High-Level Musical Concepts in µO

Stéphane Rollandin
hepta@zogotounga.net

draft - 08 May 2013

Abstract

We comment here a short musical composition fragment programmed in the µO Smalltalk system with a few lines of code instanciating very high-level musical objects.

The emphasis in µO is on reifying musically meaningful concepts into objects that appear "real" in a practical sense, and on providing ways to organize them into music in a manner as natural as possible.

Hopefully you will get the taste of µO by the end of this paper.

Notation

In the following, the printed evaluation of a Smalltalk expression is represented following a ▶ symbol. When a graphic representation is available (a screenshot of a µO editor in most cases), it is displayed after a ▶. All code is written in `Consolas font`.

1. The code & the music

Besides is the Smalltalk source code for a 16-seconds composition fragment.

Its piano rendition is available online at:
http://www.zogotounga.net/zik/purdie%20variation.flac

2. The algorithm, explained

We start from a rhythm and a chord progression. Each chord in the progression is instanciated as a tetratonic mode and associated to a measure of the rhythm. This defines an harmonic framework.

We play a half-time Purdie shuffle[1] drum pattern over the rhythm, but there is a trick: instead of striking drums, we strike harmonic degrees of the underlying chord.

We complement the rhythmic melody with a couple of notes: the degrees 1 to 4, in turn, of the underlying harmony, on specific beats only; which beats are to be played depends on a Xenakis sieve.

1 http://en.wikipedia.org/wiki/Bernard_Purdie

```
rhythm :=
 #(M 2 ((2 2) 4) 2 ((2 2) 4.3) 2 ((4 2) 4)) sig.

rhythm ritardando.

rhythm bpm: 115.

mode := Mode D major.

harmony := rhythm asCanvas on: #downBeats place: {
 mode I7 asChromaticMode.
 mode bVII7 asChromaticMode.
 mode bIV7 asChromaticMode.
 mode bVI7 asChromaticMode.
 mode IV7 asChromaticMode.
 mode I7 asChromaticMode}.

(drumKit := GrooveDrumKit new)
 flavorAt: #bass
   put: (Stroke drum: -3 for: 2) mf;
 flavorAt: #snare
   put: (Stroke drum: 2 for: 1.5) ;
 flavorAt: #rideCymbal
   put: (Stroke drum: 4 for: 1) mp;
 flavorAt: #hiHat
   put: (Stroke drum: -9 for: 3) mf.

ride :=
 (HalfTimeShuffle purdie3 drumKit: drumKit)
   on: rhythm.

harmonicDrummer :=
 HarmonicDrummer new harmony: harmony.

phrase := harmonicDrummer drumPattern: ride.

degrees := #(1 2 3 4) cyclicGenerator.

rhythm asCanvas
 on: #beats
 onSieve: (3@@2 * (2@@1) * (4@@0))
 do: [:beat |
       phrase addNote:
            (((harmony bolAround: beat time)
              degree: degrees next)
               forte time: beat time; length: 3)].

phrase disturbTimes; disturbAmplitudes.
```

Finally we very slightly blur the melody, by randomly offsetting by small amounts note onsets and amplitudes.

2. The algorithm, detailed

2.1 `rhythm`

The rhythm of the melody is stored in variable `rhythm`. It is defined by the numbers

```
2 ((2 2) 4) 2 ((2 2) 4.3) 2 ((4 2) 4)
```

which mean:

 2 measures of 4/4,
 followed by 2 measures of 4/4, slightly faster,
 followed by 2 measures of 6/4, more exactly of 4+2/4
  (that's an additive time signature)

We apply a *ritardando* to the whole rhythm. This could be defined with arbitrary precision but in here we stick with the default `#ritardando` selector.

We also slightly accelerate the overall tempo by setting the BPM to 115 where by default it would be 120. This is the number of quarter notes per minutes (the reference being the duration of a quarter note at the very beginning of the rhythm, before the *ritardando*).

2.2 `harmony`

The mode is D major, stored in variable `mode`.

The chord progression is `I7 bVII7 bIV7 bVI7 IV7 I7`, but you can see that it is defined in a more complex way:

```
harmony := rhythm asCanvas on: #downBeats place: {
  mode I7 asChromaticMode.
  mode bVII7 asChromaticMode.
  mode bIV7 asChromaticMode.
  mode bVI7 asChromaticMode.
  mode IV7 asChromaticMode.
  mode I7 asChromaticMode}.
```

`harmony` is the harmonic framework. It is more than the chords progression: each chord symbol (such as `IV7`) is replaced by a full-fledged chromatic mode (or scale, in common musical parlance). This will allow us to get notes from the harmony from a plain integer, the note degree in the harmonic mode.

For example,

```
Mode D major IV7
▶ 'g b do4 f'
```

and so

```
Mode D major IV7 asChromaticMode degree: 2
▶ 'b'
```

The `rhythm asCanvas on: #downBeats place:` part means "take each downbeat in turn in `rhythm`, and have the corresponding harmonic mode start there". Of course each mode overrides the previous one, so at the end of the expression what is returned into variable `harmony` is the progression of six harmonic modes associated to six measures of `rhythm`.

2.3 `ride`

Now we can start grooving. Really: `HalfTimeShuffle` is a subclass of `Groove`, that is an object who knows how to populate a rhythm in a specific style.

For example:

```
HalfTimeShuffle new on: #((2 2) 4) sig
▶ D([#bassDrum1 0.45t0d0.5]
 [#closedHiHat 0.45t0d0.5]
 [#acousticSnare 0.26t0.17d0.17]
 [#rideCymbal1 0.45t0.33d0.17]
 [#rideCymbal1 0.45t0.5d0.5]
 [#acousticSnare 0.26t0.67d0.17]
 [#rideCymbal1 0.45t0.83d0.17]
 [#acousticSnare 0.55t1.0d0.5]
 [#rideCymbal1 0.45t1.0d0.5]
 [#acousticSnare 0.26t1.17d0.17]
 [#rideCymbal1 0.45t1.33d0.17]
 [#rideCymbal1 0.45t1.5d0.5]
 [#acousticSnare 0.26t1.67d0.17]
 [#rideCymbal1 0.45t1.83d0.17])L2.0
```

The above list is the string representation of a drum pattern, a half-time shuffle played over a single measure of a plain 4/4 time signature at 120 BPM.

So the expression

```
ride :=
  (HalfTimeShuffle purdie3 drumKit: drumKit)
    on: rhythm.
```

stores a drum pattern in variable `ride`.

But we do not want to play battery, only piano. So we need two things: a special drumkit, and a special drummer.

2.4 drumming on the piano

The drumkit tells the groove which symbols should be associated to its inner `#bass`, `#snare`, `#rideCymbal` and `#hiHat`.

So the part

```
(drumKit := GrooveDrumKit new)
  flavorAt: #bass
    put: (Stroke drum: -3 for: 2) mf;
  flavorAt: #snare
    put: (Stroke drum: 2 for: 1.5) ;
  flavorAt: #rideCymbal
    put: (Stroke drum: 4 for: 1) mp;
  flavorAt: #hiHat
   put: (Stroke drum: -9 for: 3) mf.
```

tells, for example, that when the groove hits a `#bass`, what it really means is that it strikes an integer, -3, for 2 seconds, mezzo-forte.

Of course only a special drummer can strike an integer !
A `Drummer` is an object that knows how to convert a drum pattern into a musical phrase. A drum pattern is kind of abstract: each drum is represented by a symbol (even the integer -3 is actually encoded as the symbol `#'-3'` into the drum pattern). To be able to get the music out of a drum pattern, we need a `Drummer`.

The default drummer would map drum symbols into MIDI events in channel 10, but here our symbols are numbers and what we want is get the corresponding notes, according to an underlying harmonic framework: each number will be mapped to the corresponding degree of the appropriate harmonic mode. This is what `HarmonicDrummer` does.

The expression

```
harmonicDrummer :=
 HarmonicDrummer new harmony: harmony.
```

instanciates an HarmonicDrummer based on `harmony`.
The expression

```
phrase := harmonicDrummer drumPattern: ride.
```

has the drummer interpret the `ride` drum pattern and stores the resulting musical phrase in variable `phrase`.

2.5 overlaying a sparse melody

To spice up the rhythmic melody we now have in `phrase`, we add some notes at specific points.

To get the note pitches, we iterate over the degrees 1 to 4, via a `Generator`. This is an object acting like a dispenser: you give it a list of objects and it gives them back one by one when you ask it what's `#next`.

```
degrees := #(1 2 3 4) cyclicGenerator.
degrees next
▶ 1
degrees next
▶ 2
degrees next
▶ 3
degrees next
▶ 4
degrees next
▶ 1
```

etc.

To get the note onsets, we do something similar to the way we built `harmony`, by walking the beats of `rhythm`. The expression

```
rhythm asCanvas
  on: #beats
  onSieve: (3@@2 * (2@@1) * (4@@0))
  do: [:beat |
        phrase addNote:
            (((harmony bolAround: beat time)
              degree: degrees next)
               forte time: beat time; length: 3)].
```

says something like: "at each beat in `rhythm`, provided that its index is accepted by the `XenakisSieve` of formula `3@@2 * 2@@1 * 4@@0`, add a 3-seconds forte note of the degree yielded by the `degrees` generator".

The Xenakis sieve (which I will not explain here)[2] is a `SieveFunction`, a function accepting an integer and returning a boolean, which is a nice and compact way to get patterns of 0 and 1 and is used here to define a rhythm.

Needless to say, the `3@@2 * 2@@1 * 4@@0` formula has been found by random attempts until it sounds good. No other reason for its precise form.

3. The algorithm, illustrated

Discarding the petty technical details we can see the `phrase` melody as a simple combination of three musical structures: `rhythm`, `harmony`, and `ride`.

These structures are quite complex Smalltalk objects with a rich protocol; we can play with them, explore them, display them and edit them in many ways.

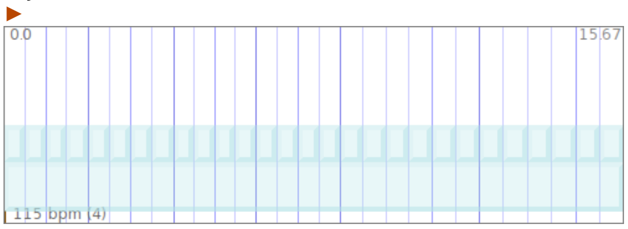
They can notably be interactively edited via specific graphical interfaces; although instanciated by code, they can be tweaked manually in minute details.

In the following we will introduce screenshots of the corresponding editors.

2 This type of sieve is described in detail by Christopher Ariza in
http://www.mitpressjournals.org/doi/pdf/10.1162/0148926054094396
(the µO implementation is based on that paper)

Here is our base rhythm:

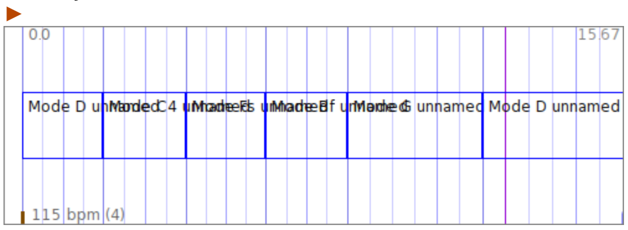
rhythm



The down beats, on beats and off beats are displayed as more and more lighter vertical lines. The blue boxes allow interactive stretching and moving of individual beats or whole measures, for *rubato* effects.

Here is our harmonic framework:

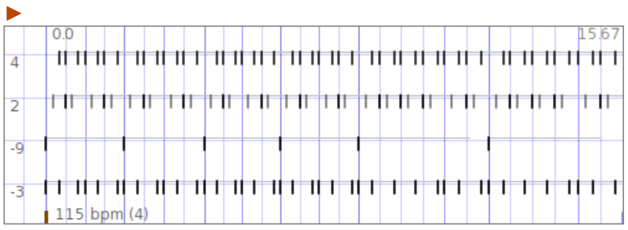
harmony



Okay this one is rather ugly... the bols editor needs some love. Still you can see how each harmonic mode is indeed in sync with a measure of `rhythm`.

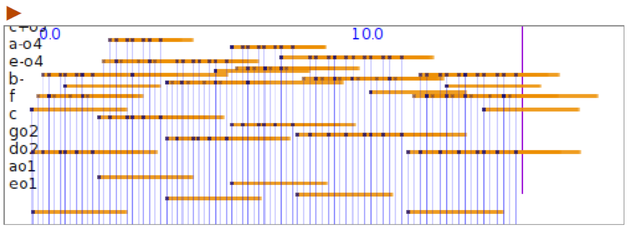
Here is our drum pattern:

ride



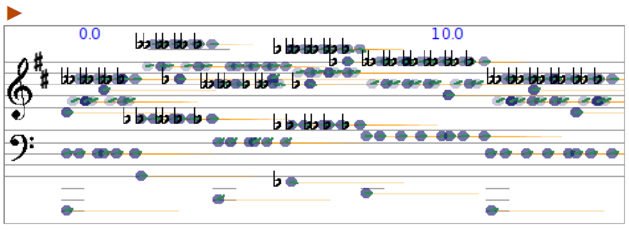
Last but not least, here is our composition:

phrase



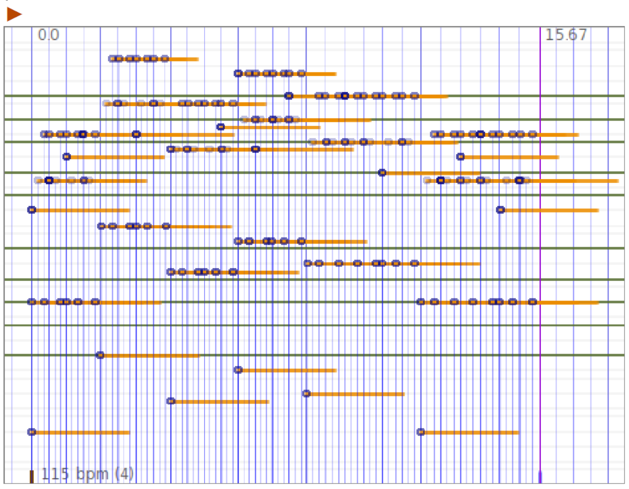
or, if you prefer a more traditional style of score representation:

phrase



With the right display options you can actually mix the representations:

phrase



4. The algorithm, explored

Let's see more in detail how to play with the high-level musical objects we are now familiar with.

4.1 `rhythm`

`rhythm` is an instance of class `RhythmicCell`[3]. It has been defined via the representation of a time signature.

```
#(4 4) sig
```

is a 4/4 signature where all beats but the first would be weak.

```
#((2 2) 4) sig
```

is the usual 4/4, with two strong beats alternating with two weak beats.

3 See "The representation of Rhythmic Structures in µO": http://www.zogotounga.net/surmulot/The%20Representation%20of%20Rhythmic%20Structures%20in%20muO.pdf

```
#((2 3 2) 4) sig
```

is a 2+3+2/4 time signature.

In our example,

```
#(M 2 ((2 2) 4) 2 ((2 2) 4.3) 2 ((4 2) 4)) sig
```

the rhythmic cell is defined as the succession of 2 measures of different cells: 4/4, a faster 4/4, and 4+2/4,

```
#((2 2) 4.3) sig
```

is a faster 4/4 because instead of a quarter (1/4) note, we use a 1/4.3 note as beat. We are familiar with 1/2, 1/4, 1/8 and 1/16 notes (respectively quarter, half, eighth and sixteenth in American english) but nothing prevents a computer to use any number as denominator; and that's what we do here. The effect is to increase the tempo of the middle section of `rhythm`.

The `M` at the beginning of `rhythm` signature forces the succession of cells that follows to be considered as a single measure, that is a single rhythmic cell. Without the `M`, the expression

```
#(2 ((2 2) 4) 2 ((2 2) 4.3) 2 ((4 2) 4)) sig
```

would instead return an instance of `RhythmicCanvas`. A rhythmic canvas is an object composed of several rhythmic cell; we will see more about it below.

The *ritardando* we apply to `rhythm` is the default `#ritardando` method defined in µO for all musical elements[4].

If we look at the implementation of method `#ritardando`, we can see (not here, you have to this in a Squeak image) that it is

```
self accelerandoBy: 0.840896415253715
```

so the general method for *ritardando/accelerando* is `#accelerandoBy:` which accepts a numerical argument. The default argument `0.840896415253715` is such that applying four times in a row `#ritardando` will end up decreasing the tempo by half, which is neat.

We can go further and see how `#accelerandoBy:` is implemented. It is:

4 `MusicalElement` subclasses are the main building block for musical structures in µO.
See "The Mixing Algebra of Musical Elements in µO": http://www.zogotounga.net/surmulot/The%20Mixing%20Algebra%20of%20Musical%20Elements%20in%20muO.pdf

```
self tempoFollow: (GenericEnvelope
 parabolicAccelerandoBy: aFactor
 from: self startTime
 to: self endTime)
```

It is worth spending some time on this implementation because it shows clearly a couple of crucial points of general importance in the overall design of µO:
- *provide simple entry points*
- *allow arbitrary level of detailing*
- *use high-level concepts whenever possible*
- *allow interactive edition at any level*
- *provide a generic API to all musical elements*

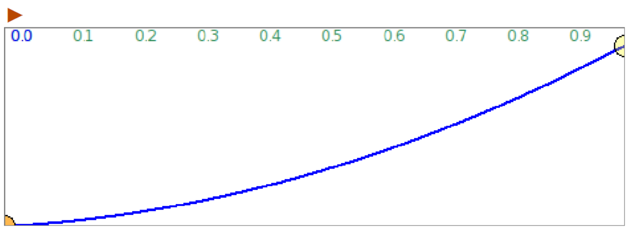
We just encountered the first two points (*provide simple entry points*, *allow arbitrary level of detailing*) by exploring the implementation of `#ritardando`, which is the simplest entry point, but can be replaced by `#accelerandoBy:` in order to define precisely via a numeric argument the amount of *ritardando*, and which itself can be replaced by the more general `#tempoFollow:`.

`#tempoFollow:` distorts the temporal structure of a musical element according to an envelope; this illustrates the third point (*use high-level concepts whenever possible*).

Envelopes (instances of class `GenericEnvelope`) are a very important class of objects in µO. They provide the base for all continous gestures, along with functions (instances of class `NFunction`)[5].

Here we can see that the default nature of a *ritardando **or** accelerando*, as implemented in `#tempoFollow:`, is parabolic. With argument 0.1 (the default 0.84 makes it almost linear) it looks like:

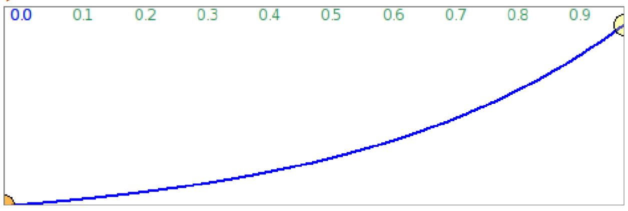
```
GenericEnvelope parabolicAccelerandoBy: 0.1 from: 0
to: 1
```



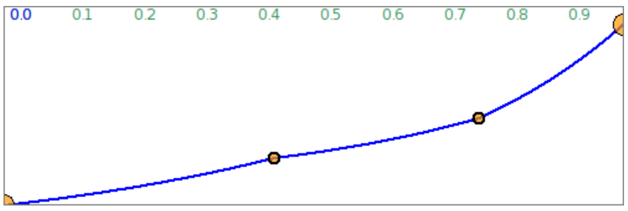
We could have a different, more dramatic, *ritardando* by using an exponential shape:

5 Functions and envelopes are intimately related: any function segment can be converted into an envelope, any envelope can be converted into a periodic or aperiodic function. Moreover, the interpolation curves between two envelope breakpoints are defined via functions.

```
GenericEnvelope exponentialAccelerandoBy: 0.1 from:
0 to: 1
```



And we could also have a custom type of *ritardando* by defining our own curve, either programmatically or interactively; the above picture is the display of an envelope editor. Working a little in this editor we can come up with something like



This illustrates the fourth point: *allow interactive edition at any level.*

The last point (*provide a generic API to all musical elements*) is illustrated in our example by the fact that `#tempoFollow:`, `#startTime` and `#endTime` are defined for all subclasses of `MusicalElement`.

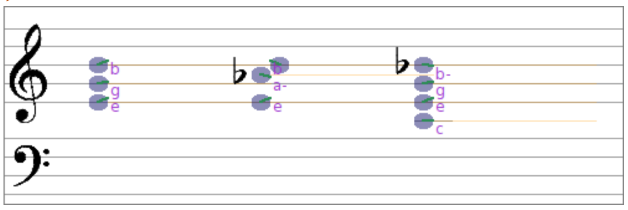
4.2 mode

Our `mode` is D major. This is an instance of `ChromaticMode`[6].

The harmonic background is build from `mode` chords.

`ChromaticMode` implements a wealth of messages such as `#I`, `#ii` `#vio`, `#bIV7` giving direct access to common chords of a given mode:

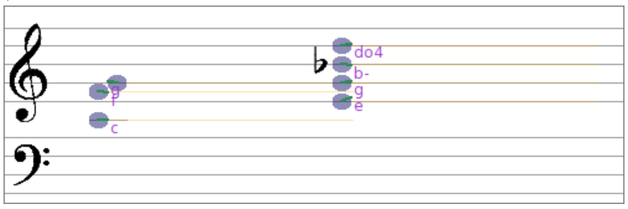
```
Mode C major iii, Mode C major bIV, Mode C major I7
```



6 See "Modes and Scales in µO": http://www.zogotounga.net/surmulot/Modes%20and%20Scales%20in%20muO.pdf

More unusual chords for which a method is not provided can be built on specific degrees by using methods `#I:`, `#II:`, `#III:`, etc.

```
(Mode C major I: #sus4), (Mode C major bIV: #m7b5)
```



`mode` is equal-tempered: its notes map directly to MIDI notes. We could have chosen another temperament, by coding for example

```
mode := Mode D major perfectFifths
```

or

```
mode := Mode D major.
mode temperament: ChromaticScale mean16
```

Similarly, we could have chosen another tuning than the default 440 Hz A:

```
mode A: 415
```

or

```
mode withBaroquePitch
```

The notes produced by a mode keep a reference to it, so they transpose accordingly:

```
Mode major withBaroquePitch degree: 4
▶ f
```

```
(Mode major withBaroquePitch degree: 4) mode tuning
▶ 9->415.0
```

4.3 harmony

A mode can be generated from any musical phrase. In our example we use `#asChromaticMode` to generate the harmony associated to each chord.

"Chromatic" here tells that the newly created mode scale is the usual 12-TET; this means that, when transposing a note in that mode one scalar step up, it will be raised by a semitone.

Alternatively we could have used `#asOctavicMode` which creates a mode whose scale is made by the initial note pitches cycled every octave. In that case scalar and modal transposition are equivalent and change a pitch by moving from one note to the next in the initial phrase. For the record there is also `#asMode` which creates a mode with a non-octavic interval; this will not be discussed here[7]. The main point to note is that a `Mode` in µO is a very high-level object that can encode subtle intervallic behaviors.

Let's get back to `harmony`. Using `#asChromaticMode` we created the modes responsible for interpreting the integers defining our melody as degrees. Now to map each of these modes to a specific part of `rhythm` we used the idiom

```
rhythm asCanvas on: #downBeats place: { the modes }
```

Later in the code, we added some notes to `phrase` this time with the idiom

```
rhythm asCanvas
  on: #beats
  onSieve: (3@@2 * (2@@1) * (4@@0))
  do: [ a block ]
```

These are methods to walk a `RhythmicCanvas` and do something at specific *places*.

A rhythmic canvas structures time according to the rhythmic cells it is made of. As a consequence, any point in time is associated to a unique beat by a rhythmic canvas.

Now reversely, we can iterate over the canvas beats. There is only a finite number of beats because a canvas has a beginning (the start time of its first cell) and an end (more complex to define: it is the start time of its last cell if the canvas has more than one cell, else it is the end time of its single cell).

A beat can be either `#strongest`, `#strong`, `#weak` or `#void`. In common western music this can be interpreted as:

`#strongest` -> down beat
`#strong` -> on beat
`#weak` -> off beat
`#void` -> rest place-holder[8]

This means that each beat in a rhythmic canvas has a qualitative aspect. We can refer to all beats that are strongest by the symbol `#downBeats`; similarly we have `#onBeats`, `#offBeats` and `#voidBeats`.

7 Again see "Modes and Scales in µO"
8 The void beat has another interpretation in Indian music, where it represents the *khali*.
See http://chandrakantha.com/articles/indian_music/khali.html

We can use these qualities to define other groups of beats according to their relative positioning: for example `#upBeats` (for all beats preceding a downbeat), `#backBeats` (the beats following a on-beat), or `#lastBeats` (the beats before a new time signature).

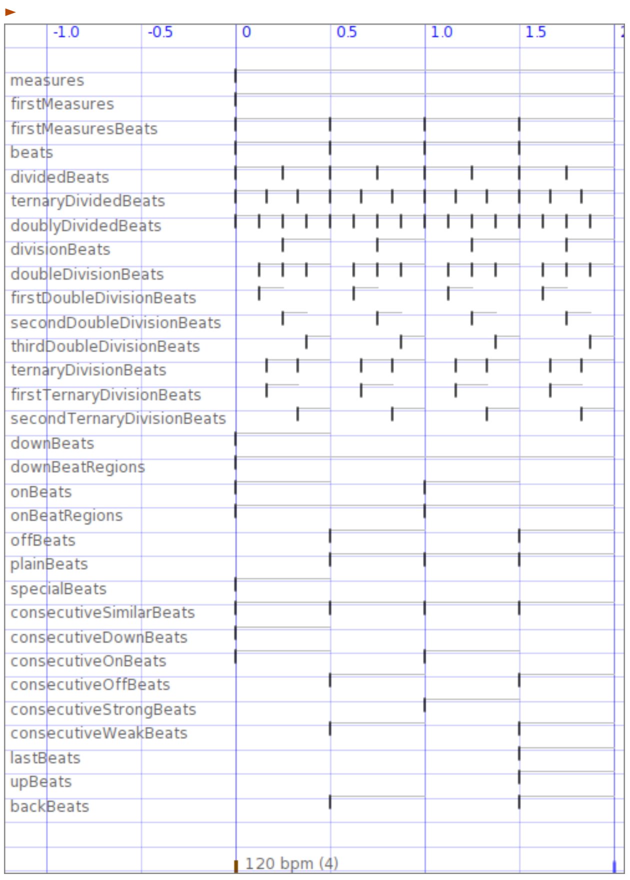
All theses symbols correspond to canvas *places*.

Other places span several beats: for example `#measures` (the repetitions of a rhythmic cell), `#meters` (the region structured by a rhythmic cell and its corresponding measures) or `#consecutiveOffBeats` (the region covered by consecutive `#weak` beats).

Finally some places refer to beats that do not exists as such in the canvas but are created on the fly; for example `#ternaryDividedBeats` (the beats in the triple division of the canvas).

Let's see what this looks like for a plain 4/4 time signature:

```
#((2 2) 4) sig asCanvas displayPlaces
```



`RhythmicCanvas` provides a protocol for iterating over these places, doing different kind of operations along the corresponding beats.

For example

```
on: #downBeats collect: [ a block ]
```

evaluates for each downbeat in the canvas the code in the block with the beat as argument. It returns the results as an array of objects.

In our code we already saw

```
on: #beats onSieve: a sieve collect: [ a block ]
```

which evaluates the block for all beats that also fit the sieve function provided as second argument; this way we can count over the places.

There are many more of these methods; this is not the place to discuss them. The important point here again is how we can work at a very high-level on a musical structure (the rhythmic canvas in this case) by using musically meaningful concepts.

In μO we want the code to tend to look like a declarative description of the structure of the composition; Smalltalk syntax makes it possible.

4.4 ride

The HalfTimeShuffle groove that makes most of our melody is precisely based on the notion of canvas places: in fact any Groove is defined in terms of places.

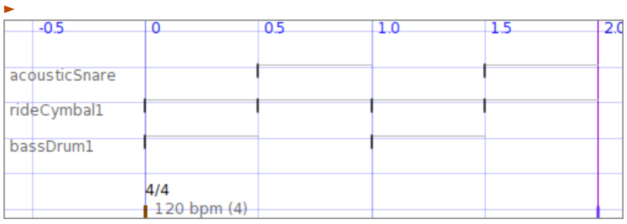
For example the SimpleRide groove is defined by its #run:over: method which source code is:

```
run: operator over: aRCanvas

  aRCanvas on: #onBeats
    do: (operator add: self bass).
  aRCanvas on: #offBeats
    do: (operator add: self snare).
  aRCanvas on: #voidBeats
   do: (operator erase: self snare).
 aRCanvas on: #beats
   do: (operator add: self rideCymbal).
```

The above defines the most basic pop/rock drum pattern:

```
SimpleRide on: #((2 2) 4) sig
```

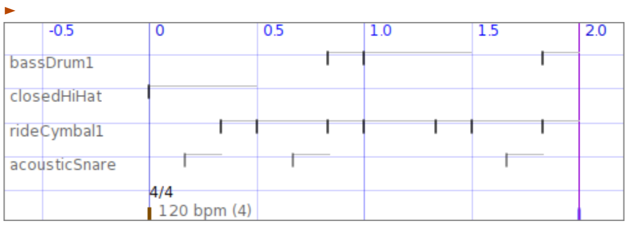


A groove need not be encoded in a specific class such as SimpleRide or HalfTimeShuffle. It can also be created on a *ad hoc* basis via a small domain-specific language interpreted by class GrooveOnDemand.

For example:

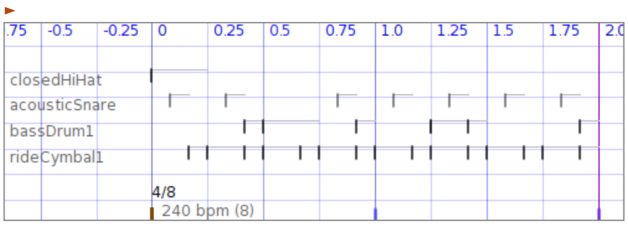
```
groove := GrooveOnDemand with:
      #((onBeats addLouder: bass)
        (downBeats erase: bass)
        ((beats TDb2) add: rideCymbal)
        (downBeats erase: rideCymbal)
        (downBeats add: hiHat)
        (TDb1 addGhost: snare)
        (TDb2 onMSieve: 2 2 add: bass)
        (onBeats atCounts: 2 add: bass)
        (TDb1 atCounts: 3 erase: snare))
```

```
groove on: #((2 2) 4) sig
```



Very complex patterns can be build easily this way, independently of any time signature; let's apply the above groove to another rhythm:

```
groove on: #((2 3 3) 8) sig
```



4.5 phrase

We will end this tour of some of the more important high-level objects in μO with the musical phrase, which is the object eventually holding our melody.

phrase is an instance of MusicalPhrase, which is a MusicalCollection of MusicalNote-s.

Musical notes can be mapped into MIDI data[9], Csound score or OSC messages. By itself a MusicalNote is format-agnostic: it provides pitch (possibly structured in

9 The audio file referenced at the beginning of this paper has been synthesized by a VST instrument playing the Salamander Grand Piano soundfont, available at http://rytmenpinne.posterous.com/pages/salamander-grand-piano-46556

reference to a mode), onset, length and amplitude. Pitch and amplitude can be envelopes; moreover pitch envelopes are transposable according to the note mode. This allows Indian *meends*[10] to be easily programmable.

A specific format allows a compact specification of complex phrases; this is discussed elsewhere[11].

5. References

The home page for µO (aka Musical Objects for Squeak) is http://www.zogotounga.net/comp/squeak/sqgeo.htm

10 http://www.itcsra.org/alankar/meend/meend_index.html

11 See "String Representation of Musical Phrases in µO" http://www.zogotounga.net/surmulot/String%20representation%20of%20musical%20phrases%20in%20muO.pdf