an instrument or object
  - For example, modelling a flute by simulating the flow of air inside the flute

- These models are necessarily very simplified
  - accurate model of airflow in a flute would be extremely complex
  - could never realistically be performed in realtime
  - usually involve delay lines to model one-dimensional waves
  - filters and nonlinear elements are used to interconnect these "waveguides"

- Physical modelling can be very expressive, because the parameters of the simulation can be modulated in natural ways and many types of stimulation can be applied
  - e.g. simulating a snare drum which responds to where and how hard you hit it
  - might allow brush strokes as well as stick hits
    - just a change of input

Delay Lines

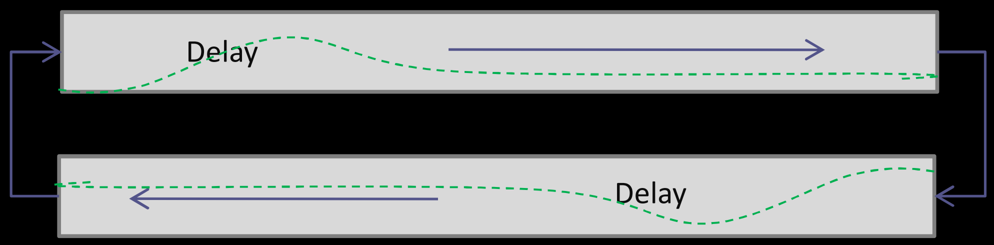
- Much of physical modelling synthesis extensively uses *delay lines*
  - A delay line just delays a signal by a certain number of samples
  - A length n delay line takes **x[t]** and returns **x[t-n]**
  - This can implemented very efficiently using just an array of samples

- By feeding back the output of a delay line back into itself, a recirculating delay line can be produced
  - This resonates at a frequency given by the length of the delay line

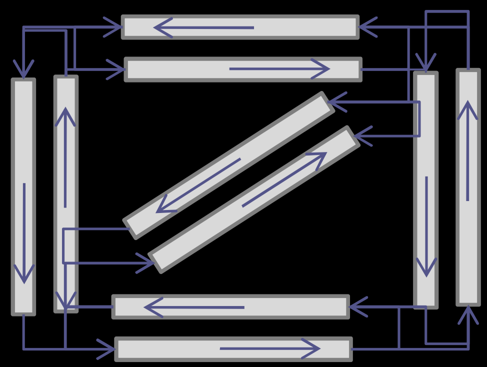
- Multiple delay lines can be linked together
  - Filters and other elements can be introduced into the linkages to simulate mechanical effects
  - loss of energy, or high frequency damping

Waveguide

- A waveguide is a simple model for one-dimensional wave propagation
  - Consists of a pair of delay lines, one running in each direction



- Different topologies of waveguides can be connected together
  - e.g. a simple drum head can be constructed like this:



Losses

- Real wave propagation involves losses
  - waves do not recirculate forever
- This can be simulated with simple damping
  - multiplying the output of each delay line with a constant < 1.0 before passing it into the other delay line
- There are also frequency dependent losses
  - high frequencies decay faster than low frequencies in real physical systems
  - putting a lowpass filter at the delay line junction simulates this property
    - lowpass filter must have a total gain of 1.0 or less, otherwise energy will increase!

Impulses

- To actually "play" a waveguide, energy must be injected
- Impulses are introduced into the delay lines
  - these then recirculate, gradually decaying due to the modelled losses
- Simple impulses can just be a single sample with a large value (a spike), or a short burst of white noise
- More complex impulses can be used
  - for example, extracting impulse models from real instruments
  - modelling a guitar's pick

Fractional Delay Lines

- Note that we often need delays with non-integer sample lengths
  - Otherwise, for example, notes will be out of tune!

- e.g. if you want a delay line which resonates at 1808Hz at 44100Hz sampling rate, it would need to be 24.391 samples long
  - 24 sample long delay line is 1837Hz -- this is very significantly out of tune!

- Special filters can be used to simulate delays of 0.0 - 1.0 samples
  - Lagrange filters, allpass filters
  - One of these is applied after the integer delay line to correct the tuning

- You will implement a simple fractional delay line as part of the lab tomorrow

Plucked string model

- A very simple plucked string model was developed by *Karplus and Strong*

- A delay line recirculates (feeds back), with
  - damping (reducing amplitude over time)
  - and some filtering (reducing high frequencies over time)

- This simulates the signal propagating up and down the string, losing energy at its termination points
- The string is plucked simply by filling the delay line with random values
  - This is very crude, but sounds surprisingly good