Chuck Filter Lab
PAT 462

Subtractive synthesis and filters are some of the fundamental building blocks of sound design – computer musicians use them in almost every piece of music they create. In this lab, we will walk through three examples of how you might use filters in ChucK.

Example 1: simulating ocean waves
Example 2: filtering audio files
Example 3: chip tunes

In each example, we follow our general template of the 'clip' – that each of these processes can be predefined to 'live' in ChucK for a certain period of time, in a very controlled manner, using `spork`.

Example 1 - Simulating Ocean Waves

Waves are a noisy sound, with higher frequencies being more prominent during the crash of the waves, and lower frequencies being more prominent at the waves' lowest point. Here, we will simulate them by using noise as an input to one of the filters that ChucK has in its arsenal, and sweeping the cutoff of the filter at the rate at which waves typically roll and crash.

First, we define our 'clip' function, connect some noise to a filter, and then to the `dac`. While we could use any of the filters that ChucK has in its arsenal (the lowpass, `LPF`, the highpass `HPF`, and more), we choose `ResonZ` as it has a constant gain at the true peak of the filter and thus is more predictable than some other filters.

```
fun void clip(dur myDur)
{
	// noise generator, resonator filter, dac (audio output)
	Noise n => ResonZ f => dac;
```

Then, we set the parameters for our filter. Filter Q determines how "sharp" to make the resonance of the filter – this is a value to play with in designing your ocean waves. In the line following, we set the gain for the filter.

```
	// set filter Q (how sharp to make resonance)
	1 => f.Q;
	// set filter gain
	.25 => f.gain;
```

Next is to sweep the resonant frequency of the filter, done in the `while()` loop below. By sweeping the resonant frequency, we oscillate between hearing mostly low and then mostly high frequencies. Note some lines in the middle of the following code serve as 'bookkeeping' for how long the function should actually live for when eventually run.

```
	// our variable to help smoothly sweep resonant frequency
	0.0 => float t;

	//taking care of duration
	 <<<"\tclip start at",now/second,"seconds">>>;
	now => time myBeg;
```

```
        myBeg + myDur => time myEnd;

        // timeloop
        while( now < myEnd )
        {
            // sweep the filter resonant frequency
            100.0 + (1+Math.sin(t))/2 * 3000.0 => f.freq;
            // advance value
            t + .005 => t;
            // advance time
            5::ms => now;
        }
      <<<"\tclip end at",now/second,"seconds">>>;
}
```

The bit of math above in the first line of the `while(true)` loop gets the values of the frequencies that we are sweeping into a desired frequency range. The middle section of the code, `(1+Math.sin(t))/2`, outputs values between 0 and 1, and oscillates through these values at a rate determined both by `t` and how quickly time is advancing. The rest of the numbers on the scale the values into a reasonable frequency range for the effect we want. If it is more intuitive for you to work in MIDI numbers than in raw frequency values, you could substitute that line with either of the below. Note that the function `Std.mtof` from the Standard library in ChucK maps midi values to frequency values. (Our filter will only accept frequencies, thus if we want to conceptualize our work in pitches, we must translate our pitch values to frequencies by means of the `Std.mtof` function.)

```
24 + (1+Math.sin(t))/2 * 48 => Std.mtof => f.freq;
// same effect as the line above
Std.mtof(24 + (1+Math.sin(t))/2 *48) => f.freq;
```

Finally, we include the code we need to call and run the function:

```
// TIME 0, start the clip
spork ~clip(10::second); // launch clip in independent shred
10::second => now;       // advance time so the clip can play

// last item in this program is this print statement
<<<"program end at",now/second,"seconds">>>;
// and with nothing left to do this program exits
```

Example 2 - Filtering Audio Files

Chapter 4 in the ChucK book, *Programming for Digital Musicians and Artists,* rather extensively covers the capabilities that ChucK has regarding sound file manipulation (e.g. playing a sound file backwards), and it is suggested that you read that chapter for an overview. Within the context of filtering here, however, we will load in the sound and filter it, this time using a *lowpass filter*, or `LPF`.

First, let's just practice getting a sound file playing in Chuck.

```
fun void clipSndBuf(dur myDur)
{
	// declare the object which will store the sound file, and pass it through to
	the dac
	SndBuf buffy => dac;
```

To load in your own sound file, you could use the following command:

```
	me.dir()+"/SUBFOLDER/FILENAME.wav" => buffy.read;
```

in which the first part of the command, `me.dir()`, gets the directory where the .ck file is saved (so, save your script before running), and rest gets the full filename an stores it in our `SndBuf` object (through the `.read` member of `SndBuf`). In this example, we will use one of the test sounds stored in ChucK.

```
	// load a sound (in this case, a internal test sound)
	"special:dope" => buffy.read;
```

This file is short. Before we manipulate the sound using filters, let's repeat it a lot so that when we change the filters, the filter effect will be audible over time. We are going to delegate this repetition to a function, 'trigger', which also slightly varies the sound each repetition (so it doesn't sound *too* repetitive. Note this is declared outside of the function we have been writing thus far!

```
// trigger the sndbuf
fun void trigger( SndBuf buf, float pitch, float velocity )
{
	// set pitch
	pitch => buf.rate;
	// set velocity (really just changing gain here)
	velocity => buf.gain;
	// play from beginning
	0 => buf.pos;
}
```

The above function expects a `SndBuf` (called `buf` locally within the function), and two float values that then are passed to the playback rate (here, effectively the pitch) and the gain of the `SndBuf`. It then sets the playback position of the `SndBuf`. So, if we call this function in a loop and allow time to pass, the `SndBuf` will be triggered and played (in this case, with randomly variable pitch and a constant gain of 1), every T seconds.

```
	// some length
	500::ms => dur T;

	//taking care of duration
	<<<"\tclip start at",now/second,"seconds">>>;
	now => time myBeg;
	myBeg + myDur => time myEnd;
	//time loop
	while (now < myEnd)
	{
		// play sound
		trigger( buffy, Math.random2f(.9,1.1), 1 );
		// wait
```

```
        T => now;
    }
    <<<"\tclip end at",now/second,"seconds">>>;
}
```

Note that by passing in the SndBuf as an input, our function could, for example, trigger multiple SndBuf (or the same SndBuf) at overlapping times as it makes a local copy of the SndBuf and operates on that. If we did not pass the SndBuf object in, rather internally operated on our SndBuf buffy, and made multiple overlapping calls to our `trigger()` function, the SndBuf would not be able to finish playing since the function yanks the playback position to 0.

By including the below lines, we would get a (slightly) varying Homer Simpson ‘doh’ for ten seconds.

```
// TIME 0, start the clip
spork ~clipSndBuf(10::second); // launch clip in independent shred
10::second => now;      // this master shred needs to remain alive while it's
playing

// last item in this program is this print statement
<<<"program end at",now/second,"seconds">>>;
// and with nothing left to do this program exits
```

Now, let’s make it more interesting by sweeping a filter over our sound. We must initially set the filter cutoff as well as a resonance frequency. Recall that a *low pass cutoff* represents a frequency value is between 0 Hz and the Nyquist frequency (the highest representable frequency). Setting a cutoff frequency close to 0 will result in very little sound. Regarding the *low pass resonance*, as this value increase, a resonance at the cutoff frequency is boosted.

Food for Thought: By increasing the gain of a lowpass, and then sweeping the cutoff over a harmonic sound, what will happen to the various harmonics in the original sound as the cutoff frequency passes?

We pass the original file through the LPF (you need to change the first line!) and add the following lines to the inside of our `clipSndBuf()` function:

```
    SndBuf buffy => LPF lpf => dac;
    // set filter cutoff
    4000 => lpf.freq;
    // set resonance at cutoff frequency
    10 => lpf.Q;
```

To achieve a smooth sweep of the filter, we must update the cutoff frequency very frequently – much faster than once every 500 ms as in the other `while()` loop. While we could update the filter in the above while loop, we will write a separate process (called a shred in ChucK) to update the filter and use ChucK syntax to run that process in parallel on a fine-grained timescale. We will call it `updateFilter()`, and declare it outside of our `clipSndBuf()` function.

```
// entry point for parallel shred
```

```
fun void updateFilter( LPF lpf )
{
    // infinite time loop
    while( true )
    {
        // compute filter frequency
        400 + (Math.sin(now/second*1)+1)/2*3000 => lpf.freq;
        // advance time (also update rate)
        5::ms => now;
    }
}
```

Though we are declaring it outside of our main function, we call it from the inside of that function. Thus, we needed to 'pass in' the `LPF` so that our filter function is updating the filter that is only declared within the scope of the main function. Thus, we add the following line to before the `while()` loop in our `clipSndBuf()` function:

```
      // spawn a parallel shred
      spork ~ updateFilter(lpf);
```

Note: For the most part, in ChucK, it doesn't matter a whole lot where on the page you declare functions and variables, though when sporking functions, or "sporking shreds", they must be sporked *before* entering any while(true) loop that is not encapsulated in a function. If they are not, when the code is executed, the while(true) loop will be entered, and the shred (if sporked below the while loop), will never get to run. Therefore, if we place the line that sporks the function sometime above our while(true) loop, it will successfully run in the background the entire time the program is running (or until the shred itself exits). To read more about concurrency in ChucK, check out Chapter 8 of the ChucK book.

If you run the example again, the filter should be slowly sweeping over multiple versions of Homer's 'doh'!

Example 3 - Chip Tunes

The final example here uses a function to trigger individual notes being played, and results in a cool melody. We start with a rich-timbred oscillator, the `SqrOsc`, to hear a maximally strong effect of the filter, and after Chucking it through the lowpass filter, we pass through an envelope that shapes the `SqrOsc` into notes. We will sweep the filter in the same exact way as the previous example (through a sporked shred), and have a 'play' function that controls the envelope on the `SqrOsc`.

We initially set the ADSR parameters (attack, decay, sustain and release) of the envelope using `env.set()`. Notice that of the four input parameters in `env.set()`, three of them are times (AD and R), but S is a gain value, so it takes a float rather than a duration.

Otherwise, the beginning of the code is much the same as the previous example:

```
fun void clipSqr(dur myDur)
{
      // patch
```

```
    SqrOsc sqr => LPF lpf => ADSR env => dac;

    // set filter cutoff
    4000 => lpf.freq;
    // set envelope
    env.set( 10::ms, 5::ms, .5, 10::ms );
    // some note length
    120::ms => dur T;

    // spawn a parallel shred
    spork ~ updateFilter(lpf);


    //taking care of duration
    <<<"\tclip start at",now/second,"seconds">>>;
    now => time myBeg;
    myBeg + myDur => time myEnd;
```

And our `LPF` function is the same as well:

```
// entry point for parallel shred
fun void updateFilter(LPF lpf)
{
    // infinite time loop
    while( true )
    {
        // compute filter frequency
        400 + (Math.sin(now/second*1)+1)/2*3000 => lpf.freq;
        // set resonance at cutoff (for cool effect when swept)
        5 => lpf.Q;
        // advance time (also update rate)
        5::ms => now;
    }
}
```

Here is where it differs. To be able to controllably play a string of notes, we write a function that takes in 1) what note should be played as a MIDI value, 2) the gain of the note, and 3) how long it should be played for. Hopefully this template of a function will come in useful in your other compositions.

> Note: The last 8 lines (including comments) of the below function take care of the ADSR envelope. Note that the length of the ADS portion is handled by taking the input note length `T` and subtracting the amount of time for R, called by `.releaseTime()`, .`keyOn()`, and `keyOff()` open and close the envelope.

```
// play a note (taking care of duration)
fun void play( SqrOsc sqr, ADSR env, float pitch, float velocity, dur T )
{
    // set pitch
    pitch => Std.mtof => sqr.freq;
    // set velocity (really just changing gain here)
    velocity => sqr.gain;
    // open envelope (start attack)
    env.keyOn();
    // wait through A+D+S, before R
    T-env.releaseTime() => now;
    // close envelope (start release)
    env.keyOff();
    // wait for release
    env.releaseTime() => now;
}
```

We add a final loop to our main function, `clipSndBuf()`, to continually play a string of notes over the updating filter:

```
    //time loop
    while (now < myEnd)
    {
        play( sqr, env, 60, 1, T );
        play( sqr, env, 67, .5, T );
        play( sqr, env, 70, .6, T );
        play( sqr, env, 72, .8, T );
        play( sqr, env, 76, .3, T );
        play( sqr, env, 82, .5, T );
        play( sqr, env, 84, .7, T );
        play( sqr, env, 91, .9, T );
    }
    <<<"\tclip end at",now/second,"seconds">>>;
}


// TIME 0, start the clip
spork ~clipSqr(10::second); // launch clip in independent shred
10::second => now;      // this master shred needs to remain alive while it's
playing

// last item in this program is this print statement
<<<"program end at",now/second,"seconds">>>;
// and with nothing left to do this program exits
```

As in the last lab, you should record each of these examples as your deliverables. Feel free to tweak them if you like, though credit is given for turning in the simple audio. To record the output of a ChucK file to a .wav, include the following in your code *before you spork each of your clips*:

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

The following can appear at the end of the file, for good measure:
`out.closeFile()`

Label your files as LASTNAME_Ex1.wav, LASTNAME_Ex2.wav, and LASTNAME_Ex3.wav.