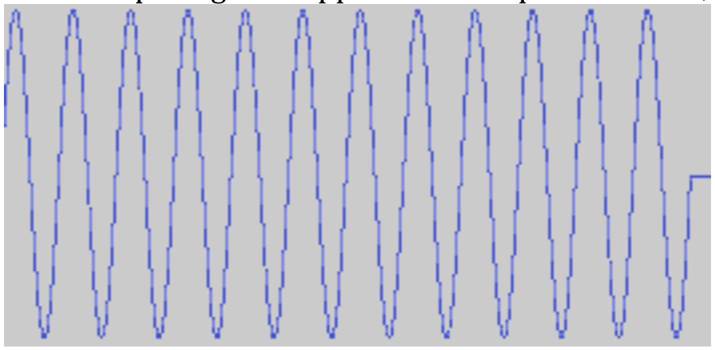
COS/MUS 314

Envelopes, pull model, and blackholes in ChucK

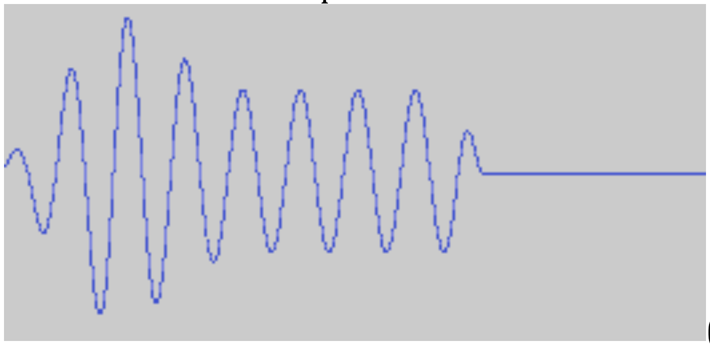
Envelopes

An envelope is a function that is multiplied with a waveform to impart a large-scale shape to a sound. An envelope is most often used to play a single note, for example to mimic the exponentially decaying amplitude shape of a struck object, or to mimic the gradual note onset and release of a bowed string.

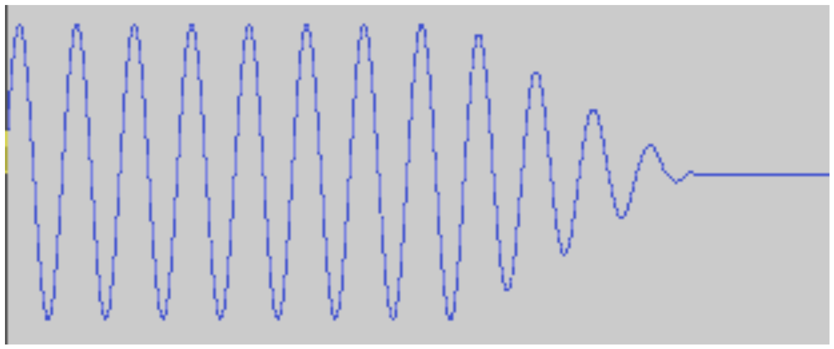
An envelope might be applied to a simple sine wave, for example this:



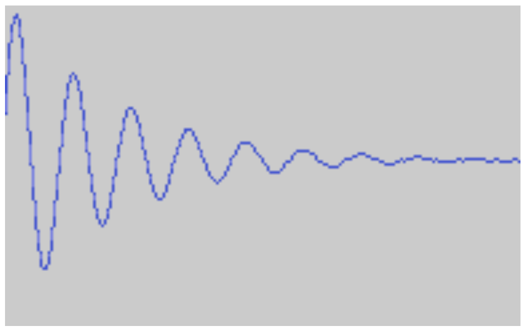
to turn it into an enveloped sine that looks like one of these:



(ADSR with noticeable A, D)

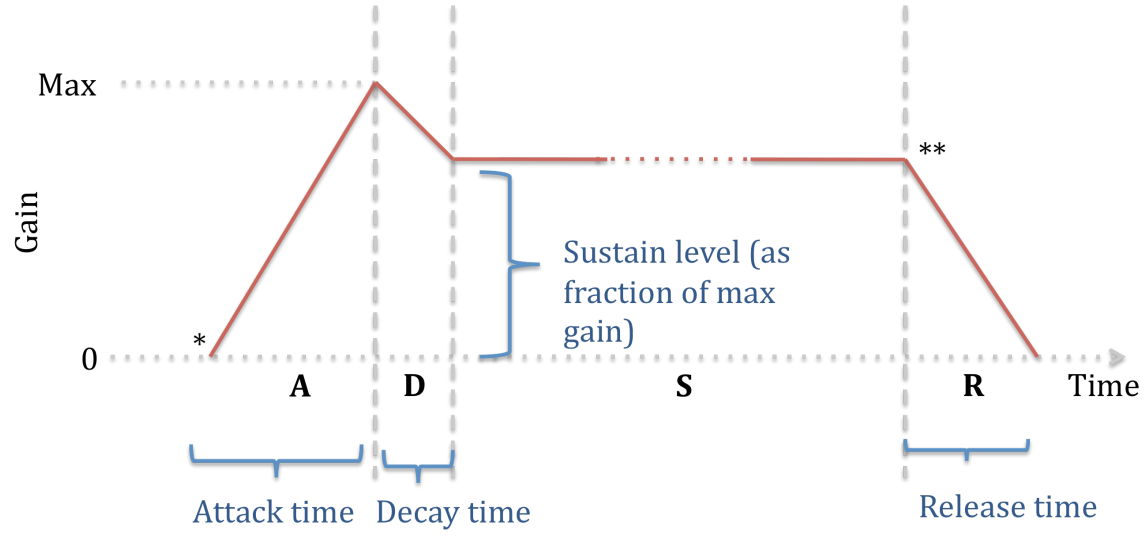


(ADSR with long decay)



(exponential decay)

ADSR Envelopes
An ADSR envelope imparts a simple shape to a sound. It has 4 sections: the attack, decay, sustain, and release. The shape of the envelope appears below in red:



In ChucK, the ADSR UGen object has a .set method that allows you to set the attack time (in seconds), decay time (in seconds), sustain level (as a fraction of the max gain), and release time (in seconds) (parameters shown in blue above).

To use an ADSR object called "a" in ChucK, you'll 1) call
a.keyOn();
to initiate the attack phase of the envelope (shown with * in the diagram);
2) advance time to play the note (the A and D sections will be applied according to their specified durations, and then the S part of the envelope will sustain indefinitely), and
3) then call
a.keyOff();
to initiate the release section (shown with ** in the diagram).

Note that you **must still advance time** for the duration of the ADSR release section, at minimum, in order for the release to complete!

For example, the following code plays two enveloped quarter notes:

```
SinOsc s => ADSR a => dac;
(.01, .01, .99, .12) => a.set;

.5::second => dur quarter;

a.keyOn();
(quarter - a.releaseTime()) => now;
a.keyOff();
a.releaseTime() => now;

a.keyOn();
(quarter - a.releaseTime()) => now;
a.keyOff();
a.releaseTime() => now;
```

The Envelope UGen

The Envelope UGen is more general than an ADSR. Setting a new **target** value causes it to ramp from its current value over the number of seconds set by its **duration** method.

For example, to play a note whose gain goes from 0 to .7 over .5 seconds, sustains for 1.5 seconds, then goes back to 0 over 2 seconds:

```
SinOsc s => Envelope e => dac;
.5::second => e.duration;
.7 => e.target;
2::second => now;
2::second => e.duration;
0 => e.target;
2::second => now;
```

It is also possible to use Envelope to *smooth* other parameters, such as the frequency of an oscillator, as discussed below.=

The ChucK pull model

So far, we've always attached unit generators to the **dac** (digital analog converter) because we want our patches to make sound. When we add a shred to the VM that has a patch like the one below, this has a very specific implication for how the shred executes.

```
SinOsc s2 => ADSR a => dac;
```

The ChucK internal sample clock requires the dac to output one sound sample every 1/44100 th of a second. Every time the dac is tapped for this sample, it looks "behind" it, or "upstream" in the UGen patch, and sees the ADSR object a. It asks a for its next value. In turn, a looks upstream from it and sees the objects on which it depends for *its* next value, namely the sine oscillator called s. a asks s for its value (i.e., the next value in the sine wave s is generating). A does its computation on s (i.e., multiplies s's value with the next value of a's envelope) and passes the result to the dac. The dac pushes this sample out to be played.

This is a *pull model* for computation, in that the dac *pulls* samples from a, causing a to pull samples from s, and so on. If we have a lonely UGen that's not being pulled by anything, for example

```
SinOsc lonely;
```

this oscillator called lonely will never have samples pulled from it, and it will never compute.

This is often a good thing—we don't want it to compute if we're not using it at all. But what if we want to do something else with its results instead of make sound?

The blackhole object

The **blackhole** object is used to pull samples from a patch, triggering computation of any upstream UGens, without making sound. For example, the following patch causes lonelynomore to compute a sine wave that gets printed out instead of played:

```
SinOsc lonelynomore => blackhole;
```

```
now + .01::second => time later;
while (now < later) {
   <<< lonelynomore.last() >>>; //gets last
sample of lonelynomore
   1::samp => now;
}
```

Using envelopes for smoothing anything at all
We can use a blackhole to pull samples from an Envelope object, thus triggering computation of the envelope itself, then apply the envelope to something other than gain. For example, in the loop below we manually set the frequency of an oscillator based on the current value of an envelope. We can advance this loop every samp if we want, but here every .01 seconds sounds fine (and incurs many fewer computations per second).

```
SinOsc s => dac;
Envelope e => blackhole;
100 => e.value; //set without sliding
1000 => e.target;
2::second => e.duration;
e.value() => s.freq;

e.duration() + now => time later;
while (now < later) {
   e.value() => s.freq;
   .01::second => now;
}
```