Live coding the code: an environment for "meta" live code performance

Andrew Thompson Centre for Digital Music, Queen Mary University
andrew.thompson@qmul.ac.uk

Elizabeth Wilson Centre for Digital Music, Queen Mary University
elizabeth.wilson@qmul.ac.uk

Abstract

Live coding languages operate by constructing and reconstructing a program designed to create sound. These languages often have domain-specific affordances for sequencing changes over time, commonly described as patterns or sequences. Rarely are these affordances completely generic. Instead, live coders work within the constraints of their chosen language, sequencing parameters the language allows with timing that the language allows.

This paper presents a novel live coding environment for the existing language lissajous that allows sequences of text input to be recorded, replayed, and manipulated just like any other musical parameter. Although initially written for the lissajous language, the presented environment is able to interface with other browser-based live coding languages such as Gibber. This paper outlines our motivations behind the development of the presented environment before discussing its creative affordances and technical implementation, concluding with a discussion on a number of evaluation metrics for such an environment and how the work can be extended in the future.

Introduction

Live coding practice, as well as existing as a novel vehicle for the performance of algorithmic music, also acts an extension of musical score as a way of traversing the musical domains proposed by (Babbit 1965) - from the ***graphemic*** to the ***acoustic*** and ***auditory***. Furthermore, the cognitive innards of a live coder are often encouraged to be exposed in live coding practice (Collins 2011) through the projection of patterns they encode displayed for an audience. To further ideas of notation and exposing cognitive processes, the possibilities of revealing performer's notation ***changes*** is presented as a novel way of live coding.

To this end, we present a new browser-based live coding environment for the live coding language ***lissajous***. This environment allows for the sequencing of ***the text buffer itself***, a feature not often available in live coding languages themselves. Before describing the environment's creative affordances, we provide a brief overview of the lissajous language; describing its basic features and limitations.

Lissajous describes itself as "a tool for real time audio performance using JavaScript" (Stetz 2015). In practice, lissajous is a domain specific language embedded in JavaScript. It exposes a number of methods for creating and manipulating musical sequences and these can be conveniently chained together using method chaining (colloquially known as dot chaining).

The single most important of element the lissajous language is the `track`. Tracks are able to synthesise notes or play samples such as:

```
a = new track()
a.tri().beat(4).notes(69,67,60)
```

This creates a new track, sets the waveform to a triangle wave, sets the internal sequencer to tick every four 1/16th notes, and creates a pattern of three MIDI notes that will cycle indefinitely. Similarly, we can do:

```
b = new track()
b.sample(drums)
b.beat(2).sseq(0,2,1,2)
```

Here, "drums" refers to an array of drum samples. We set the sequencer to tick every two 1/16th notes and then create a pattern to choose samples based on their index in the array.

Most parameters of a track can be given a pattern of values instead of just one as is the case for notes and sseq above. These are controlled by a track's internal sequencer, the timing of which is set by the `beat` parameter (which can also accept a pattern of timing values). For a comprehensive reference of the language API, we direct readers to the lissajous language documentation.

Lissajous was chosen as the target language for the environment for two reasons: (1) the language is relatively straightforward and so it is easy to explain and modify, and (2) the sequencing capabilities of the language are restricted to a predetermined collection of parameters, making it an ideal candidate to show how a "meta" environment can be leveraged for more creative control.

Motivation

Many live coding languages can be described as embedded domain-specific languages; that is, they are libraries and functions implemented directly on top of some existing programming language rather than an entirely new language in itself. This can be observed in many popular live coding languages such as TidalCycles (McLean and Wiggins 2010a), FoxDot (Kirkbride

2016), and Gibber (Roberts 2012) which are embedded in Haskell, Python, and JavaScript respectively. This benefits both live coding language developers and live coding performers. Developers can piggy-back on an existing language's semantics and tooling, allowing them to focus exclusively on the domain-specific nature of the embedded language. Similarly, performers can exploit existing knowledge of the host language and perhaps even use third-party libraries for that host language.

As noted in a more general review of music programming languages, music programming greatly benefits from the power afforded by a general-purpose programming language (Dannenberg 2018). In the case of live coding specifically, this creates a disconnect between what the live coder is able to do when exploiting the general-purpose host language and what is possible with the embedded domain-specific language. Consider lissajous, a language embedded in JavaScript. We saw how to sequence a pattern of notes to cycle through at quarter note intervals:

```
a.tri().beat(4).notes(69,67,60)
```

Because lissajous is embedded in JavaScript we have access to everything a normal JavaScript program would; such as changing the window background colour:

```
document.body.style.backgroundColor = "red"
```

Now suppose we wish to cycle through the colours red, green, and blue in time with the note sequence. There is suddenly a disconnect. The musical timing available to our lissajous track is not available for any arbitrary code. To attempt to remedy this situation, some languages allow callback functions to be inserted into sequences instead of primitive values.

```
a.tri.beat(4).notes(function () {
  document.body.style...
  return ...
})
```

This can quickly becomes a mess. Live coding languages often focus on brevity and shorthand to allow complex ideas to be described concisely (McLean and Wiggins 2010b). This effort is largely wasted in situations like the one presented above. The live coder must now manage state that was originally managed by the embedded language itself such as which note to return from the function and which colour to set the background.

At present live coders have a handful of choices to overcome this issue: (1) simply don't attempt things that are not easily afforded by the embedded language, (2) seek a new language that potentially does offer the feature needed as part of the embedded language, or (3) modify the embedded language in some way to include the feature you need. We present a meta environment as a fourth solution that allows the text input itself to be sequenced, allowing for completely arbitrary code actions to be performed in time.

Meta Livecoding

In lissajous, and indeed many other live coding languages, parameters of a track are sequenced with patterns. Multiple parameters can be sequenced by the same pattern, and multiple tracks can be synchronised in time. The question arises of what to do when we wish to sequence actions ***not*** supported by the host language's sequencing capabilities. Such a result can be achieved in Gibber, for example, by encapsulating arbitrary code in an anonymous function and supplying that function as a callback to a Sequencer constructor. In lissajous, a track's prototype can be modified to add extra parameters controllable through patterns.

Attempting to modify the prototype of an object is both expensive to compute and awkward to do live, making this a far from optimal solution when performing. Gibber's sequencer is more flexible in this regard, but the limited nature of only being able to supply ***one*** callback function encourages using this feature for minor modifications rather than grand macro changes to a performance.

A fundamental problem is that while live coders are able to express things in an arbitrary manner, our code must follow the rules of the system it lives in. In other words the live-coder can assume the role of the sequencer, changing and adding various pieces of code at fixed time intervals, but the sequencer is bound by the rules of the host language and cannot assume the role of a live-coder.

The idea of metaprogramming is not entirely novel. Lisp, for example, has a famously powerful macro system and a number of live coding Lisp dialects take advantage of this fact, including Extempore (Sorensen and Gardner 2010) and Overtone (Aaron and Blackwell 2013). What makes Lisp's macro system so powerful is twofold. First, macros are capable of returning arbitrary Lisp expressions to be compiled and evaluated at ***run time***. Second is the fact that Lisp macros are written in Lisp itself, in contrast to macro features available in other languages such as C. Importantly, Lisp macros can themselves be manipulated and created at run time by other Lisp functions.

The Siren environment for TidalCycles provides a graphical environment for so-called hierarchical composition (Toka et al. 2018). Snippets of Tidal code are laid out in a grid interface representing a scene, with columns representing individual tracks. Scenes can be switched and tracks modified in real time, allowing for larger structural changes to be made to a performance that are otherwise awkward to achieve in Tidal.

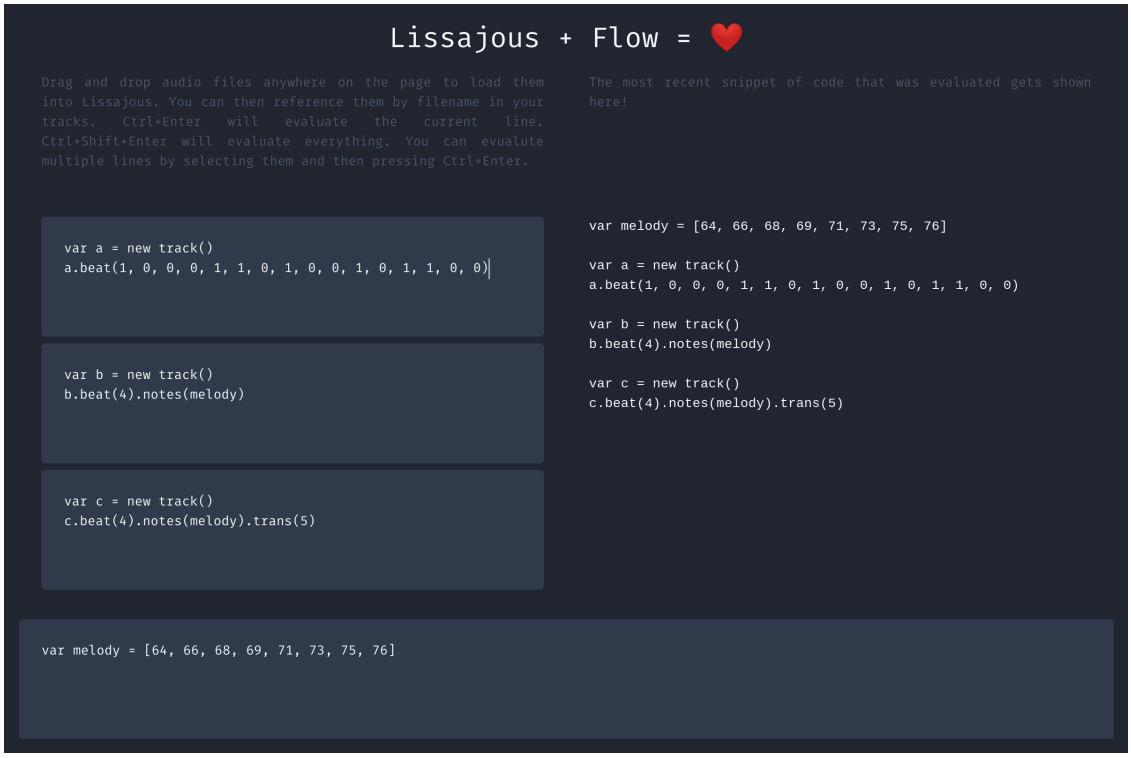

Figure 1: The Flow-Lissajous environment. (Left) The code editor. Here, as with many other live coding environments, lines of code are entered and evaluated. (Right) The last snippet of code that was evaluated. This section of the environment cannot be edited, instead always showing the most recent piece of code that was evaluated. This provides a visualisation of what the program is doing that is especially useful when sequences of source code are being played back and manipulated.

With this in mind, we have developed a metaprogramming live coding environment that allows for sequences of code expressions to be manipulated in the same manner that note sequences can. While Lisp macros are capable of producing Lisp expressions, the environment, known as Flow-Lissajous, is built around manipulating the source code directly. In this way, it functions similarly to Siren. Where Flow-Lissajous diverges, however, is in its use of program structures that already exist in the lissajous language rather than a graphical interface.

A global `meta` object is exposed in the environment that allows for the contents of a particular text buffer to be recorded and played back. While recording, every time the source code is evaluated that source code is saved into an array for playback. During playback, this sequence of source code is fed into lissajous for evaluation. Unlike Lisp macros, however, this process is not opaque. As the source code itself is being modified, these modifications are reflected in real time for both the performer and audience to see. Importantly, the `meta` object is a slightly modified lissajous track. This means it is possible to take advantage of lissajous' existing pattern sequencing and manipulation abilities

This opens the possibility of a number of creative performance techniques not easily achieved in other environments, or indeed the host language's themselves. Allowing the live

coder control over both the implicit parameters of each variable and overall structure of encoded music provides an interesting use-case for live coding research. Long-term structure is not often addressed in the literature, due to the demands of immediacy and real-time musical feedback. Introduction of ideas of the “meta” provides a context with which the coder can explore both musical parameters and structural changes through symbolic expression.

It also adds a new dimension to live coding improvisation akin to using looping pedals or recording audio clips into a DAW. Typically, an improvised live code performance involves the constant re-evaluation and tweaking of code snippets with the end result being some musical idea that represents the ***sum*** of the code snippets evaluated before it. Often the journey is as important as the destination, however, and after tweaking 50 parameters it is difficult to remember how we started or how we got here. To that end, the environment embraces the live code mantra of "repetition" by allowing performers to capture and repeat sequences of a performance without the necessary premeditation of setting up the appropriate tracks and sequencers.

As mentioned, this lends itself to a kind of improvisation seen in live looping performances seen, for example, within guitar music practices. Although both live looping guitar performances and live coding performances are typically built up by layering loops of musical ideas, there's a degree of permanence found in the guitar performance that is ***not*** found in live coding performance. If we are unhappy with how a sequence sounds we can tweak any number of parameters; we can change the notes, add effects, alter the timing. While this is undoubtedly powerful, it is perhaps somewhat antithetical to the TOPLAP manifesto's preference of "no backups":

> *No backup (minidisc, DVD, safety net computer)*

Instead, much like the guitarist, live coders must now commit to any particular sequence of events or start over entirely.

Additionally, this enables a new method of performance distribution beyond distributing an audio recording or a complete source code file. It is not uncommon for live code musicians to distribute code files alongside music releases for other musicians to experiment with. This is largely unfeasible in traditional music distribution as distributing multi-track recordings known as stems comes at the expensive of both file size and production time. The distribution of source code removes a certain amount of authorial intent however, presenting a collection of ideas curated by the musician and giving the consumer free reign over their ultimate arrangement and use.

The environment presented here allows for a different means of source code distribution. Instead of a complete source code file, perhaps annotated with comments suggesting which snippets to evaluate in what order, musicians can distribute a sequence of code expressions to be evaluated by the environment. Given that the environment is designed to operate in real

time, listeners are free to interact with the code at any time to make alterations or add new elements while still affording the original musician macro control over the performance. Transitions to new sections, or new compositions entirely can be orchestrated while still allowing the listener some amount of creative freedom; encouraging a stronger dialog between the two parties that is not afforded by full source code distribution.

Implementation

In the previous section we described a global `meta` object as a slightly modified lissajous track. Here we will describe in more detail the technical implementation of the environment and how it operates. Lissajous was initially designed to be used exclusively in the browser's console; heavily relying on global variables and objects. Most obviously, Flow-Lissajous provides a more graphical interface for inputting and displaying code.

The Flow JavaScript framework was chosen to construct this interface because of it's implementation of the Model-View-Update (MVU) application architecture. Central to the MVU architecture is the notion of actions; messages passed to an update function that are used to construct a new model of application state. This creates a deterministic, stepped sequence of application states for any fixed sequence of actions (Thompson and Fazekas 2019). Such modelling of application state maps cleanly to the core concept of sequencing source code changes in live coding performance.

Actions are objects with a unique string tagging the action, and an optional payload. For Flow-Lissajous, there are three important actions: EVAL, REC_START, and REC_STOP. The EVAL action contains a string payload of the snippet of code to pass to lissajous for evaluation. These actions are not unlike the actions found in the collaborative live coding environment CodeBank (Kirkbride 2019). Figure 2 diagrams how the Flow framework and lissajous communicate with one another.

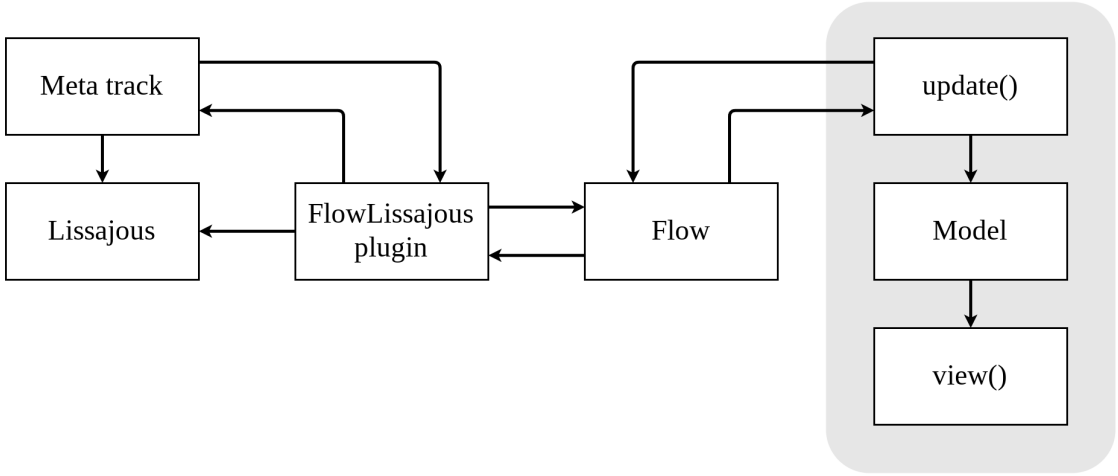

Figure 2: The overall architecture of the FlowLissajous environment. The grey shaded area represents the side-effect free portion of a Flow program. This is important to guarantee the same sequence of actions produces the same sequence of rendered views. Upon receiving

an action, Flow calls the user-defined update function to create a new model of application state and optionally a side effect for the runtime to perform. These side effects take the form of code snippets that are passed to lissajous to evaluate. Lissajous then sends actions back to the Flow runtime when stepping through a sequence of code snippets.

The Flow Lissajous plugin provides a thin wrapper around lissajous, initialising the `meta` object when the Flow program is initialised and provides Flow a means to call `eval` without polluting the Flow program with side effects. While recording, the Flow program keeps track of every payload evaluated by the EVAL action. Once recording stops, the array is passed to the FlowLissajous plugin where each snippet is transformed into it's own EVAL action. When a sequencing method such as `beat` or `beat32` is called, lissajous then dispatches an EVAL action with the current code snippet in the sequence. This is necessary so the Flow program can update it's view with the most recently evaluated code snippet.

A minor addition to the lissajous source code was made to make it possible to interface with Flow in the way that we required. This took the form of the creation of an additional property on the lissajous track prototype:

```
self._flowSequencer = new Sequencer(function() {});
self._attachSequencers(self._flowSequencer);
```

and an additional function that enables a sequence of code snippets to be stored on the track:

```
track.prototype.flow = function() {
  var self = this;
  var arguments_ = _parseArguments(arguments);
  return function ($dispatch)  {
    self._flowSequencer.set(arguments_.map(snippet => () =>
$dispatch(snippet)));
    self._setState('snippets', arguments)
  }
};
```

It is important to note there is a clear boundary between Flow and lissajous. These modifications do not require lissajous to have knowledge of Flow specifically, and so alternative frameworks such as React or Vue are equally as applicable for the view layer of the application. As we have described, the host language - in this case lissajous - required minimal modification to work inside the environment. This boundary between editor and language is not unlike the boundary between language and synthesis engine found in SuperCollider (McCartney 2002). In SuperCollider, this is powerful because any arbitrary

language can communicate with SCServer as long as it knows the necessary OSC messages to send. Likewise, conceivably any arbitrary language can exist inside the meta environment as long as it implements the necessary Flow plugin to receive the text buffer and dispatch actions.

Future Work

The Flow-Lissajous environment presents a novel practice of live coding by manipulating the code itself. In doing so, it affords a two new interactions otherwise not available in existing live coding environments: (1) the ability to act as a “code conductor” for a more macro level of control over a performance, and (2) the possibility to distribute a performance as text sequences rather than audio or a single text file.

The environment is very much presented as a work-in-progress however, and there are a number of research and development avenues to pursue in the future. Most pressing is the addition of the ability to have multiple `meta` objects and editors for each lissajous track created during the performance. This would bring the environment closer in line to the Siren environment and allow for much more interesting performance opportunities. Thanks to the largely compartmentalised design of the environment, such a feature can be easily implemented in the future.

The creative implications of such a system have been described here but could be assessed using formal evaluation metrics. Within the NIME community particularly, development of new interfaces and instruments is becoming increasingly coupled with their evaluation through existing HCI research. Systematic and rigorous evaluation allows critical reflection on both the musical outputs that can be produced by an interface and the ways and means by which the tool is used. (Wanderley and Orio 2002) posit a useful framework for experimental HCI techniques applied to musical tasks, whereas (Stowell et al. 2008) advocates for structured qualitative methods like discourse analysis for evaluation of live human-computer music-making.

Beyond the added dimension to live performance, the presented environment opens up interesting opportunities for the research community and the study of live code performance as a whole. Smith et al. present an analysis of multiple live coding performances by two separate artists through a processing of coding the screen capture of each performance (Swift et al.2014). In this context coding refers to labelling specific events along the performance’s timeline, noting particular actions such as code insertion or changing instrument pitch. The authors note “future work could extend our study through the instrumentation of the programming interface to enable automatic data collection” and the presented environment enables precisely this sort of automatic data collection.The presented environment enables this kind of automatic data collection, and could easily be expanded to include useful additional information such as a times-tamp for the event. While it will still be required for researchers to choose when to code a musical event,coding of textual events

now becomes trivial; simple analysis between events can determine whether the current event is an insertion, deletion, or “quick edit” for example.

Acknowledgments

This work was supported by EPSRC under the EP/L01632X/1 (Centre for Doctoral Training in Media and Arts Technology) grant.

References

Aaron, S., and Blackwell, A. F. (2013) From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages, in Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modelling & Design, available: https://doi.org/10.1145/2505341.2505346

Babbit, M. (1965) The use of Computers in Musicological Research, in Perspectives of New Music 3(2), available: https://doi.org/10.2307/832505

Collins, N. 2011. Live Coding of Consequence, in Leonardo 44(3), available: https://doi.org/10.1162/LEON_a_00164

Dannenberg, R. B. (2018) Languages for Computer Music, in Frontiers in Digital Humanities, available: https://doi.org/10.3389/fdigh.2018.00026

Kirkbride, R. (2016) FoxDot: Live Coding with Python and SuperCollider, in Proceedings of the International Conference on Live Interfaces

Kirkbride, R. (2019) CodeBank: Exploring public and private working environments in collaborative live coding performance, in Proceedings of the 4th International Conference on Live Coding

McCartney, J. (2002) Rethinking the Computer Music Language: SuperCollider, in Computer Music Journal 26(4), available: https://doi.org/10.1162/014892602320991383

McLean, A. and Wiggins, G. A. (2010a). Tidal–pattern language for the live coding of music. In Proceedings of the 7th sound and music computing conference.

McLean, A. and Wiggins, G. A. (2010b) Petrol: Reactive Pattern Language for Improvised Music, in Proceedings of the 2010 International Computer Music Conference, available: http://hdl.handle.net/2027/spo.bbp2372.2010.066

Roberts, C. and Kuchera-Morin, JA. (2012) Gibber: Live Coding Audio in the Browser. in Proceedings of the 2012 International Computing Music Conference, available: http://hdl.handle.net/2027/spo.bbp2372.2012.011

Sorensen, A. and Gardner H. (2010) Programming with time: cyber-physical programming with impromptu, in Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications, available: https://doi.org/10.1145/1869459.1869526

Stetz, K. (2015) Lissajous: Performing Music with JavaScript, in Proceedings of the 1st International Web Audio Conference, available: https://medias.ircam.fr/x6b4e2d

Stowell, D., Plumbley, M. D., and Bryan-Kinns, N. (2008) Discourse Analysis Evaluation Method for Expressive Musical Interfaces, in Proceedings of the 8th International Conference on New Interfaces for Musical Expression, available: http://www.nime.org/proceedings/2008/nime2008_081.pdf

Swift, B., Sorensen, A., Martin, M., and Gardner, H. (2014) Coding Livecoding, in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, available: https://doi.org/10.1145/2556288.2557049

Toka, M., Ince, C. and Baytas, M.A. (2018) Siren: Interface for Pattern Language, in Proceedings of the 18th International Conference on New Interfaces for Musical Expression, available: http://www.nime.org/proceedings/2018/nime2018_paper0014.pdf

Thompson, A. and Fazekas, G. (2019) A Model-View-Update Framework for Interactive Web Audio Applications, in Proceedings of the 14th International Audio Mostly Conference, available: https://doi.org/10.1145/3356590.3356623

Wanderley, M. M., and Orio, N. (2002) Evaluation of Input Devices for Musical Expression: Borrowing Tools from HCI, in Computer Music Journal 26(3), available: https://www.jstor.org/stable/3681979