Tidal – Domain specific language for live coding of pattern

Homepage and mailing list: http://yaxu.org/tidal/

Tidal is a language for live coding pattern, embedded in the Haskell language. You don't really have to learn Haskell to use Tidal, but it might help to pick up an introduction. You could try Graham Hutton's "Programming in Haskell" or Miran Lipovača's "Learn you a Haskell for Great Good" (which has a free online version). Or, you could just try learning enough by playing around with Tidal.

Installation

Tidal is developed under Linux, and although some have got it to work under Macs, the process hasn't been fully documented, and Dirt synthesiser has not yet been ported to Windows. Feel free to ask questions and share problems and success stories on the mailing list.

Installing Dirt

Tidal does not include a synthesiser, but instead communicates with an external synthesiser using the Open Sound Control protocol. It has been developed for use with a particular software sampler called "dirt". You'll need to run it with "jack audio". Here's an example of the commands needed to compile it under a debian-derived linux distribution (including ubuntu and mint):

```
sudo apt-get install build-essential libsndfile1-dev libsamplerate0-dev λ
                     liblo-dev libjack-jackd2-dev qjackctl jackd git
git clone https://github.com/yaxu/Dirt.git
cd Dirt
make clean; make
```

Then you'll have to start jack, using the 'qjackctl' app under Linux, or otherwise from the commandline:

```
jackd -d alsa &
```

(On MacOS X, you would do this instead: jackd -d coreaudio &)

If that doesn't work, you might well have something called "pulseaudio" in control of your sound. In that case, this should work:

```
/usr/bin/pasuspender -- jackd -d alsa &
```

And finally you should be able to start dirt with this:

```
./dirt &
```

If you have problems with jack, try enabling realtime audio, and adjusting the settings by installing and using the "qjackctl" software. Some more info is here: https://help.ubuntu.com/community/HowToJACKConfiguration

Tidal

Tidal is embedded in the Haskell language, so you'll have to install the haskell interpreter and some libraries, including tidal itself. Under debian, you'd install haskell like this:

```
sudo apt-get install ghc6 cabal-install
```

Or otherwise you could grab it from http://www.haskell.org/platform/

Once Haskell is installed, you can install tidal like this:

```
cabal update
cabal install tidal
```

Emacs

Currently about the only interface to Tidal is the emacs editor. Debian users can install emacs, along with its haskell front-end, this way:

```
sudo apt-get install emacs24 haskell-mode
```

To install the emacs interface to tidal, you'll need to edit a configuration file in your home folder called `.emacs`. If it doesn't exist, create it. Then, add the following, replacing `~/projects/tidal` with the location of the `tidal.el` file.

```
(add-to-list 'load-path "~/projects/tidal")
(require 'tidal)
```

If tidal.el did not come with this document, you can grab it here: https://raw.github.com/yaxu/Tidal/master/tidal.el

Testing, testing...

Now start emacs, and open a new file called something like "helloworld.tidal". Once the file is opened, you still have to start tidal, you do that by typing `Ctrl-C` then `Ctrl-S`.

All being well you should now be able to start making some sounds, lets start with some simple sequences.

Sequences

Tidal starts with nine connections to the dirt synthesiser, named from `d1` to `d9`. Here's a minimal example, that plays a bass drum every loop:

```
d1 $ sound "bd"
```

In the above, `sound` tells us we're making a pattern of sounds, and `"bd"` is a pattern that contains a single sound. `bd` is a sample of a bass drum. To run the code, use `Ctrl-C` then `Ctrl-C`.

We can pick variations of a sound by adding a slash then a number, for example this picks the fourth bass drum (it starts with 0):

```
d1 $ sound "bd/3"
```

Putting things in quotes actually defines a sequence. For example, the following gives you a pattern of bass drum then snare:

```
d1 $ sound "bd sn"
```

When you do `Ctrl-C Ctrl-C` on the above, you are replacing the previous pattern with another one on-the-fly. Congratulations, you're live coding.

The `sound` function in the above is just one possible parameter that we can send to the synth. Below show a couple more, `pan` and `vowel`:

```
d1 $ sound "bd sn sn"
   |+| vowel "a o e"
   |+| pan "0 0.5 1"
```

NOTE: `Ctrl-C Ctrl-C` won't work on the above, because it goes over more than one line. Instead, do `Ctrl-C Ctrl-E` to run the whole block. However, note that there must be empty lines surrounding the block. The lines must be completely empty, including of spaces (this can be annoying as you can't see the spaces).

Note that for `pan`, when working in stereo, that `0` means hard left, `1` means hard right, and `0.5` means centre.

When specifying a sequence you can group together several events to play inside a single event by using square brackets:

```
d1 $ sound "[bd sn sn] sn"
```

This is good for creating compound time signatures (sn = snare, cp = clap):

```
d1 $ sound "[bd sn sn] [cp cp]"
```

And you put events inside events to create any level of detail:

```
d1 $ sound "[bd bd] [bd [sn [sn sn] sn] sn]"
```

You can also layer up several loops, by using commas to separate the different parts:

```
d1 $ sound "[bd ht lt, sn cp]"
```

This would play the sequence `bd bd bd` at the same time as `sn cp sn cp`. Note that the first sequence only has three events, and the second one has four. Because tidal ensures both loops fit inside same duration, you end up with a polyrhythm.

Try replacing the square brackets with curly brackets:

```
d1 $ sound "{bd ht lt, sn cp}"
```

This is a different way of specifying a polyrhythm. Instead of both parts taking up the same amount of time, each event within the second part takes up the same amount of time as the second part. You can embed these different forms inside each other:

```
d1 $ sound "{bd [ht sn, lt mt ht] lt, sn cp}"
```

You can make parts of patterns repeat by using ∗, for example the following example produces the same pattern as the previous one:

```
d1 $ sound "[bd*3, [sn cp]*2]"
```

Conversely, you can slow down patterns by using /, the following pattern plays part of each subpattern each cycle:

```
d1 $ sound "[bd sn sn*3]/2 [bd sn*3 bd*4]/3"
```

Beats per second

You can change the beats per second (bps) like this:

```
bps 1
```

If you prefer to think in beats per minute, simply divide by 60

```
bps (140 / 60)
```

Samples

All the samples can be found in the `samples` subfolder of the Dirt distribution. Here's some you could try:

```
flick sid can metal future gabba sn mouth co gretsch mt arp h cp
cr newnotes bass crow hc tabla bass0 hh bass1 bass2 oc bass3 ho
odx diphone2 house off ht tink perc bd industrial pluck trump
printshort jazz voodoo birds3 procshort blip drum jvbass psr
wobble drumtraks koy rave bottle kurt latibro rm sax lighter lt
```

Each one is a folder containing one or more wav files. For example when you put `bd/1` in a sequence, you're picking up the second wav file in the `bd` folder. If you ask for the ninth sample and there are only seven in the folder, it'll wrap around and play the second one.

If you want to add your own samples, just create a new folder in the samples director, and put `wav` files in it.

Continuous patterns

As well as making patterns as sequences, we can also use continuous patterns. This makes particular sense for parameters such as `pan` (for panning sounds between speakers) and `shape` (for adding distortion) which are patterns of numbers.

```
d1 $ sound "[bd bd] [bd [sn [sn sn] sn] sn]"
   |+| pan sinewave1
   |+| shape sinewave1
```

The above uses the pattern `sinewave1` to continuously pan between the left and right speaker. You could also try out `triwave1` and `squarewave1`. The functions `sinewave`, `triwave` and `squarewave` also exist, but they go between `-1` and `1`, which is often not what you want.

Transforming patterns

Tidal comes into its own when you start building things up with functions which transform the patterns in various ways.

For example, `rev` reverses a pattern:

```
d1 $ rev (sound "[bd bd] [bd [sn [sn sn] sn] sn]")
```

That's not so exciting, but things get more interesting when this is used in combination another function. For example `every` takes two parameters, a number, a function and a pattern to apply the function to. The number specifies how often the function is applied to the pattern. For example, the following reverses the pattern every fourth repetition:

```
d1 $ every 4 (rev) (sound "bd*2 [bd [sn sn*2 sn] sn]")
```

You can also slow down or speed up the playback of a pattern, this makes it a quarter of the speed:

```
d1 $ slow 4 $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

And this four times the speed:

```
d1 $ density 4 $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

Note that `slow 0.25` would do exactly the same as `density 4`.

Again, this can be applied selectively:

```
d1 $ every 4 (density 4) $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

Note the use of parenthesis around `(density 4)`, this is needed, to group together the function `density` with its parameter `4`, before being passed as a parameter to the function `every`.

Instead of putting transformations up front, separated by the pattern by the `$` symbol, you can put them inside the pattern, for example:

```
d1 $ sound (every 4 (density 4) "bd*2 [bd [sn sn*2 sn] sn]")
   |+| pan sinewave1
```

In the above example the transformation is applied inside the `sound` parameter to `d1`, and therefore has no effect on the `pan` parameter. Again, parenthesis is required to both group together `(density 4)` before passing as a parameter to `every`, and also around `every` and its parameters before passing to its function `sound`.

```
d1 $ sound (every 4 (density 4) "bd*2 [bd [sn sn*2 sn] sn]")
   |+| pan (slow 16 sinewave1)
```

In the above, the sinewave pan has been slowed down, so that the transition between speakers happens over 16 loops.

Mapping over patterns

Sometimes you want to transform all the events inside a pattern, and not the time structure of the pattern itself. For example, if you wanted to pass a sinewave to `shape`, but wanted the sinewave to go from `0` to `0.5` rather than from `0` to `1`, you could do this:

```
d1 $ sound "bd*2 [bd [sn sn*2 sn] sn]"))
   |+| shape ((/ 2) <$> sinewave1)
```

The above applies the function `(/ 2)` (which simply means divide by two), to all the values inside the `sinewave1` pattern.

Parameters

These are the synthesis parameters you can use

- `sound` - a pattern of strings representing sound sample names (required)
- `pan` - a pattern of numbers between 0 and 1, from left to right (assuming stereo)
- `shape` - wave shaping distortion, a pattern of numbers from 0 for no distortion up to 1 for loads of distortion
- `vowel` - formant filter to make things sound like vowels, a pattern of either `a`, `e`, `i`, `o` or `u`. Use a rest (~) for no effect.
- `cutoff` - a pattern of numbers from 0 to 1
- `resonance` - a pattern of numbers from 0 to 1
- `speed` - a pattern of numbers from 0 to 1, which changes the speed of sample playback, i.e. a cheap way of changing pitch

Pattern transformers

brak

```
brak <pattern>
```

Make a pattern sound a bit like a breakbeat

Example:

```
d1 $ sound (brak "bd sn kurt")
```

Beat rotation

```
<number> <~ <pattern>
```

or

```
<number> ~> <pattern>
```

Rotate a loop either to the left or the right.

Example:

```
d1 $ every 4 (0.25 <~) $ sound (density 2 "bd sn kurt")
```

Reversal

```
rev <pattern>
```

Reverse a pattern

Examples:

```
d1 $ every 3 (rev) $ sound (density 2 "bd sn kurt")
```

Increase/decrease density

```
density <number> <pattern>
```

or

```
slow <number> <pattern>
```

Speed up or slow down a pattern.

Example:

```
d1 $ sound (density 2 "bd sn kurt")
   |+| slow 3 (vowel "a e o")
```

Every nth repetition, do this

```
every <number> <function> <pattern>
```

Applies to , but only every repetitions.

Example:

```
d1 $ sound (every 3 (density 2) "bd sn kurt")
```

Interlace

```
interlace <pattern> <pattern>
```

Shifts between two patterns, using distortion.

Example:

```
d1 $ interlace (sound  "bd sn kurt") (every 3 rev $ sound  "bd sn/2")
```

Plus more to be discovered!

You can find a stream of minimal cycles written in Tidal in the following twitter feed: http://twitter.com/tidalcycles/

Acknowledgments

Special thanks to l'ull cec (http://lullcec.org) and hangar (http://hangar.org) for supporting the documentation and release of tidal as part of the ADDICTED2RANDOM project.