Phex

A concise live coding environment for SuperCollider

Abstract

This document describes the design and some of the reasoning behind Phex, a live coding environment built on top of SuperCollider whilst describing how to use it. It assumes some knowledge of both the pattern system, control specifications and the construction of synthesis graphs within SuperCollider. It does not cover all of Phex's features but should allow a user to get started working with the system.

Introduction

The practice of live coding is becoming increasingly prevalent[1] as a means of expression in artistic fields. Phex has been developed to allow me to explore it as a tool for the performance of music and to look into how instruments and musical patterns can be designed with the minimal amount of SuperCollider code. It aims to bridge the divide that still exists between more minimalist performance environments and SuperCollider to some degree.

The Phex environment is inspired by languages such as Ixi Lang[2] and Tidal[3] where melodic and rhythmic patterns can be defined quickly using strings of characters but also includes facilities for the quick mapping of parameters of these patterns to external hardware controllers, a simple means for applying effects to these patterns, and provides a familiar mixing desk style user interface for monitoring and mixing of audio signals all the while providing tight integration with the typical SuperCollider workflow and the ability to use more standard object oriented or functional programming techniques.

It also allows SynthDefs of various types to be programmed by its users more quickly than other facilities provided by the SuperCollider system.

Functionality

Phex's and the Phex language

The primary component of the system is the Phex class from which the system takes its name. This can be utilised within the Phex environment or in any of the situations ListPatterns such as Pseq and Pshuf can be used. They provide a very compact notation for performing the same tasks as the aforementioned ListPatterns. Although not usually instantiated as described in this section of the document I've opted to create instances of them directly in the code examples below in the hope that this will help describe how they function.

They utilize a language consisting of hexadecimal digits, spaces, question marks, asterisks and full stops.

A simple Phex

A simple Phex that provides a stream of values from 0 to 15 can be defined as follows:

```
Phex(“0123456789abcdef”)
```

This is equivalent to the following Pseq in SuperCollider:

```
Pseq([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], inf)
```

Rests in Phexs - Spaces

Rests can be embedded in Phex sequences as spaces, for example:

```
Phex(“a b c “)
```

which is equivalent to:

```
Pseq([10, \rest, 11, \rest, 12, \rest], inf).
```

Repetition in Phexs - the * symbol

The asterisk symbol allows Phex’s users to indicate that sections of a pattern should be repeated a certain number of times. The digit following an asterisk indicates the number of times that a section of a pattern should be repeated. The section that’s repeated starts immediately after this and is closed with a full stop. A simple expression that utilizes this operator is:

```
Phex(“*f12 .*ffe”)
```

which is equivalent to:

```
Pseq([Pseq([1, 2], 15), Pseq([15, 14], 15)], inf)
```

These expressions can be nested. An example of this might be:

```
Phex(“*30  .*2fe*4da.bc.”)
```

which is equivalent to:

```
Pseq([Pseq([0, \rest, \rest], 3), Pseq([15, 14, Pseq([13, 10], 4), 11, 12],
2), inf)
```

Shuffling sections of Phexs - the ? symbol

The question mark symbol causes a section of a Phex to be shuffled. The end of a section being shuffled is once again the full stop operator.

The following Phex:

```
Phex(“12?123.”)
```

is equivalent to:

```
Pseq([1, 2, Pshuf([1, 2, 3])], inf)
```

Shuffled sections of Phexs can be embedded within other repeated or shuffled sections of Phexs allowing extremely complex patterns to be defined with relatively little typing.

Mapping Phex’s to different ranges

By default the hexadecimal digits present in a Phex represent numbers from 0 to 15. A spec can be provided to a Phex that defines a mapping for these values. These specs can be specified as symbols, arrays, ControlSpecs or any other object in SuperCollider that responds to the .asSpec message. Here a Phex’s values are mapped between 1 - 1000 exponentially.

```
Phex(“08f”, [1, 1000, \exp])
```

This Pseq is its equivalent:

```
Pseq([ 1, 39.81071705535, 1000 ], inf)
```

Phatterns - combining Phexs and Patterns and playing them back

The Phattern class is responsible for a number of different tasks in the Phex environment including managing FX,

Creating a Phattern

A Phattern can be created as follows, the user should be careful to keep a reference to it somewhere such that it can be manipulated later as shown here:

```
~myPhattern = Phattern(\default)
```

The above Phattern is mapped to the default SynthDef provided by SuperCollider.

Playing back a Phattern

To start the Phattern we’ve just created we can send it a .play message.

```
~myPhattern.play
```

This starts a single note repeatedly playing with the default settings specified by the `\default` SynthDef.

Making a Phattern playback some Phexs and patterns

Phatterns combine references to value streams that can be replaced whilst playing. Phexs and other patterns generate these streams.

Here we use a Phex to define the degrees of a scale that its going to play back. When a Phatterns parameter is set to a string it converts that string to Phex. Here we set our Phattern to play back a sequence of notes:

```
~myPhattern.degree = “0 2*25.”
```

Above we see that the Phattern supports the degree key in precisely the same as a Pbind or Pmono class, it’s just accessed in a different way. The same goes for all of the keys defined in the default event type in SuperCollider. Just like Pbind keys can be used to modulate any parameter of a synth.

When values are provided to a Phattern it utilizes them in the same was a Pbind would. Here we set each step of the pattern to a quarter beat:

```
~myPhattern.dur = 1 / 4
```

It could have just as easily been set to a ListPattern of any type (including Phexs) or even a reference to another key via Pkey, here we set it to use a Prand that makes each note last 1 or 3 quarters of a beat:

```
~myPhattern.dur = Prand([1, 3], inf) / 4
```

Now for fun we’ll set the octave of the Phattern that’s playing back to a shuffled Phex:

```
~myPhattern.octave = "?567 ."
```

Note that in the same way as a Pbind etc. a rest in any of it’s frequency related parameters results in a note not being sounded.

Here we’ll set the Phattern to play back it’s pattern in the Dorian mode:

```
~myPhattern.scale = Scale.dorian
```

Recalling Phattern’s patterns for manipulation

A Phatterns patterns/parameters can be recalled:

```
~myPhattern.degree
```

This returns the Pseq created by our Phex earlier. This can be manipulated like any Pseq. Let's transpose our "melody" up a couple of degrees a few times. Run the following line several times:

```
~myPhattern.degree = ~myPhattern.degree + 2
```

Phexs can be combined with other list and filter patterns to create even more complex patterns.

Stopping a Phattern

A Phattern can be stopped by sending it a .stop message: `~myPhattern.stop`

The PhexEnv environment

The PhexEnv class provides an environment well suited to working with groups of Phatterns and can be used to set parameters/keys for all existing and future Phatterns at once. It also allows for the quick creation of SynthDefs for instruments and effects, displays a mixer which can be used to set the volume for a Phatterns and provides facilities for quickly mapping Phattern keys to MIDI controllers.

Creating and entering a PhexEnv environment

The simplest means of starting up a PhexEnv is as follows, as with all SuperCollider environments in order to make it the current one it needs to be pushed:

```
e = PhexEnv().push
```

Upon doing so an empty window opens up. This'll be used to display mixer channels that can be used to set the volume and panning of Phatterns.

Creating instruments for use with Phatterns

The PhexEnv provides to facilities for the quick creation of instruments. These are exposed via two functions, one that creates SynthDefs with percussive envelopes mapped to their amplitude and have an initialisation rate frequency and amp arguments: *~addPercDef* and another that that creates SynthDefs with ADSR envelopes mapped to their amplitude with a control rate frequency and amp argument *~addAdsrDef*.

Both have the same parameters and provide access to the envelopes they define as the first parameter to UGen function provided by its user. They also both have their amplitudes reduced as frequency is increased such that high notes sound at what's perceived to be the same volume as low notes. The percussive SynthDef's length are controlled by a sustain parameter. An ADSR synths sustain is controlled by a gate parameter and it's attack, decay, sustain and release parameters can be accessed via *\att, \dec, \sus,* and *\rel* arguments.

Here we'll define an instrument that uses PM modulation to generate it's sound, it's predefined envelope is passed in as its first parameter and modulates the index of the synth:

```
~addPercDef.(\pmPing, { |env|
      PMOsc.ar(\freq.ir(440), \mfreq.ir(330), \index.ir(10) * env)
})
```

Just like with the majority of SynthDef's designed to work with patterns freq is used to set the synths playback frequency and amp is used to set its amplitude. It can be played like any normal Synth: `Synth(\pmPing)`

~addPercDef and ~addAdsrDef can take the following parameters alongside the name and Ugen function demonstrated above:

- *specs:* A dictionary of control specs. This can provide a list of default specs for each of the synths parameters. These are automatically utilised by Phatterns when specified. Populating this is the same as populating a SynthDefs metadata dictionary with a dictionary of specs.
- *lags:* Lags define how a sounds parameters are smoothed upon changing.

Using our new percussive SynthDef with a Phattern

The synth we've just created can be used expressively when mapped to a Phattern as could an ADSR based definition:

```
~ting = Phattern(\pmPing)
```

If you check the mixer window that was displayed on screen by PhexEnv a labelled volume, panning and meter that shows peak volumes for the ~ting phattern should have appeared. These can be manipulated like any standard mixer channel UI in order to balance sounds quickly and track down and avoid clipping etc.

Start the Phattern playing and you'll see the meter move.

```
~ting.play
```

You'll note that any of the synths parameters can be modulated with Phattern keys:

```
~ting.mfreq = "456789abcdef"
```

Setting a Phattern keys default spec for use with Phexs

You're likely to find the range a Phex modulates over to not be to your tastes initially, you could specify the spec for it to operate over like so:

```
~ting.mfreq = Phex("456789abcdef", [10000, 1000, \exp])
```

However this would mean that every time you updated that parameter you'd have to respecify the spec. Instead you can do:

```
~ting.specs[\mfreq] = [10000, 1000, \exp]
```

And the above spec will always be used when you set the mfreq key with a Phex string, the same line you entered above will now sound the same as when you explicitly provided a Phex for that Pattern key a moment a go[1]:

```
~ting.mfreq = "456789abcdef"
```

Other options involve setting up a default Spec with the same name as this paramater or providing specs along with the synth definitions metadata.

Creating effects for use with Phatterns

PhexEnv exposes the ~addFxDef method for creating effects that are compatible with Phatterns.

A simple reverb effect can be created as follows:

```
~addFxDef.(\verb, { |sig| GVerb.ar(sig, drylevel: 0) });
```

sig is the incoming signal from the channel and is always provided as the first argument to the UGen function.

Like the others these effect definitions can be adorned with details of specs and lag times for its parameters upon creation by tacking these details on after the Ugen function.

We can add an instance of it called verby to our ~tinks pattern by executing:

```
~ting.setFx(\verby, \verb)
```

The reverb trails will be horribly loud. You can reduce their level by using the dryWet parameter provided to all effect definitions defined with ~addFxDef:

```
~ting.fx[\verby].set(\dryWet, 0.1)
```

The effect can be removed by setting \verby to *nil*.

```
~ting.setFx(\verby, nil)
```

Setting keys of all Phatterns within a PhexEnv at once

Setting the keys of all of the Phatterns in an environment at once is quite simple.

Executing:

[1] Due to flaws with my design specs of Phex patterns are not automatically updated when a Phattern parameters specs are changed. This can be worked around by executing both the update of the spec and executing the the assignment of a Phex to a parameter at the same time as updating the spec..

```
~scale = Scale.lydian
```

will set the scale key of all Phatterns currently active and created later to Lydian.

Similarly `~transpose = "123 456"` will set all Phatterns transpose keys to a sequence that counts from 1 to 6 with a rest in the middle.

This works for any key utilised by any Phattern so long as Phattern does not implement a method with that name and the value assigned to it isn't a buffer, function or a collection other than a string since those types of objects are used no differently to normal in PhexEnv.

Sequencing buffers with Phatterns

Phex includes a different method for describing patterns involving many different samples that are tied to a *buffer* key exposed by the Phattern and a special *buffers* dictionary.

A user can assign buffers to characters and then produce sequences of characters that indicate in what order buffers will occur.

A dictionary containing buffers should be created, for example:

```
~drumSounds = (
      k: "kick.wav", h: "hh.wav", o: "oh.wav"
).collect { |p| Buffer.readChannel(s, "/path/to/me/samples" +/+ p) }
```

And then assigned to a Phattern. The PhexEnv comes with a SynthDef designed for playing single channel buffers called *\phampler* which it makes sense to map to in this instance.

```
~drums = Phattern(\phampler)
~drums.dur = 1 / 2
~drums.buffer = "khkhkhko"
```

Unfortunately this does not support repetition or shuffling in the same manner as Phexs but SuperCollider's built in patterns can be used to achieve this. [2]

MIDI mapping Phattern keys

PhexEnv exposes the function *~midiMap* which can be used to quickly map incoming MIDI values from a particular device on a particular CC to one or more Phattern parameters. The *\phampler* SynthDef described earlier has a cutoff parameter that could be mapped. In order to map this you could enter:

[2] Given more time I'd develop some form of spec that mapped from a value between 0 and 1 to a dictionary of buffers in some way to allow samples to be sequenced in a manner in line with the rest of the application.

```
~drums.cutoff = ~midiMap.([100, 16000, \exp])
```

And then manipulate a MIDI controller. The first CC received at this point will be used to assign values between 100 and 16000 to cutoff exponentially. This process allows users more immediate of certain parameters within a system. To unmap this controller again all that needs to occur is the assignment of a new value or sequence the cutoff key.

Conclusion

I feel I have succeeded in developing a system that describes complex patterns using strings that integrates better with the more typical SuperCollider workflow than pre-existing systems I've used such as Ixi Lang.

Developing a system for Live Coding is an interesting task involving many trade offs. The first and most apparent is the need for brevity competing with that of readability. All though pleased with the Phex language I created I certainly enjoy manipulating and generating Phexs more than I do reading them.

The second was that of convenience versus versatility it seems standards need to be adhered to in order to make a system quick to interact with and reduce cognitive load on its users but in doing so in this instance I've lost some of the flexibility offered for by routing busses freely etc.

As of yet I'm not fully convinced as to whether or not Live coding is my preferred means of performance and I certainly produce more professional sounding results using tools such Ableton Live or a Kaoss Pad and an iPad, however I look forward to engaging, practicing with and improving this system in the future.

References

[1] Toplap, Toplap.org, 2014
[2] Magnusson, THE IXI LANG: A SUPERCOLLIDER PARASITE FOR LIVE CODING, https://www.academia.edu/1979535/ixi_lang_a_SuperCollider_parasite_for_live_coding, 2011
[3] Alex McLean, Demonstrating Tidal, http://yaxu.org/demonstrating-tidal/, 2014