Mercury: a live coding environment focussed on quick expression for composing, performing and communicating

Timo Hoogland
Music Technology Faculty, HKU University of Arts Utrecht
info@timohoogland.com

ABSTRACT

In this paper the livecoding environment Mercury is introduced. An interpreted language that is designed with the focus on quick and hands-on composing, performing and communicating of live-coded music and sound. Mercury provides the performer with a highly abstracted programming language for music and sound complemented with visuals. By allowing for a higher level of abstraction, the coder does not need to write large amounts of code and the comprehension by the audience is improved. Combining the sound with visual elements adds value to this understanding. The environment incorporates generative and transformational list functions to assist in algorithmic composition processes. Mercury has been used for performances at various livecoding events and proven itself as a live-performance tool.

1 Introduction

Livecoding is the art of programming software, in many cases from scratch, as a performance for auditive and/or visual output. During these performances the code is projected for the audience to read along. In little time the performer has to write many lines of code in order for the music to be generated. All of the musical parameters are written in code and manipulated by the code while running in real time. During the performance code has to be written for many layers in the music, on micro level, for notes and timbre of instruments, and also on macro level, for chord progressions, tension in the music and rhythmic structures. Meanwhile the performer tries to "maintain enough variety to keep the audience engaged" (Collins 2011), with the risk that some might find the music not up to their standards (Burland and McLean 2016). Given all this, a livecoding performance is a demanding task with a high cognitive load, "leading performers to rethink the design of their language interfaces" (McLean and Wiggins 2011). In this paper the live-coding environment Mercury is introduced. An interpreted language that is designed with the focus on quick and hands-on coding for live performances. Mercury provides the performer with a highly abstracted programming language for music and sound combined with visuals. The language is designed with two end-users in mind (Blackwell and Collins 2005). First and foremost the livecoder, but secondly also the audience. Firstly, the reader will be introduced to the general syntax of the environment, to get familiar with the basic principles of the code for better understanding all the examples throughout the paper. Secondly all design choices for the language, functions, interface and sound-design will be described based on the filosophy to provide the performer with a high abstracted language and keep transparency towards the audience. Concludingly ideas for future work will be discussed.

2 General syntax

Mercury is an interpreted language, with a code structure similar to languages like javascript and ruby. The language is written as highly abstracted and uses the Max/MSP programming language for its audio engine and openGL visual engine. Every line of code is interpreted as a new line and it therefore does not need to be ended with a semicolon. The code is evaluated from top to bottom, and variables must be declared before the instantiation of a new instrument, if that instrument needs to use the variable. All functionalities are sorted in three categories. To create a list of values, the code is started with the command `ring`. A single variable is also created with this command, and is basically a list with only a single element. To produce sound a new instrument can be created, such as a sampler or synthesizer, with the command `new`. If a parameter from an instrument needs to be changed, the `set` command can be used. The entire code needs to be run in order to hear it take effect. This is done by using the shortcut key `CMD + R`. This all helps to give it an "immediate code and run aesthetic" (Collins et al. 2003).

2.1 Ring arrays

The ring is a type of array that wraps back when a lookup index exceeds the length of the array. This is also known as a circular array or circular buffer. The reason for the choice of this datatype is discussed in chapter 5.1. All rings are global variables, and can be used by any instrument. The instantiation of a variable with the command `ring` must always be followed by a name. This name can be any word, if at least two or more letters and without spaces. When a ring is declared by manually providing values, the name of the variable is follow by a space and the list is typed within parenthesis. When using a function to generate a ring, the third argument is the name of the function directly followed by parenthesis enclosing the arguments.

```
ring bassLine (0 3 12 7 5 10 3)
ring bassGain (0.7)
ring bassRhythm random(8 2)
```

In the examples above there are three rings added to the code, named `bassLine`, `bassGain` and `bassRhythm`. The `bassLine` is a ring consisting of seven integers. The interpreter knows it is a ring because there is a space between the variable name and the parenthesis. The second variable is a single floating-point value. This single value is also writing a ring, only consisting of one single value. The third variable `bassRhythm` is a ring generated with the `random()` function.

2.2 New instruments

When using the command `new`, to generate an instrument, it must be followed by the keyword of the instrument-type. For example `synth` to generate a synthesizer, or the keyword `sample` to use a prerecorded sample from the file library.

```
new synth saw gain(0.8)
new sample kick_909 time(1/4)
```

As shown in the example above both the synth and the sample have one keyword. This keyword determines the basis of the sound and is written as an extra argument after a space. In the case of the synth the basis is a saw (sawtooth waveform), in case of the sample it is an audiofile named `kick_909`. The rest of the parameters of that instrument are written as a function. You can set as many parameters for an instrument in the same line as you want. Both sound-generators have a large list of various functions and arguments that are used to change there sounds and compositional parameters such as timing and pitch. These functionalities will be discussed in more detail in chapters 2.5 and 7.

2.3 Set parameters

The length of a code line can grow large when a lot of parameters have to be set for an instrument. This is where the `set` command comes in, combined with the `name()` function for an instrument. When an instrument is instantiated, by default it belongs to the `all` group. Using the `set` command, followed by `all`, all other parameters can be called upon with their function.

```
new sample kick_909
new synth sine
    set all time(1/16) gain(0.4)
```

In the example above the synth (with a sinewave waveform) and the sample (with the `kick_909` audiofile) has no parameters set on there own line of code. Using the `set all` command followed by functions for parameters (time and gain), these functions are called for both instruments.

```
new sample kick_909
new synth sine name(wave)
    set wave time(1/16) gain(0.4)
```

When the synth in the example above is created with the function `name(wave)`, the `all` group will be overwritten. Now functions can be called for this instrument specifically. Optionally the user can give multiple instruments the same name to call functions with the same line of code.

2.4 Global settings

Besides these three basic commands, there are also a set of global parameters that can be changed to effect the entire composition and all running instruments. The syntax for these parameters is as follows; type the keyword for the variable and then a space followed by the argument(s). Multiple arguments are separated by spaces. An example for setting the tempo in beats per minute, add a seed for random number generators and syncing all midi notes to a certain scale and tonic:

```
set_tempo 108
set_random_seed 4532
sync_to_scale minor_harmonic A#
```

2.5 The timing system

Writing for-loops or while conditions to create a repeating melody or beat is not necessary in Mercury. The environment is continuously running a loop, and all instruments are synced to that loop using the `time()` function. All timing is therefore relative to the global variable `set_tempo`. This function determines the timing at which the instrument gets (re-)triggered. For example, set the tempo at a value of 120 (`set_tempo 120`). The 120 in set_tempo is in beats per minute, measured in quarter notes. When a time of 1 is set (`time(1)`), the instrument gets triggered every bar, which is 4 times the time interval of the beats per minute (every 500 milliseconds). This results in one trigger every second (`4*500 = 1000 ms`). A time of 4 (`time(4)`) means trigger this instrument every 4 bars (`4*(4*500) = 4000 ms`). A smaller timing can be made as a floating-point value or a fraction. For example `time(1/4)` will result in a quarter-note beat, which corresponds to the beats per minute of 120 (every 500 ms). Complex timings can be made with all other rational numbers. For example a time of 5/13 (`time(5/13)`) will result in a timing interval of 769.23 ms, (`(60000/120*4) * (5/13)`). For an in-depth explanation on the use of polyrhythms with the `time()` function the reader can refer to section 6.2.

```
set_tempo 120
new sample kick_909 time(1/4) //this kick is played every quarter note
new synth sine time(4) //this sine synth is triggered every bar
new sample hat_909 time(5/13) //this hihat is triggered every 769.23 ms
```

3 Quick expression

While many environments for live-coding music already focus on performance by utilizing a very extensive and impressive library of functions for sounddesign and composition tools, they still require many lines of written code. When evaluating livecoding by design features from the Cognitive Dimensions of Notation framework (Blackwell and Green 2002) it comes of with “low visibility, low closeness-of-mapping and low role-expressiveness” (McLean and Wiggins 2011). Although this gives those environments a lot of flexibility and options, it requires the performer to either prepare a lot of code-snippets for functionalities, or write many lines and memorize all syntax and functions through hard and intensive practice (Nilson 2007). Using abstraction allows the performer to “focus on the compositional structure behind the piece” (McLean and Wiggins 2011). Through a questionnaire audience members of live coding events have stated that the “events can get boring quickly due to little change over time” or that “the music quality is not up to their standards” (Burland and McLean 2016). Some also stated to be disappointed when code is already written prior to the performance and shown on screen. Keeping the audience engaged to the performance is a difficulty Collins also acknowledges (Collins 2011). Because of this, the end-user in mind for Mercury is first and foremost the live-code performer, but secondly the audience, who will be the consumers of the music generated with Mercury. Mercury will try to “meet the needs of this user population in whatever way is most appropriate.” (Blackwell and Collins 2005).

Mercury takes care of a lot of basic functionalities that live-coders use during their performance. Among others these functionalities include various pre-written sound synthesis instruments, a sampler, sequencing, (algorithmic) composition techniques and audioprocessing effects. This takes form as a high level of abstraction (high visibility in terms of CDN) that allow the performer to quickly generate their first sound when on stage, move on to compositional aspects of the performance and combine given functions as they please. With this the performer can “explore electronic music through its natural tool, the computer” (Nilson 2007). Because Mercury is running a loop, creating the first sound will immediately turn into the start of a sequence. All parameters for the sounds have initial values, so that the output of a sound is guaranteed.

To preserve the speed of typing code, Mercury has a few shortcut keys built-in to navigate through the code and edit quickly. These shortcuts are chosen in such a manner that the coder can leave their hands at the starting-position keys (left index finger at `F`, right index finger at `J`). Navigating shortcut keys that replace the arrow keys are a combination of `Alt + A, S,`

D, W. With the commands Alt + Q or Z the user jumps to the top or bottom of the editor. The backspace can be used with Alt + B. Using the command Alt + C duplicates the current line to the line beneath it. With the command Alt + X the user can remove the line where the cursor currently resides. A common used coding trick is the so called commenting of a line to disable its use when running the code. In Mercury you can comment or uncomment a line with the shortkey Alt + /.

4 Transparency towards the audience

Besides the reasons for quick hands-on performance, another reason for the high level of abstraction in Mercury is the transparency towards the audience. A typical issue during live-coding performances is the large amount of code projected on the screens behind the performers with the gesture of openness. The amount of code is obfuscating for audience members and because of that audience members can feel excluded from the performance "by a gibberish of code in an obscure language"(McLean and Wiggins 2011). This while "many of the participants enjoy the opportunity to learn from the projected code" (Burland and McLean 2016).

When working with a high level of abstraction it is easier for the audience members to follow what is going on in the music and what the performer is working on. Using musical words for parameters, settings and functions result in more transparency. For example using a term like `set_tempo` gives a better indication on what it is about then `set_bpm`, which might be a normal abbreviation for musicians, but not so much for non-musical spectators. Other methods like `note()`, `gain()` and `filter()` clearly indicate what their function is. For list processing techniques there are clear function names such as `scramble()`, `mirror()`, `join()`, `random()` and many more described in chapter 6.

Another feature in Mercury to preserve transparency in coding is the automatic adjustment of text size on the screen and the character and text lines limitations. When the performer adds more lines of code, the text will automatically resize to fit the screen, so all text remains in sight of the audience and the coder. This also happens when more characters are typed on a line. To make sure that the text will always be readable, Mercury does not allow for more then 30 lines of code. These restrictions will help the audience keep an understanding of what is going on, and will also force the performer to delete lines of code when working on a new part in the composition.

To complement the visual part of the already projected code, Mercury also produces visual objects closely related to the audio (and connected to the instruments) during the performance. These elements are geometric shapes that appear when an instrument is created based on the type of instrument. Properties of the sound such as triggering, amplitude and frequency are closely linked to respectively the movement, scaling and rotation of the shape. These visual features will help the audience to understand the code better and also make for a more interesting live performance to keep the audience engaged and entertained. Something that had already been practiced back in 1938 by Fischinger's "An Optical Poem" in which he gave life to music through the use of geometric shapes and many colors (Fischinger 1938). Also Barri (2009) wrote in his paper that "by letting the audience literally see how the composer works and how the composition is constructed, great value could be added to the listening experience". See figure 1 for a screenshot of Mercury.

5 Algorithmic & generative composition strategies

Livecoding is in many cases about improvising, experimentation and conversation with music and sound (Nilson 2007). The livecoder has practiced a lot and reuses the gained experience, knowledge and written code to piece together new code, similar to improvised, in-the-moment, jazz performances (Blackwell and Collins 2005, @Burland2016). A live-coder might write melodic or rhythmic patterns from heart, but is more likely to utilize (pseudo)randomness and algorithmic composition techniques to generate patterns in melody and time. The techniques present in Mercury to make algorithmic composing possible are ring-arrays, fuzzy-logic, seeding of random number generators, scale mapping, and various list processing techniques. These features build upon the idea Spiegel describes as "a basic library consisting of the most elemental transformations which have consistently been successfully used on musical patterns" (Spiegel 1981).

5.1 Using circular arrays

When writing melodies or rhythms a composer can think of these in terms of patterns. A sequence of numbers that will be repeated in time and represent the pitch of a note, the rhythmic structure or something else. To allow for continuously looping through a sequence of numbers, Mercury uses circular arrays, in short called rings. By applying a modulus operation on the given index value, wrapping back to the start, a ring-array will always return a value from its array. This is similar to what Sorenson describes as "by changing the modulus it is possible to loop subsections as a method of motivic development" (Sorenson and Brown 2007). Every instrument in Mercury uses an internal counter. This counter increases

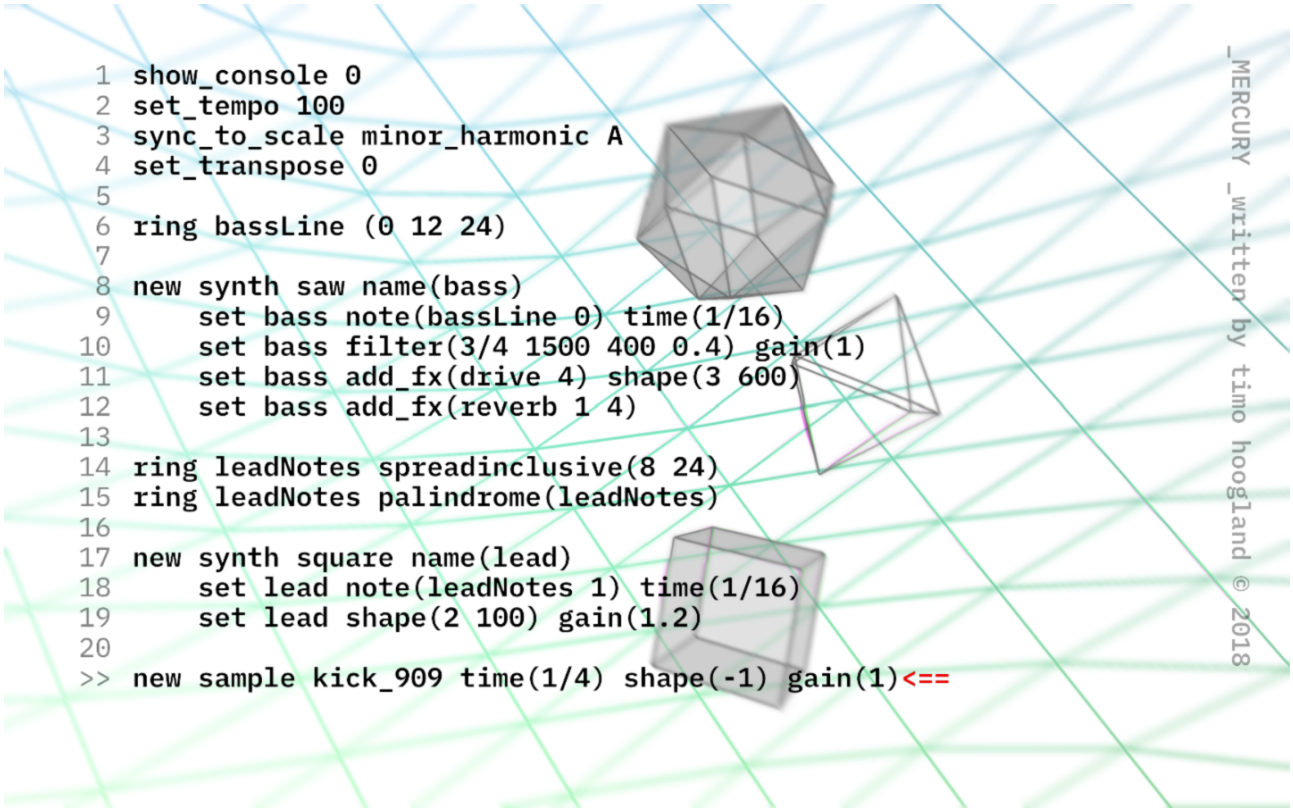

Figure 1: *A screenshot of the Mercury environment, showing three instruments written as code (sawtooth bass, square wave lead and 909 kick) and represented by three various platonic solids.*

with one every time the instrument triggers its sound. The counter value is used to retrieve the next value from the ring. This method allows the coder to generate arrays of arbitrary lengths and still hear a note play every time the instrument triggers. Consider the following code:

```
set_tempo 120
ring someNotes (3 7 12)
new synth saw note(someNotes) time(1/16)
```

The synthesizer is triggering its sound every 16th note, simultaneously incrementing the counter by one. The first three times this results in the index values `0, 1, 2`. The fourth time the counter is at 3, but the ring only has three values. Therefore the current count modulus the length of the list returns an index value of 0. After eighth notes, the played notes are `3 7 12 3 7 12 3 etc`. In all the following examples note values between 0 and 12 are used. This might seem odd if the reader comes from the standard midi-notation. How the note system works in Mercury will be described in chapter 7.1.

When creating rhythmic patterns, or beats, a ring with zeroes or ones can be declared. The 1 will represent a `TRUE` value, and result in the triggering of the instrument, a 0 represents a `FALSE` value and will be ignored. For example:

```
set_tempo 120
ring aRhythm (1 0 1)
new sample hat_909 beat(aRhythm) time(1/16)
```

The sample `hat_909` is counting in a 16th beat division. Similar to the previous example, every count is used as an index for the ring. After 8 beats, this results in the following pattern: `x - x x - x x -`.

In the case a rhythmic pattern is used in conjunction with a melodic pattern, the triggering of the instrument also increments a second internal counter. That second counter is used as lookup value for the melodic pattern. In the example below the same rhythm is applied as in the example above. This will result in an identical rhythmic pattern. When the instrument gets triggered the second counter is incremented and looks up the next note value from the `someNotes` ring. The final result over 8 beats will be: `3 - 7 12 - 3 7 - 12`. A dash means the instrument is not triggered.

```
set_tempo 100
ring aRhythm (1 0 1)
ring someNotes (3 7 12)

new synth sine name(wave)
    set wave note(someNotes) beat(aRhythm) time(1/16)
```

The rhythmic and melodic patterns can also be truncated at a certain length. This is done by adding a second argument to the `beat()` function, specifying the reset time (using the same timing-notation as the `time()` function). In the following example the instrument plays a pattern of four notes, the counter resets every bar and the time is set to quarter notes. This will result in the pattern: `0 3 7 0 0 3 7 0`, after two bars.

```
ring someNotes (0 3 7)
new synth saw note(someNotes) time(1/4) beat(1 1)
```

5.2 Fuzzy logic

Besides static rhythms constructed of zeroes and ones a more algorithmic approach will be to use probabilities to generate rhythmic patterns. As discussed by Collins (2001), using lists with probabilities is a way to "generate subtle variations without breaking with the consistency of a style". These probability lists, also referred to as templates, are constructed with floating-point values from 0 to 1. Here a one means, always play this note, a zero means don't play the note, and all values in between are a probability the note will or will not get played. This fuzzy logic method is implemented in the language of Mercury. Consider the following code:

```
set_tempo 90
ring fuzzy (1 0 0.3 0.2)
new sample kick_909 beat(fuzzy) time(1/16)
```

The kick sample will always be played on the first count (every quarter note), will never be played on the second count (2nd 16th note), and on the third and fourth 16th note will play respectively 30 and 20 percent of the time.

5.3 Seeding random number generators

Random values and functions that use Random Number Generators (RNG's), play a major role in algorithmic composition. Besides using them to generate rhythms such as with the fuzzy logic method, the can for example also be used to generate pitch values for melodies. An issue with random values, is that when re-running the code the values will be different then before. This is of course expected behaviour from a random number generator, but it might be leave randomly generated values intact every time the code gets evaluated. Or generate recurring random values, therefore create a form of pseudorandomness. A solution to this problem is using a seed for the RNG. The seed can be any value other than zero. A seed of zero results in using the system-time as input. The RNG outputs a uniform distribution.

```
set_random_seed 8429
ring randomNotes random(8 12)
```

In the example above the number `8429` is used as the seed for all the functions in the code that use. The list function `random()` will return a ring of 8 values, with random numbers between 0 and 12 (excluding 12). This will always result in the ring `(11 3 6 6 2 0 0 10)`, when using `8429` as a seeding value, and the random function has not been called earlier in the code.

```
set_random_seed 8429
ring moreNotes random(3 12)
ring randomNotes random(8 12)
```

Here you can see that an extra ring is added with a `random()` function. Therefore, when running this code, the `moreNotes` ring will have the first three values `(11 3 6)`, and the `randomNotes` will be filled with `(6 2 0 0 10 9 10 3)`. The reader will see that the first eight values are equal to the values from the previous example.

5.4 Scale mapping

Working with random numbers is useful, but mapping these values directly to corresponding MIDI-notes of that pitch will always result in a chromatic scale (all integer midi-pitches are possible). To allow for various scales, with a different tonic, a `sync_to_scale` function is included. The function must have one argument, namely the name of the scale, but can have a second argument for the tonic (by default c). The function will map the used midi-pitch of all instruments to the closest pitch from the selected scale according to a lookup-table of predefined scales. This function is in some ways similar to Ableton Live's scale midi-effect ("Live Midi Effect Reference, 23.6 Scale" 2018).

```
sync_to_scale minor D

ring someNotes (0 4 7 9)
new synth saw name(lead)
    set lead notes(someNotes)
```

In the code above the scale is set to d-minor. The ring `someNotes` contains 4 note values. The performer should always try and think of these values as the 12 semi-tones in a scale, where 0 is the tonic, 7 is the perfect fifth, and 12 is the octave. In this case the scale is set in minor. The internal lookup table for minor gives the following 12 values `(0 0 2 3 3 5 5 7 7 8 8 10)` that belong to the minor scale. The chromatic note values are then used to look for a corresponding value in this table to map to that scale, which will result in the new note list `(0 3 7 8)`. The reader will see that the notes that do not belong in the minor scale (4, a major third, and 9, the major sixth), will be mapped to the scale values 3 and 8. The last step is offsetting the list to match the tonic of the scale, D. This means all values in the list will get 2 added to them, resulting in `(2 5 9 10)`.

6 Composition techniques toolset

Besides algorithmic composition techniques to generate sequences and patterns, a strong toolset for composers also consists of techniques to transform those patterns or construct other patterns in different ways than with a RNG (Sorenson and Brown 2007; McLean and Wiggins 2010). Some of these techniques are forms of list processing functions such as reversing (retrograde), inversion, combining and duplicating. Others are techniques to generate lists with random values or a sequence. Many of those techniques are found in algorithmic-composition genres such as twelve-tone and (total) serialism (Magnuson 2008).

6.1 Ring transformations

The ring functions can be sorted into two categories: the generating ring functions and the transformational ring functions. The generating ring functions are spread, spreadinclusive, fill and random. Functions for ring manipulation are shuffle/scramble, reverse/mirror, invert/flip, rotate/turn, duplicate, clone, palindrome, combine/join, thin/unique and merge/add. Some have multiple names for the same function, because programmers come from different languages and backgrounds where similar functions have different function names. While many of the functions are self-explanatory, some might need some extra explaining.

The clone functions takes one ring as the first argument and after that a number for every clone that has to be made from the ring. The number specified will be added to the entire clone of the ring. In the following example the resulting output of `clone()` in theClones ring will be `(0 5 9 12 0 5 9 12 -5 0 4 7 7 12 16 19)`.

```
ring shortList (0 5 9 12)
ring theClones clone(shortList 0 0 -5 7)
```

The palindrome function returns a ring constructed of the specified ring as argument, combined with that same ring reversed. The result from the example below is `(0 3 7 9 12 12 9 7 3 0)`.

```
ring shortList (0 3 7 9 12)
ring thePalin palindrome(shortList)
```

When using the merge/add function, two rings are provided as arguments. Every value from both rings with the same index is summed. The result is a new ring with the same length as the longest ring from the input. If one of the rings is shorter than the other, extra zeroes are appended. In the next example the result of the merge will be `(1 1 0 1 0 2 0 1 1)`.

```
ring firstBeat (1 0 0 1 0 1)
ring secondBeat (0 1 0 0 0 1 0 1 1)
ring mergedBeat merge(firstBeat secondBeat)
```

The generative ring function `fill()` is used to fill a ring repeatedly with a specified values. The arguments are provided in number pairs. The first number represents the number to be added to the ring and the second value represents the amount of times that number will be added. In the next example the resulting ring will be `(0 0 0 0 0 0 7 7 -1 -1 -1 -1 12 12 12 12)`.

```
ring aFilledPattern fill(0 6 7 2 -1 4 12 4)
```

6.2 Creating polyrhythms with timing

As described in the first chapter about the general syntax, the timing system in Mercury allows for complex rhythmic patterns. Using the `time()` function sets the timing interval between every trigger of the instrument. When combining two or more instruments, intricate polyrhythms can be achieved. For example using a `3/8` against a `2/8` timing, at a tempo of 120 bpm, will result in an instrument triggered every 750 milliseconds, and an other instrument triggered every 500 milliseconds. Visually this will look like the following pattern:

```
x - - x - - x - - x - - x - - x - - etc.
x - x - x - x - x - x - x - x - x - etc.
```

Using other ratios as values for the time function will result in more complex polyrhythms. For example combining a `2/8` with a `5/6` results in a trigger every 500 milliseconds and a trigger every 1666.67 ms (in 120 bpm). Writing this down as a pattern will look like this:

```
x - x - x - x - x - x - x - x - - x - - x - - x - etc.
x  -  -  -  -  x  -  -  -  -  x  -  -  -  -  x  - etc.
```

When combined with the `beat()` function, every trigger of the timing will look up the next value from a ring provided to beat. This allows for combination of pre-written patterns, combined with fuzzy logic, combined with complex timing.

```
set_tempo 100

ring rhythmPattern (1 0.9 0.2)
new sample snare_909 time(3/16) beat(rhythmPattern)
new sample hat_909 time(1/8)
```

The example above shows a polyrhythmic pattern of 3 16th notes against 1 8th note. But on top of that the snare also has a ring provided to the beat function to lookup probabilities for triggering events. This can result in the following pattern:

```
x - - x - - - - - x - - x - - x - - x - - - - - x - - x
x - x - x - x - x - x - x - x - x - x - x - x - x - x -
```

7 Synthesizer & sampler

As described in chapter 3, Mercury's instruments already come with preset parameters to guarantee sound when adding an instrument to the code. But these presets are not always what the performer might want to hear. Therefore all parameters and sound effects can be set with functions. The instruments are divided in two categories, namely synthesizers and samplers. The samplers trigger one-shot samples from the audiofile library included or provided by the performer. The synthesizers make use of wavetable-synthesis.

7.1 2-dimensional notation

Mercury uses the standard MIDI notation, but with a slight twist. The notes are described in a 2-dimensional system, where the x-axis is the interval in semi-tones (0 to 11), and the y-axis is the octave offset (-3 to 7). In order to describe a pitch, the `note()` function is used and given two arguments, the interval between 0 and 11, and the octave. The origin of the coordinate system, `note(0 0)` corresponds to MIDI value 36 (when `sync_to_scale` is not applied) and this is also the default note value. `note(0 2)` corresponds with MIDI value 60 (C4, middle C, 261.626 Hz). If no octave value is provided, the default 0 is used.

```
new synth saw name(bass)
    set bass note(4 0) time(1/8)
```

In this example the synth is playing the fourth semi-tone in the octave starting at 36. This results in the note 40, or E2. If an interval value lower than 0 or higher than 12 is used, the interval wraps between 0 and 11 and will respectively decrement or increment the octave. Consider the following example:

```
ring aMelody (0 -1 -5 0 7 12 13 12)
new synth saw name(lead)
    set lead note(aMelody 1) time(1/8)
```

The ring `aMelody` has interval values both lower and higher than the standard 0 to 11 range. Besides that, the octave is set at 1. The result of this melody in standard MIDI notation will therefore be: `48, 47, 43, 48, 55, 60, 61, 60`.

The main reason to use this system of notation for notes is to be able to re-use lists of values. These values can then be used as the basis for other list processing techniques and form new melodies for other instruments. These instruments can then also easily offset the notes to different octaves.

```
sync_to_scale major C

ring bassLine (0 7 12)
new synth saw name(bass)
    set bass note(bassLine 0) time(1/16)

ring leadLine clone(bassLine 0 12 7 4)
new synth square name(lead)
    set lead note(leadLine 1) time(1/16)
```

In the example above, three notes of the `bassLine` are re-used for the clone function and stored in the `leadLine` ring. The lead instrument then uses those values as notes in the octave 1. At the end all the notes are synced to the C-major scale.

7.2 Further sound sculpting

Last but not least, other parameters of the instruments can also be set. When choosing a `synth`, the user has to choose a waveform as a starting point. This can be a `sine`, `triangle`, `saw` or `square`. All these have various parameters that can be modified by calling their function. Besides the already described functions such as `note()`/`pitch()`, `time()`/`timing()` and `beat()`/`rhythm()`, the other functions are `amp()`/`gain()` for amplitude, `envelope()`/`shape()` for setting the envelope, `filter()`/`cutoff()` for applying a lowpass filter and `add_fx()` for adding sound effects such as overdrive, reverb or delay. For example a low sounding bass with distortion, a long envelope, large amount of reverb and some lowpass filtering will be written like this:

```
set_tempo 80

new synth saw name(rumble)
    set rumble note(-4 0) envelope(10 10000) time(2)
    set rumble filter(2 1500 300 0.5)
    set rumble add_fx(drive 4) add_fx(reverb 2 9)
```

8 Conclusion

In this paper the reader is introduced to the live coding environment Mercury. An environment with the focus on quick and hands-on composing, performing and communicating of livecoded music. We see that, by allowing for a higher level of abstraction, the performer does not need to write large amounts of code and the audience can comprehend what is happening and is kept engaged. Complementing the sound with visual objects for every instrument adds value to this understanding. When adding a restriction to the amount of lines and characters on screen the livecoder is forced to delete parts and move on with the composition. By changing the musical notation system from one dimension (pitch) to two dimensions (interval and octave) we can effectively use the same lists for multiple instruments and create coherence in the musicality of the performance. Incorporating generative and transformational list functions assist in the algorithmic composition process and are methods that must be included in a live-coders toolset. By making the environment functional as a sequencer, the performer can immediately make sound after coding the first instrument. Exposing all instruments to global variables for setting musical parameters (tempo, scale and random-seed) greatly improve musicality. Mercury has already been used for many performances at various Livecoding events and proven itself as a performance tool.

8.1 Future work

Mercury is still in its prototyping phase and is a ongoing project. The environment will undergo many changes and additions to the software in terms of programming efficiency, sound design libraries, visual libraries, algorithmic composition technique libraries and more.

First of all more synthesizers will be added for more control of sounddesign while performing. All parameters of these synths should allow to be modified through functions. A feature that allows for modulation of parameters with the audio-signal of other instruments instead of looked-up values from a ring is also on the todo list. Besides that two extra instruments, namely a looper and a poly_synth, will be added to the environment to be able to loop sounds and play chord progressions with one instrument.

Secondly, on the subject of algorithmic composition, more complex techniques will be looked into to the environment. Functions such as Markov-chain, L-systems, Game-of-Life and Attractors will be used as generative and manipulative list functions. Furthermore, to allow for compositional structure on macro level looking into usage of breakpoint envelopes is helpful, as described by Andrew Sorenson. The genera-two version for probability sequencing as demonstrated by Nick Collins in his paper is also an interesting technique to generate variation in macro structure.

In the third place, when polyphony is made possible, the ring will get a multidimensional variation that can be use for chord progressions or fuzzy logic lookup tables for variations in melodies.

Finally and most importantly an idea of genre-based programming as an overall abstracting layer is an idea that will get attention for research and development. In this genre based paradigm, global parameters like tempo and scale, sound design parameters, and many more variables will be changed or interpolated based on provided genre keywords. Allowing the performer to immediately change it's entire sound by only changing the genre while using the same code.

References

Barri, Tarik. 2009. "Versum : Audiovisual Composing in 3d." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 264–65. Pittsburgh, PA, United States.

Blackwell, Alan, and Nick Collins. 2005. "The Programming Language as a Musical Instrument." In *Proceedings of the Psychology of Programming Interest Group*, 120–30.

Blackwell, Alan, and Thomas Green. 2002. "Notational Systems - the Cognitive Dimensions of Notations Framework." In, 103–34. Morgan Kaufmann.

Burland, Karen, and Alex McLean. 2016. "Understanding Live Coding Events." In *International Journal of Performance Arts and Digital Media*, 12 (2):139–51.

Collins, Nick. 2001. "Algorithmic Composition Methods for Breakbeat Science." In *Proceedings of the Music Without Walls Conference*.

———. 2011. "Live Coding Consequence." In *Leonardo*, 44, (3):207–11.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (3): 321–30.

Fischinger, Oskar. 1938. "An Optical Poem." Vimeo; https://vimeo.com/155998018. 1938.

"Live Midi Effect Reference, 23.6 Scale." 2018. In *Ableton Reference Manual Version 10*. Berlin, Germany: Ableton AG.

Magnuson, Phillip. 2008. "Serialism." In *Sound Patterns, a Structural Examination of Tonality, Vocabulary, Texture, Sonorities, and Time Organization in Western Art Music*.

McLean, Alex, and Geraint Wiggins. 2010. "Tidal - Pattern Language for Live Coding of Music." In *Proceedings of the 7th Sound and Music Computing Conference, Stockholm*.

———. 2011. "Texture: Visual Notation for Live Coding of Pattern." In *Proceedings of the International Computer Music Conference*, 621–28. University of Huddersfield, UK.

Nilson, Click. 2007. "Live Coding Practice." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 112–17. New York City, NY, United States.

Sorenson, Andrew, and Andrew R. Brown. 2007. "Aa-Cell in Practice: An Approach to Musical Live Coding." In *Proceedings of the International Computer Music Conference, Copenhagen*, 292–99.

Spiegel, Laurie. 1981. "Manipulations of Musical Patterns." In *Proceedings of the Symposium on Small Computers and the Arts*, 19–22. IEEE Computer Society Catalog No. 393.