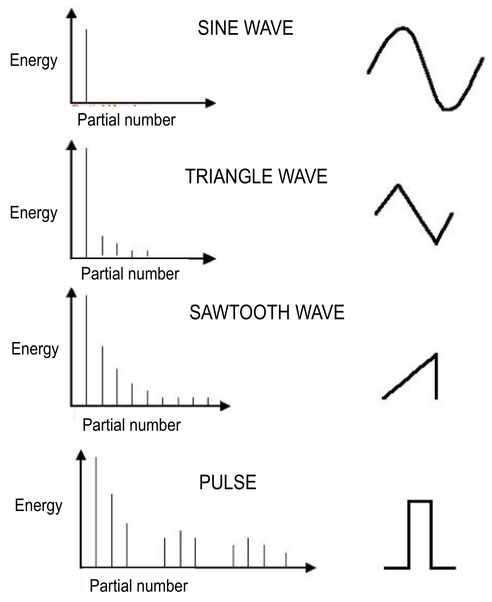
Music 220A – Homework 1 – Lab, Part 2: Recreating the Original Deep Note

If you've ever watched a movie in a movie theater, chances are that you are familiar with Deep Note, the audio logo of THX. That sound is one of the first sounds we hear at the beginning of movie trailers in a THX-certified venue. Created by Andy Moorer (Dr. James Andy Moorer, one of CCRMA's founders who was a PhD student in CS) in 1982, it not only one of the most recognizable computer-generated sounds, but a great example of additive synthesis. It was created using around 30 different oscillators (the original waveform used was a digitized cello tone), which are tuned and modulated to different random frequencies in the first portion of the sound (beginning in the range 200-400 Hz). Shortly thereafter, they modulate to stacked up octaves of the fundamental sound, fusing into a unified sonic object, before fading out.

In this lab, we will walk through the creation of (a version of) this sound in ChucK.

The first consideration is what kind of waveform should we use as our building block. Given that Dr. Moorer's original sound was a cello sound, which has a rich spectrum, our building block should similarly have a rich spectrum. The basic oscillators have the following waveform shapes (on the right), and frequency components (on the left). Sawtooth wave may be a good choice for an oscillator, though you are welcome to try others. The sawtooth oscillator in ChucK is `SawOsc`.



There are many, *many* different ways to write this in ChucK, but we might approach the code in the following way: creating a process, or a *function* by which a single `SawOsc` is instantiated, and its frequency and gain is controlled is over the entire duration of the THX sound. We can ensure that the oscillator begins at a random frequency, but moves to one of a set of predefined frequencies in the final stacked octave chord. We then have 30 of instantiations of the function running at the same time.

Regarding the timecourse of the sound, we will handle it in four parts:

1) a steady period in the beginning, at which the sound is steady (i.e. the **initial hold**)
2) the frequency **sweep**, during which each oscillator moves from its initial frequency to its final frequency
3) a period where the target sound is held, i.e. the **peak hold** and a
4) **decay** at the very end of the sound.

Since all of our oscillators have the same general timing instructions, the same general frequency instructions (that the frequency changes during the 2nd time period), and the same general gain change, we will declare the timing of these four sections as *global variables* – variables that are accessible to all functions. We'll see how this plays out in a second.

Let us declare the timing of each of the four sections:

```
// time for each component of the sound
800::ms => dur initialHold;
3::second => dur sweep;
```

```
3::second => dur peakHold;
2::second => dur decay;
```

Feel free to change to tweak the timing! Note that we are creating variables for these parameters, instead of hard-coding numbers into this upcoming section of code. This is so that you can easily change just these numbers above, and then that will then permeate every section of your code. It helps you avoid having to go through code and change numbers in multiple places.

Next, let's determine how we are going to change the frequency of the SawOscs. There are a few different ways you can do this, but one of the ways that ChucK allows you to do this is by directly controlling the frequency of the SawOsc at very short timescales – even down to the level of the sample. During the second section of the sound, we are going to update the frequency of the SawOsc very quickly, say, every sample. The actual amount of the frequency change will be different for every SawOsc, since each will have a different initial and target frequency, however, we will update each at the same rate.

A question that remains is, given how long the "sweep" section of the sound is, how many times will we need to update the frequency? We can actually accomplish this in a very succinct line of code. We divide the length of time of the sweep by the amount of time in a sample (at 48000 Hz sampling rate):

$$\frac{3 :: second}{\frac{1}{48000}}$$

Given that the result of this equation may turn out as a float, and for reasons to be seen, we will need it as an int, we *cast* it as an int, meaning that we have converted the type of object from a float to an int. In ChucK, this is accomplished with `$ int`. Take a moment to understand the following line of code, as it is an example of how multiple steps in programming can be accomplished in just one line.

```
// number of steps needed to update the frequencies every sample during the sweep
(sweep/samp) $ int => int sweepSteps;
```

We can follow the same model for the end of the sound, in which, instead of changing the frequency, we're changing the gain in small increments, fading it out over the duration of the fourth part of the sound. So, we add another line to the chunk of code above:

```
//we want to update the frequencies every sample during the sweep and decay
(sweep/samp) $ int => int sweepSteps;
(decay/samp) $ int => int decaySteps;
```

Okay, and before we get into writing the function, there is one last thing we need to decide – what the frequencies will be of the climactic chord that we are moving to. Since we need the oscillators to arrive at a predefined set of frequencies, we will declare them outside of the main processing function, in an array. While we have discussed frequencies, sometimes it is easier to think in pitches. Computers don't easily represent the actual note names, but numbers are easier. There are plenty of charts of MIDI note numbers online for you to get familiar with, but a taste would be that middle C, or C4, is represented as 60. C4# as 61, D4 as 62, etc. ChucK can eventually convert these numbers into frequencies for us, so for now, we will think in MIDI note numbers.

While we could just list out the values that we want in an array, this wouldn't be flexible with, say, changing the number of oscillators to something other than 30. So, we develop an algorithm to populate the array with a set of C's and G's from a wide pitch range, that is sensitive to potentially changing the number of oscillators. The loop below declares two counter variables – one of which ascends to `nOscs` (`i`), and one of which loops between the values 0 and 7 (`j`). For the oscillators corresponding to even numbers, the pitch assigned is some flavor of C, beginning with MIDI note 0. This is achieved in the first line, in which we use a handy, succinct technique called the 'modulo' operator (%), which returns the remainder of a division process in which whatever is on the left of the operator is divided by what is on the right (i.e. to the left is the numerator, and to the right is the denominator). Here, if we always have a demoninator of 2, and step through integers 0-29 on the left, we will always either get 0 or 1 in alternation as a result of this operation. Thus, `i%2==0` returns TRUE or FALSE – when TRUE, a C MIDI pitch is assigned to our array. When it is false, corresponding to the odd oscillators in the next line, the MIDI pitch assigned is some flavor of G. Since we don't want to assign MIDI pitches that get too high, by the time we ascend through 8 octaves, the value of `j` becomes 0 again and we start assigning MIDI pitches in the bottom of the range. While there may be more elegant ways to achieve this result, this one works just fine for our desired purpose.

```
//a process that generates the target frequencies
//based on MIDI note numbers
//this one starts on C-1 (MIDI pitch 0)
//and includes the 5th (G-1, MIDI pitch 7)
30 => int nOscs;
int targetPitch[nOscs];
0 => int i;
0 => int j;
while( i < nOscs )
{
    if ( i%2==0 ) 0+(j*12) => targetPitch[i];
    else 7+(j*12) => targetPitch[i];
    if ( j < 8 ) j++;
    if ( j == 8 ) 0 => j;
    i++;
}
```

Note: A different ChucK operator can be used if all of the values are being plugged into the array upon its declaration. If we were defining all values of `targetFreq[]` at once, instead of being declared while empty, the chuck operator used is @=>.

Since the frequencies at the beginning of the sound are going to be random, we don't need to predefine them; we will simply declare a random frequency for each one of our oscillators in the function.

So, let's get down to writing the function. In a nutshell, a function is a named procedure that does a distinct task. It is a subroutine that you write and declare in code, and then 'call', or run when needed. Here, we will write a function that instantiates a `SawOsc`, controls the frequency and gain changes it undergoes, and controls the time course of its life. In this way, we will eventually layer 30 instantiations of this file together in the same moment, creating the THX sound. ChucK's syntax to declare a function begins with the word '`fun`', followed by a return type (used if you want your function to *return*, or 'spit out', an object – when you don't, you write `void`), followed by the name of the function (what you will call when needed) and any input arguments (what the function would need to carry out operations). Chapter 5 in the ChucK book deals heavily with functions, so again, refer to that, or the online documentation if this is your first time working with them.

We name our function 'sawVoice'. The first thing we might do in this function is to declare the one SawOsc the function will control. We will eventually change this, but for now, we'll plug it into the dac for simplicity.

```
fun void sawVoice( )
{
	// instantiate and connect SawOsc
	SawOsc s => dac;
}
```

Our oscillator needs to know its starting frequency and target ending frequency. Let's also define these two input parameters.

```
fun void sawVoice( float initialFreq, float targetPitch )
{
	// instantiate and connect SawOsc
	SawOsc s => dac;
}
```

Now that we know that information, we need to attend to some unfinished business. Before, we calculated how many steps it would require to move our oscillator from its initial frequency to its target frequency, but now that our function has access to those values, it can calculate by how great of a frequency change, the 'delta', that that SawOsc has to move at each step. We add that to the function below. We also add one more input parameter, the gain of the SawOsc, and set the initial frequency of the oscillator. (Even though we've passed in the value as a float, we need to still set the frequency of the actual SawOsc!)

```
fun void sawVoice( float initialFreq, float targetPitch, float gain )
{
     // instantiate and connect SawOsc
     SawOsc s => dac;
     // compute delta freqency between successive updates
     (Std.mtof(targetPitch) - initialFreq) / sweepSteps => float delta;
     // set initial gain
     gain => s.gain;
     // set freq
     initialFreq => s.freq;
```

Note: The command Std.mtof(targetPitch) above refers to a library of functions, some of which deal with MIDI number to frequency conversion. So, 'mtof' stands for 'MIDI to frequency'. Likewise, there is a Std.ftom() function.

Okay. At this point, we have everything that we need to lay out the timecourse of how the oscillator is changing -- the function knows the initial frequency, the target frequency, the gain of the oscillator, and the size of the frequency from the initial to the target. Recall that the durations of time that ChucK needs to advance through the sound, (by advancing time through the 'now' construct), were declared outside of the function. Our function still has access to those values. We will move through the four parts of the sound below:

```
    // steady cluster
    initialHold => now;
```

Hopefully the above is clear! By 'ChucKing' the duration of the initialHold to now, we advance time by that amount. There are many, different methods by which to control time in ChucK. You could have ChucK complete some sort of computational process for a certain amount of time, by declaring a future time, later than the current time, and say that, until ChucK reaches that time, that the process should be carried out. That is how we will approach the frequency sweep portion of the THX sound.

```
    now + sweep => time endOfSweep;

    // sweep freqs
    0 => int j;
    while( now < endOfSweep )
    {
        initialFreq + (delta*j) => s.freq;
        1::samp => now;
        j++;
    }
```

Similarly, we hold the oscillator once it has reached its target frequency, and then decay it to 0 until the end of the sound.

```
    //hold chord
    peakHold => now;

    now + decay => time endOfDecay;
    0 => j;

    // decay
    while( now < endOfDecay )
    {
        gain * (decaySteps-j) / decaySteps => s.gain;
        1::samp => now;
        j++;
    }
}
```

Alright! Our function is practically done! The last step is to handle *where* the sound is going to be going. Now, we could send all of the oscillators to one channel (mono), but since we'll probably be listening through headphones in stereo (with a left and a right channel in our headphones), we might as well put half of the oscillators out on the left side and half out on the right, for a slight spatialization effect. We create a reverb for each of the channels we're sending to, and connect the reverbs to the dac.

```
//Number of channels - set to two for stereo out
2 => int nChannels;
//creation of reverbs of number of channels
```

```
JCRev r[nChannels];

// connect channel reverbs to dac channels
for( int i; i < nChannels; i++ ) {
      r[i] => dac.chan(i);
      // ‘turn down’ the reverb
      0.025 => r[i].mix;
}
```

Now, we will need to be able to tell our `SawOsc` *which* reverb it is going to be plugged into – that is specific to the oscillator. So, we will add one more input parameter to our function – the `UGen speakerNum` – which will be the placeholder for our reverb, and change the patch so that our `SawOsc` is no longer going straight to the `dac`, but to one of the reverbs that then lead into the `dac`. Thus, we will change the beginning to the function to look as follows:

```
fun void sawVoice( float initialFreq, float targetPitch, float gain, UGen
speakerNum )
{
     // instantiate and connect SawOsc
     SawOsc s => speakerNum;
```

Now, what we need to do is instantiate 30 of the functions at the same time, and play them concurrently, with the proper inputs.

For our target frequency input, we already know that we are going to set each one of the oscillators to one of the values in `targetPitch[]`. However, our initial frequency is random, but constrained to a range, so we are going to use the following command to get a random frequency between 200-400 Hz: `Math.random2f( 200.0, 400.0 )`. Our gain, we need to be careful about. The default gain is .5***, and if we created 30 oscillators, all with that high of a gain, we might put our ears in danger! We keep our gain low, setting the gain of each oscillator to 1/30. And finally, the last piece of the puzzle, is setting which reverb and thus which channel – the right or the left – each oscillator is going to. We use the modulo operator again to alternate between the numbers 0 and 1, alternating between the two elements in the (small) array of reverbs we have, which our function then hooks our `SawOsc` up to. Putting that all together:

```
// for each voice
for( int i; i < nOscs; i++ )
{
    // spork each voice
    spork ~ sawVoice( Math.random2f( 200.0, 400.0 ), targetPitch[i],
                      1.0/nOscs, r[i%2] );
}
```

Note: **Math** is a standard library in ChucK, which includes utility functions such as random number generation, unit conversions, and absolute value. The function we are using, `Math.random2f`, generates a random, floating point number between our first and second parameters, 200.0 and 400.0.

This thing called ‘`spork`’ used above is a way to tell ChucK to run a process (which, once its running, becomes a ‘shred’), and continue to run other processes. So, the above for loop sporks 30 shreds at once, each one of our 30 SawOscs.

To record the output of a ChucK file to a .wav, include the following in your code before sporking all of the sawVoice() functions:

```
// write to a file
dac => WvOut2 out => blackhole;
me.sourceDir() + "/LASTNAME_THX.wav" => string _capture;
_capture => out.wavFilename;
```

Finally, there is one more very important line. We must tell ChucK to keep ‘master time’ running while the sporked functions continue to run:

```
// wait for sequence to finish
initialHold + sweep + peakHold + decay => now;

out.closeFile();
```

Your deliverable is a .wav file of your THX sound, to be labeled LASTNAME_THX.wav, as in the code above.