Music 220A – Homework 5 – Lab, Part 2: Panning

We will walk through two examples in this lab of ways to achieve different spatialization effects using ChucK: one in which it makes it seem like there is rain falling at different locations in your speaker setup (we are assuming a stereo, i.e. two channel, setup), and another makes it seem as if a car is passing from one side of your stereo setup to another. These scripts should prove useful when working with spatialization for your radio play! This lab will move through the code faster than previous labs, and the code is entirely finished – just turn in a .wav file of each of the sections, and follow/understand what the code is doing!

Practically, both examples and both kinds of spatialization techniques hinge on sending your sounds to both the left and right channels, but controlling the balance so that, perhaps the gain on one channel is set to a value higher than that of the other channel. When wearing headphones or listening through a stereo setup, this effect is that the sound seems to be coming from somewhere in between the two speakers, but slightly closer to one side than the other.

Simulating Rain

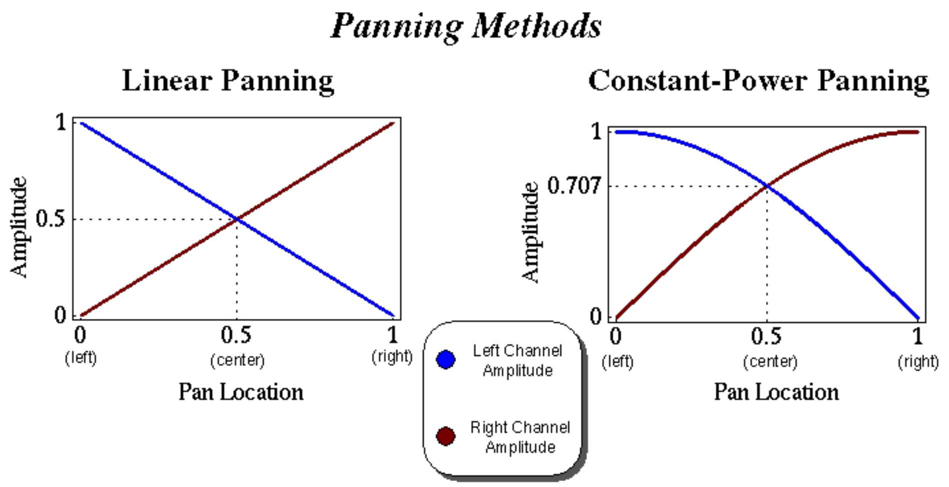
In real life, rain is falling all around you. In this lab, we will attempt to 'distribute' our rain drops over a stereo field, to simulate more closely how rain would sound in real life. First, we will simulate one rain drop, then several, and then work with two different kinds of spatialization across a stereo field.

In able to instantiate many rain drops, we declare a `RainDrop` class, and hook our rain drops up to a `Gain` on both the left and right sides. We then begin our 'clip', `rainClip()`, which will hook up our raindrops to the reverbs, which are hooked up to the `dac` on both left and right channels.

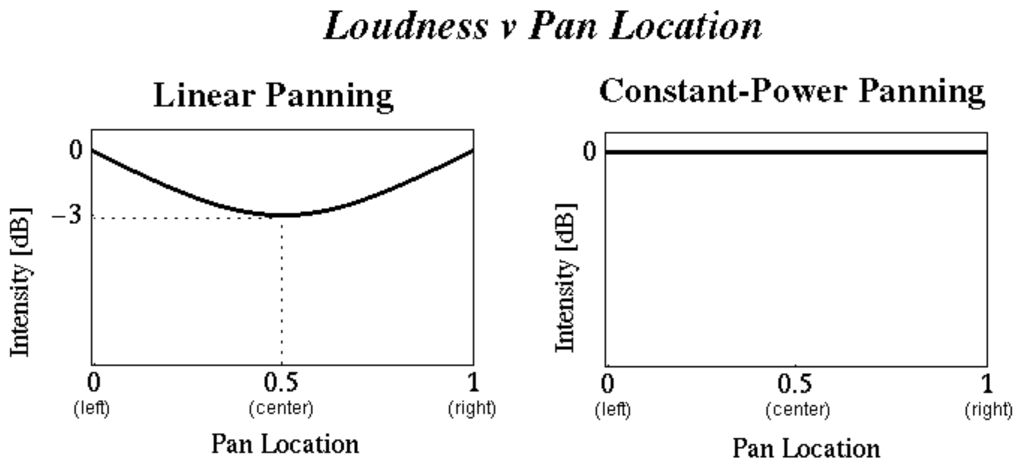
```
class RainDrop
{
    // "rain"
    Impulse rain => LPF filter;
    // connect
    filter => Gain left;
    filter => Gain right;
}

// drops per second (try changing this over time)
40 => int N;

// rain drops, so we can rotate around and give each time to
// have a tail (due to the filter) before being reused for the rain drop

// define the "clip" as a function
fun void rainClip(dur myDur, int N)
{
    // rain drops, so we can rotate around and give each time to
    // have a tail (due to the filter) before being reused for the
    // rain drop
    RainDrop rain[N];

    // reverb
```

```
JCRev rL => dac.left;
JCRev rR => dac.right;
// mix
.2 => rL.mix => rR.mix;

// connect
for( int i; i < rain.size(); i++ )
{
    rain[i].left => rL;
    rain[i].right => rR;
}
```

We will finish this clip at the end of the example. In the meantime, we then consider using two different kinds of techniques for spatialization: *linear panning* and *constant power panning.* Constant-power panning maintains a constant loudness for any valid location in the stereo field. This type of panning requires that the sum of the squares of the amplitudes for each output channel remains constant. Linear panning, on the other hand, keeps the sum of the amplitude envelopes for each output signal constant. This results in a drop in loudness towards the middle of the stereo field. Here are the amplitude envelopes as functions of pan location for each type of panning method:



and here are plots showing the dip in the total loudness of both output channels for the different panning methods:



Images from: http://hummer.stanford.edu/sig/doc/classes/Pan/overview.html

For this example, we want to be able to try both panning techniques. We declare two functions, one that will perform linear panning, and the other will perform equal power panning. Eventually, either one of these functions will randomly place a raindrop randomly in the stereo field (through the `float pan` value, which will take a random number between 0 and 1).

```
// linear panning, expect pan=[0,1]
fun void panLinear( float pan, UGen left, UGen right )
{
    // clamp to bounds
    if( pan < 0 ) 0 => pan;
    else if( pan > 1 ) 1 => pan;

    // set gains
    1-pan => left.gain;
    pan => right.gain;
}

// constant power panning, expect pan=[0,1]
fun void panPower( float pan, UGen left, UGen right )
{
    // clamp to bounds
    if( pan < 0 ) 0 => pan;
    else if( pan > 1 ) 1 => pan;

    // set gains
    Math.cos( pan * pi / 2 ) => left.gain;
    Math.sin( pan * pi / 2 ) => right.gain;
}
```

We also implement a function that allows us to switch easily between the two, based on a global variable for clarity.

```
// set a global value
0 => int LINEAR;
1 => int CONSTANTPOWER;

// pan, which=0:linear|1:constantpower
fun void panning( int which, float pan, UGen left, UGen right )
{
    if( which == LINEAR ) panLinear( pan, left, right);
    else if( which == CONSTANTPOWER ) panPower( pan, left, right );
    else <<< "[pan]: ERROR specifying which pan type!", "" >>>;
}
```

Next, we write a function to control the playing of a single rain drop. We will control the `LPF` soon to make it seem as if the rain is falling in sheets – some sheets harder than others, and thus with more high frequency components! So we include controls for both the lower and higher cutoff values of the `LPF`. The function also uses a next function that the `Impulse` object has to “fire” the impulse.

```
// the drop
fun void oneDrop( RainDrop drop, float lowerFreq, float upperFreq, int panType )
```

```
{
    // randomize filter
    Math.random2f(lowerFreq,upperFreq) => drop.filter.freq;
    // randomize pan
    panning( panType, Math.random2f(0,1), drop.left, drop.right );
    // fire an impulse
    Math.random2f(.1,.8) => drop.rain.next;
}
```

The last thing to do now is actually create our sound ‘clip’, and write the process such that the rain will fall over time!

Now, we cannot simply have rain that follows at an entirely steady rate (as in, one every 50 ms) – rain is not rhythmic that way! To harness an algorithmic solution to this, we use the Poisson process to determine how much time to wait between rain drops. (More about this and algorithmic composition in Music 220B!)

```
// time until next event, given rate
// (this is based on the exponential distribution, which models
// time until next event in a poisson process - the events in
// this model occur independently, and have a rate of lambda)
fun float timeUntilNext( float lambda )
{ return -Math.log(1-Math.random2f(0,1)) / lambda; }


// define the "clip" as a function
fun void rainClip(dur myDur, int N)
{
      // rain drops, so we can rotate around and give each time to
      // have a tail (due to the filter) before being reused for the
      // rain drop
      RainDrop rain[N];

      // reverb
      JCRev rL => dac.left;
      JCRev rR => dac.right;
      // mix
      .2 => rL.mix => rR.mix;

      // connect
      for( int i; i < rain.size(); i++ )
      {
            rain[i].left => rL;
            rain[i].right => rR;
      }

      // counter
      int counter;

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

```
        myBeg + myDur => time myEnd;
        while (now < myEnd)
        {
              // LFO on upper freq
              800 + 2200*(1+Math.sin(now/second*.2))/2 => float upperFreq;
              // drop of rain
              oneDrop( rain[counter], 500, upperFreq, CONSTANTPOWER );
              // increment
              counter++;
              // modulo by rain array size
              rain.size() %=> counter;
              // wait (Poisson: from 220b)
              timeUntilNext(N)::second => now;
        }
        //extra time for reverb tails
        200::ms => now;
        <<<"\tclip end at",now/second,"seconds">>>;
}


// TIME 0, start the clip
spork ~rainClip(20::second, N); // launch clip in independent shred
20::second => now; // this master shred needs to remain alive while it's playing
me.yield(); // on this exact sample, yield master shred so sporked one can finish
first

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

Try changing the presets between linear and equal power panning. You should hear the sound in the center "pop" more in equal power panning…

Drive-By

In this example, we will pan the sound of a car passing in a flexible way, by writing a function where you can specify the start and end points of a sound file in stereo space – i.e. if you want it to pass all the way from left to right, right to left, from the midpoint to ¾ of the way to the right headphone, it will be simple to specify that. Note that, if we didn't want to use equal power panning, we could write a rather compact script using the ChucK object, `Pan2`, to pan our sound. However, we will take advantage of the `panning(), panLinear(), and panPower()` functions that we already wrote in the previous example for flexibility between different options. So, take those functions, as well as the global variables below and start a new sound file.

```
// set a global value
0 => int LINEAR;
1 => int EQUALPOWER;
```

The following function takes a `SndBuf`, left and right `UGen` reverbs (which are plugged into the `dac` outside of the function), a start and end location for the sound file (where 0 = left channel only, 1 = right channel only, .5 = middle, etc.), and an integer specifying which kind of panning you want to use. Note that if you set the start and end locations to the same value, the script will not pan the sound but keep it in one spot. Also note that this script 'moves' sound at a 'constant velocity' in time – both in speed and direction. If you want accelerating sound, or sound that changes direction, you'll need to extend this script!

```
// determines spatial position of sound file and plays it
fun void soundClip( SndBuf theSound, UGen left, UGen right,
                       float startLoc, float endLoc, int whichPan )
{
    // get length of sound file
    theSound.length() => dur len;
    // start sound from the beginning
    0 => theSound.pos;
    // the initial pan value (starting point)
    startLoc => float pan;

    // end time
    now + len => time endTime;
    // start time
    now => time startTime;
    // go
    while( now < endTime )
    {
        // pan it
        panning( whichPan, pan, left, right );
        // update where the sound should be
        startLoc + (endLoc-startLoc)*(now-startTime) / (endTime-startTime) => pan;
        //print
        <<< "pan:", pan >>>;
        // update rate
        1::ms => now;
    }
    // optional tail
    300::ms => now;
}
```

So, the above file can take any sound file, and for the duration of the sound file, move it in some specified direction across the stereo sound space. Let's test it out with a very directional sound – the sound of a car passing. Note that here, the sound file itself starts quiet, get's loud, and then gets soft again – this takes care of some of the dynamics we would have otherwise have had to handle, and lets us just focus on the location of the car. Below we just declare our `SndBuf car`, pass it through our reverbs, and spork the clip using equal power panning, and set the locations as `0` and `1`, meaning it will start in the left headphone and travel to the right.

```
// "car"
SndBuf car;
// read
me.dir() + "/car-mono.aiff" => car.read;
// left and right nodes for panning
car => Gain carL => NRev rL => dac.left;
car => Gain carR => NRev rR => dac.right;
// set reverb mix
.08 => rL.mix => rR.mix;

// call in the sound
spork ~ soundClip( car, carL, carR, 0, 1, CONSTANTPOWER );
car.length() => now;
me.yield();

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

Feel free to use/adopt this code for your own compositions. (Playing the car while it ‘drives’ through the rain is also quite satisfying!) Other good examples of musical panning of sound objects can be found in the examples of miniAudicle >> book >> digital-artists >> chapter 4. Specifically, Listing4.2.ck and Listing4.7.ck are compact examples of panning.

As in the previous labs, please submit brief sound files of your car passing as well as the rain falling. They should be titled LASTNAME_rain.wav and LASTNAME_car.wav.

For ease:

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

out.closeFile();
```