SuperCollider Tutorial

Chapter 5

By Celeste Hutchins

2005

www.celesteh.com

Gates and Envelopes

Nicole, the professional SuperCollider programmer, has been using her stock options to buy analog synthesizers. She's found that in the Analog world, there exists a very popular envelope called **ADSR**. That stands for **Attack, Decay Sustain Release**. When you hit a piano key, for example, or blow into a clarinet, the sound does not come up to full volume immediately. There is an **attack time**, which is the amount of time it takes for the volume to peak. Then, on maybe instruments, especially the piano, the sound **decay**s a bit from the initial attack. The volume then reaches the **sustain** volume. This is the meat of the note. (Or tofu of the note for vegetarians.) This is the volume that the note stays at as long as it is held. Then, the note ends, but the sound does not immediately die. As it is **release**d, the amplitude goes down to nothing, like the echo of a released note on the piano.

Obviously, real-life envelopes of things like clarinets are much more complex. However, in analog hardware, every envelope stage is another circuit. That is expensive, and her startup's stock price has to break $100 before she can afford an 8-stage envelope. SuperCollider is software and doesn't have the same restrictions. She can have as many Envelope stages as you want (and as her computer can handle), as you can see in the Env help file. There are two

costs of envelope stage: one is processor usage (which is miniscule), and the other is complexity. You, as the composer and the programmer, have to be able to manage the complexity of your sounds. Nicole has been working overtime and is stressed and has gotten to like ADSRs because they allow a lot of flexibility but are still not very complex. They are also traditional and have a nice retro sound that she digs.

One very interesting thing about ADSR envelopes is that they are not fixed duration. You do not have to know what the length of the note will be when you start it. You can play the note for as long as you like it and then end it when you are ready. You control this with a **gate**. These terms originate from analog synthesizers, so let's imagine Nicole's vintage synthesizer. She has a keyboard. They keyboard is actually a switch. When the key is depressed, the switch is closed and electricity is flowing. When they key is lifted, the switch opens and electricity is no longer flowing. The key is a gate. It can be open or closed. When she presses the key, the envelope starts, going through the ADS parts. When she takes her finger off the key, the envelope goes on the R part. An open gate has a value of 0. A closed gate has a value of 1.

Nicole has written an ADSR into a SynthDef:

```
(

    SynthDef.new("example 4", {arg out = 0, freq, amp = 0.2, gate = 1,
                                a, d, s, r;

        var env, sin;

        env = EnvGen.kr(Env.adsr(a, d, s, r, amp), gate,
                        doneAction: 2);
        sin = SinOsc.ar(freq, mul:env);
```

```
        Out.ar(out, sin);
    }).send(s);

)

a = Synth.new("example 4", [\freq, 440, \a, 0.2, \d, 0.1, \s, 0.9,
                            \r, 0.5, \gate, 1]);
```

When she runs this, the note starts to play. When she is ready for the note to end, she sends a message to the synth to set the gate to 0. She uses the set message.

```
a.set(\gate, 0);
```

The Synth plays until she sends a message to it saying to set the gate to zero. She can send messages to any parameters using set. Try running this, pausing between each command:

```
a = Synth.new("example 4", [\freq, 440, \a, 0.2, \d, 0.1, \s, 0.9,
                            \r, 0.5, \gate, 1]);

a.set(\freq, 330);

a.set(\gate, 0);
```

Remember that the attack, sustain and decay all take place while the gate is open, or 1. The release happens after the gate is off, or zero. The envelope generator uses the gate. When the gate opens, the envelope generator begins to play through the adsr. When it gets to the sustain portion, it stays there until the gate closes and it plays the release portion of the adsr. When that is finished, it looks at the doneAction. Since the doneAction is 2, it removes the

synth. We don't have to remove the synth, though. We could keep it around and keep passing new values to it. Try to run the next example. Remember to pause between set messages.

```
(

     SynthDef.new("example 5", {arg out = 0, freq, amp = 0.2,
                                gate = 1, a, d, s, r;

          var env, sin;

          env = EnvGen.kr(Env.adsr(a, d, s, r, amp), gate,
               doneAction: 0);
          sin = SinOsc.ar(freq, mul:env);
          Out.ar(out, sin);
     }).send(s);

)

a = Synth.new("example 5", [\freq, 440, \a, 0.2, \d, 0.1, \s, 0.9,
                            \r, 0.5, \gate, 1]);

a.set(\gate, 0);

a.set(\freq, 330, \a, 0.2, \d, 0.1, \s, 0.9, \r, 0.5, \gate, 1);

a.set(\gate, 0);
```

See that when you look at the server window, it says "Synths: 1. That synth is still there, waiting for us to set it again. Run it again, but this time, change the pitch in the middle of a note.

```
a.set(\freq, 330, \a, 0.2, \d, 0.1, \s, 0.9, \r, 0.5, \gate, 1);

a.set(\freq, 587);

a.set(\gate, 0);
```

When you want this synth to go away, remember that you can deallocate it by pressing apple-. Remember also that a doneAction of 0 is the default.

Nicole's boss hears her code and likes it. The contract she signed says that all of her SuperCollider code belongs to the company, so he tells her to keep working on this, but to smooth the pitch change. She incorporates a ugen called **Slew**:

```
(

    SynthDef.new("example 6", {arg out = 0, freq = 440, amp = 0.2,
                                gate = 1, a, d, s, r;

        var env, sin, slew;

        env = EnvGen.kr(Env.adsr(a, d, s, r, amp), gate);
        slew = Slew.kr(freq, 4000, 4000);
        sin = SinOsc.ar(slew, mul:env);
        Out.ar(out, sin);
    }).send(s);

)

a = Synth.new("example 6", [\freq, 440, \a, 0.2, \d, 0.1, \s, 0.9, \r,
0.5, \gate, 1]);
```

```
a.set(\freq, 587);

a.set(\freq, 330);

a.set(\gate, 0);
```

Slew is an Ugen. The .kr means that it runs at the Control Rate. Slew does not produce audio. It produces what, on Nicole's synthesizers, would be called a "control voltage." That is a number that slowly changes (much below the audio rate) to modify another Ugen.

Slew, takes a few arguments, the first is the thing to be slewed, the next two are the upslope and the downslope, and then it takes mul and add.
We can slew any of the arguments to a SynthDef.

You can see from Slew that Ugens can modify each other. You can pass a control rate ugen or an audio rate ugen into the input of and audio ugen and use it with any argument such as the freq, the phase, the mul, the add, whatever. When you are experimenting with this, make sure **not** to use the add argument of the oscillator that is the last one in a chain, because that would add something called **dc bias**. This can hurt your speakers. DC Bias means that your waveform is not centered on zero; it is offset to center around some other number. This means that the magnets in your speakers have to pull towards that number more and it stresses your speakers. Aside from that, try changing ugens. Send oscillators as arguments to oscillators. We'll talk more about the names for the different ways to do that in the future.

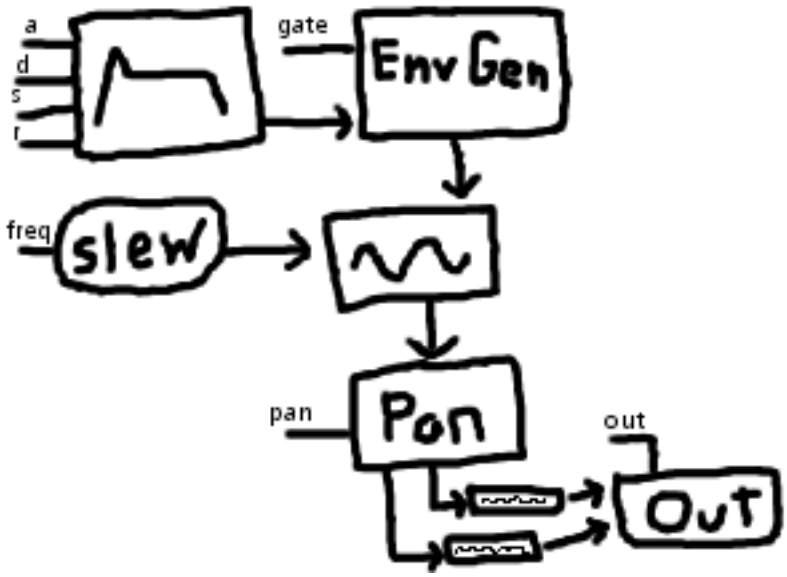
Pan

Nicole's boss is pleased, but he wants the sound to be stereo. Much like a pan knob on a mixing board, the ugen **Pan2** can move a signal between two channels. So she adds pan:

```
(

    SynthDef.new("example 7", {arg out = 0, freq = 440, amp = 0.2,
                                gate = 1, a, d, s, r, pan = 0;

        var env, sin, slew, panner;

        env = EnvGen.kr(Env.adsr(a, d, s, r, amp), gate);
        slew = Slew.kr(freq, 4000, 4000);
        sin = SinOsc.ar(slew, mul:env);
        panner = Pan2.ar(sin, pan);
        Out.ar(out, panner);
    }).send(s);

)
```

Pan2.ar is an audio rate constructor. Pan is audio rate because it outputs an audio signal. The first input is the signal to be panned. The second input is the position to pan it to. -1 is hard left, 0 is middle, and 1 is hard right. A number in between those numbers pans it between those positions. Pan works by creating an array of ugens. The first element of the array is the left channel. The second element is the right channel.

How many ugens are in that SynthDef? Nicole sketches it in a notebook:



Outputs of ugen operations are ugens. You can see that there are 8 UGens in our SynthDef. Six of them were created with constructors and Pan created two.

GUI

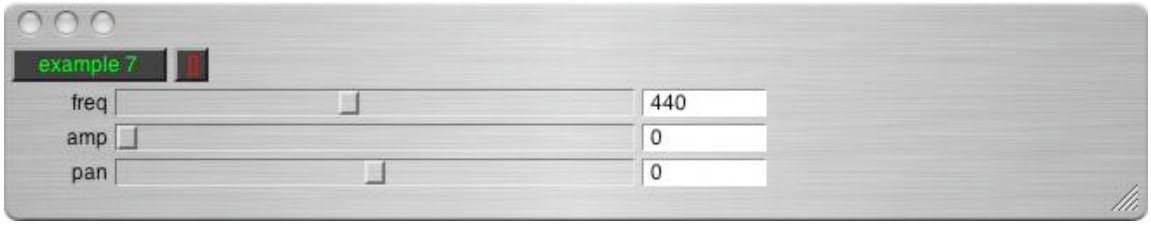
Nicole is using her sketch to keep track of what's what. However, all these set messages aren't very user-friendly. She wants a **GUI** to make her Synths easier to use. A GUI is a graphical user interface. She looks up a class called **Display**. The helpfile tells her that, "Display is a ControlEnvironment that provides a GUI."

After reading the helpfile, she writes a simple Display:

```
(
    Display.make({arg thisDisplay, freq, amp, pan;

        freq.spec_(\freq);
        amp.spec_(\amp);
        pan.spec_(\pan);
```

```
        thisDisplay.synthDef_({arg freq, amp, pan;

            var sin, panner;

            sin = SinOsc.ar(freq, mul: amp);
            panner = Pan2.ar(sin, pan);
            Out.ar(0, panner);
        }, [\freq, freq, \amp, amp, \pan, pan]);

        thisDisplay.name_("example 7");
    }).show;
)
```

A new window opens:



Press the green “example 7” button to start the display running. The button should turn yellow when it runs. It’s not making any noise because the amp is turned down. Use your mouse to scoot along the fader to turn up the amplitude.

Display.make is a constructor. It takes a function as it's one argument. The function takes an arbitrary number of arguments. The first one is the Display object returned by the constructor. Nicole called it "thisDisplay" because it literally refers to this Display. (`this` is a reserved word in SuperCollider.) The additional arguments are **ControlValues**. She called her ControlValues "freq", "amp" and "pan".

ControlValues have a property called spec. Messages that end with an underscore are called **setter** messages. Highlight the word ControlValue and then hit apple-j to see the source code for the class. SuperCollider classes are written in SuperCollider. Look at the top of the file, below the comments. "`var <spec;`" means that ControlValue has an internal variable called spec and that users can change the value of that variable with "`spec_`". We'll come back more to how classes work later. In this case, Nicole is using some predefined values, so she passes some literals (\freq, for example) as arguments, telling it which predefined value to use.

Then she set the SynthDef associated with thisDisplay. That setter messages takes two arguments. One is a function that you would pass to a normal SynthDef. The second argument is the Array that you would use creating a new instance of Synth. \freq matches the SynthDef argument freq. \amp matches the SynthDef argument amp. freq (without the slash) matches the ControlValue freq. amp matches the ControlValue amp. What's going on there is that she is telling the preset to make a SynthDef and when it creates a Synth, to use her ControlValue variables to control the Synth arguments.

Then she sets a name for the Display, calling it "example 7".

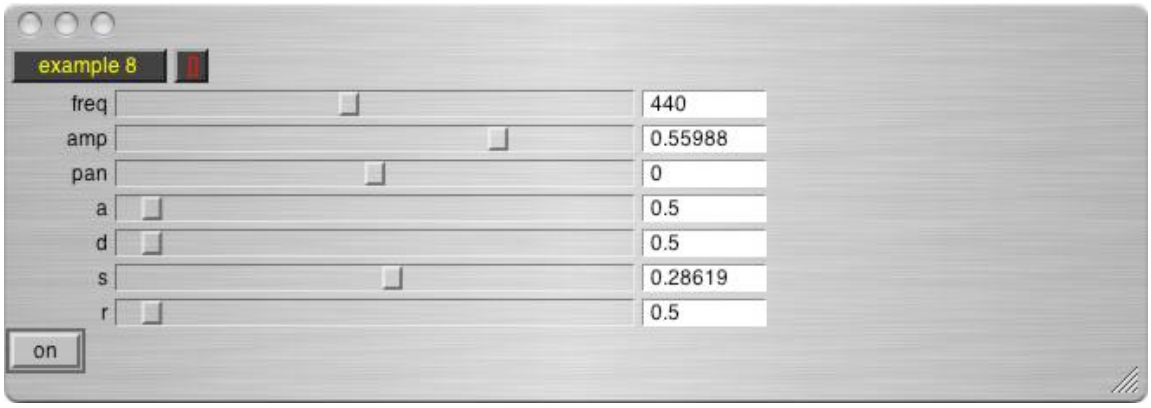
Then she takes the Display created by the constructor and passes a message to it telling it to show itself.

Nicole realizes that this is a great way to test out what different values for Synths sound like. She comes up with a way to test out her ADSR envelopes:

```
(
    Display.make({arg thisDisplay, freq, amp, pan, a, d, s, r, gate;
```

```
var button;

freq.spec_(\freq);
amp.spec_(\amp);
pan.spec_(\pan);

a.sp(0.5, 0, 10, 0);
d.sp(0.5, 0, 10, 0);
s.spec_(\amp);
r.sp(0.5, 0, 10, 0);

button = ControlButton(gate);
button.states_([["off"], ["on"]]);

thisDisplay.guiItems_([
     thisDisplay.clientgroup,
     freq, amp, pan, a, d, s, r,
     button
]);

thisDisplay.synthDef_({arg freq, amp, pan, a, d, s, r, gate;

     var sin, panner, env;

     env = EnvGen.kr(Env.adsr(a, d, s, r, amp), gate);
     sin = SinOsc.ar(freq, mul: env);
     panner = Pan2.ar(sin, pan);
     Out.ar(0, panner);
}, [\freq, freq, \amp, amp, \pan, pan, \a, a, \d, d,
          \s, s, \r, r, \gate, gate]);

thisDisplay.name_("example 8");
```

```
    }).show;
)
```



Turn up the amp and the s. Click the "example 8" button to start it running. Then click the on/off button a few times to get the gate going. How did Nicole know how to write this?

She looked at the helpfile for ControlValue and discovered all of the different default ControlSpecs she could use. There weren't any that were appropriate for the timings of the ADSR, so she used the sp method to create some with an initial value of 0.5, a minimum value of 0 and a maximum value of 10. She remembered that s is the sustain amplitude, so she used the default of /amp.

The Display helpfile mentioned ControlButton and gave a piece of example code. She copied the code, creating a button and giving it two states. She set the first state to off, because it is at the array position of 0 and an off gate is 0.

The Display helpfile explains that guiItems "is an array of objects that respond to the method **draw(window)**." Copying the example code, she included thisDisplay.clientgroup in the guiItems array, and then she listed all of the other ControlValues and her button.

She put the adsr envelope into the SynthDef and then added adsr and gate to the array on synth arguments and control values.

Problems

1. Write a SynthDef that slews the amplitude. Use set to send it some amplitude changes. Do the same for the pan.

2. Sketch out your SynthDef. Do the number of UGens displayed in the Server window match the number you expected from your sketch?

3. Write a Routine that creates multiple Synths and controls them with set messages. The synths can be instances of the same SynthDef.

4. If you have a four channel set up, experiment with Pan4.ar. How does `Pan4.ar(sin, x, y)` differ from `Pan2.ar(Pan2.ar(sin, x), y)` ?

5. Create a Display using different ugens, including Saw and Pulse. Use a ControlSpec to control the width of the pulse.

Project

You can mix multiple signals by adding them all together and dividing by the number of signals. For example:

```
result = (sine1 + sine2 + sine3 + sine4 + sine5) / 5;
```

Create a Display with multiple Oscillators, each frequency and amplitude controlled separately. Write a one or two minute piece that you play by moving the sliders.