ICLC 2020
Proceedings of the 2020 International Conference on Live Coding

5th - 7th 2020
University of Limerick - Ireland

Contents

ICLC2020

International Conference on Live Coding

WED 5th - FRI 7th 2020

University of Limerick - Ireland

Local Committee

Dr. Giuseppe Torre [Conference & Paper Chair]
Digital Media and Arts Research Centre
Dept. of Computer Science & Information Systems
University of Limerick
giuseppe.torre@ul.ie

Dr. Neil O'Connor [Performance Chair]
Digital Media and Arts Research Centre
Dept. of Computer Science & Information Systems
University of Limerick
neil.oconnor@ul.ie

Dr. Nicholas Ward [Workshop Chair]
Digital Media and Arts Research Centre
Dept. of Computer Science & Information Systems
University of Limerick
nicholas.ward@ul.ie

Dr. Nora O'Murchu [Installations Chair]
Digital Media and Arts Research Centre
Interaction Design Centre
Dept. of Computer Science & Information Systems
University of Limerick
nora.omurchu@ul.ie

International Steering Committee

Chair Forewords

I hope you do not mind if I use this small page as a personal memoir.

You are about to read the work of a relatively small yet fast-growing international community of talented technologists and amazing performers. ICLC 2020 is only the fifth edition of the International Conference on Live Coding. It was, without the slightest doubt, an immense pleasure to host it at the University of Limerick.

I recall three days of intense panel discussions, workshops and amazing performances (some literally jaw-dropping!). More importantly I recall a relaxed and friendly atmosphere where everybody was genuinely interested in the works of peers and eager to share and learn.

I also recall that just two weeks after the conference, Europe (as well as many other countries) went into lockdown for COVID-19. Then we began to appreciate the meaning of being together. The importance of a "we" before an "I", to borrow from Jean-Luc Nancy. A "we" unmediated by digital technology where the roots of digital are to be found and where all discourse on digital eventually returns - and that applies to live coding too.

Go n-éirí an t-ádh leat

Giuseppe Torre

Additional Material

Emma Cocker's keynote address can be found here (recommended!): `https://www.youtube.com/watch?v=oCuOhOZ3bw8`.

Recordings of all papers and performances can be found here: `https://www.youtube.com/channel/UCN-9RKW_izkIUMH0eQ60H2g`

ICLC Conferences pages `https://iclc.toplap.org/`

Live Coding Repository `https://toplap.org/`

Thanks!

My most sincere thanks go to all the attendees of ICLC2020 for having generously shared knowledge and their artistic practices.

All the students volunteering: Matthew Reynold, Ocean McCormack, Walt Neid, Ciara O'Brian, Emer O'Reilly, Rian Stephen,

Lastly, and without any doubt not least(!), the most sincere thanks to the audiovisual technical team: Roisin Berg, Lianne Daly, Tony Irwin, Alan Dormer, Dave O'Brien, Colm McGettrick.

PAPERS

The Live Loom

Alex McLean
Deutsches Museum
alex@slab.org

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

This paper introduces the Live Loom, a warp-weighted hand-loom augmented with computer control, using a visual live coding environment. The traditional weaving technique of colour-and-weave interference patterns are explored, revealing the digital, computational nature of weaving that predates the invention of discrete mathematics as it is commonly understood. Early results of live coding such patterns, in the process of learning how to weave, is shared.

Introduction

Live coding is the exploratory practice of changing code while it runs. A live coder uses a programming language as a live interface to a running process, and therefore its inputs and outputs. This allows the programmer to participate in the performing arts, for example to create live coded music, video or dance. Live coding may be used to refer to any use of code in any kind of live situation, but for the purposes of this paper, we will focus on live coding as live improvisation, where code is written without a fixed aim, often from a notional 'blank slate'.

Weaving is a textile craft, where parallel threads known as the warp are held under tension, allowing a second set of threads known as the weft to be passed over and under the warp threads to create a textile fabric. Different textile techniques produce different results; for example in terms of structure, weaving and knitting have very little in common with each other.

This paper investigates how live coding and weaving can be brought together, looking for ways to ground the contemporary practice of live coding in ancient craft. In the following I take a loosely auto-ethnographic approach, introducing the Live Loom, and its use both for understanding weaving and reflecting on the shared history of live coding and textile practice that it might reveal.

Background

Connections between textile craft and computation have been very well explored in the digital arts and beyond. However, as David McCallum notes, such work often does not engage with structure or long history of textiles, but instead treats the three dimensional structure of weave as a simple grid or raster (McCallum, 2018, 0.3.1). McCallum's own work explores the notion of 'glitch' in media art and how it transfers to textile structures, but given that the latter has developed over a far longer period than the former, it is not surprising that along the way he finds much that media art can learn from textiles. In the weaving industry, the technology of computer-controlled looms is of course also well developed. In the present work we are interested in analogies with live coding and 'hands on' computer control. Such interaction is not afforded by most machine looms, where the human weaver is replaced by a machine. However, hand-operated computer controlled looms do exist, and the present work is much inspired by experiments I conducted on the TC1 loom in Textiles Zentrum Haslach, Austria (McLean and Harlizius-Klück, 2018).

Stitching Worlds is a recent, far reaching project imagining a world where textiles more overtly formed the basis of contemporary electronic technology (Kurbak, 2018). Works produced from this project include the "Embroidered Computer" with Irene Posch and collaborators, a working 8-bit computer embroidered in gold. This work integrates textile electro-mechanical relays into a fabric, demonstrating that a feminist approach to linking textiles with computing goes far beyond metaphor - textiles can compute.

Feminist alternative history

Although not the primary focus of the present work, comparing the contemporary practice of live coding with the ancient craft of handweaving has potential to support and extend a somewhat obscured feminist history of computation. Feminist perspectives on computing and weaving are hardly new, a well-known reference point being Sadie Plant's influential text "Zeros and Ones: Digital Women and the New Technoculture" (Plant, 1998). However, the once dominant role of women in computer programming has been steadily erased since the 1960s and '70s (Hicks et al., 2017), and despite recent efforts, gender diversity in software engineering is an ongoing problem.

As a relatively new interdisciplinary practice that tries to reject hierarchies[1], live coding offers an opportunity to build a gender diverse culture, and this opportunity is a core topic across live coding research and practice (Armitage, 2018). Turkle and Papert related gender to the plurality of relationships between coder and program observed in children, describing a more conversational approach to coding, with mid-course corrections rather than fixed-goals as bricolage (Turkle and Papert, 1990, p. 136). This approach is certainly evocative of live coding, with the suggestion being that it is one likely to be favoured by girls, but discouraged by instructors in favour of more fixed design processes.

Armitage (2018) brings together female perspectives on live coding in the Algorave scene, relating one interviewee's experience of live coding "... as a way of working through their daily life, adding structures to it and providing functions for being. These lived patterns merge with their daydreams and expressions of colour and geometry to form her live coded visuals." (Armitage, 2018, p. 39). This again evokes Turkle and Papert's bricoleur, and indeed the ancient social and intellectual function of weaving in building a personal cosmos (Harlizius-Klück and Fanfani, 2017).

Setting aside the Jacquard machine

The Jacquard machine is a well known device for individual control of threads in the weaving process, classically through the use of punch cards. From across computer science and popular culture, the Jacquard machine is often invoked as part of an 'origin story' of

[1] See for example the Algorave Guidelines - `https://github.com/Algorave/guidelines`

computation, following Charles Babbage's mention of it as an influence. However, the Jacquard machine does not do computation, it is merely a mechanism for accepting input data. The Jacquard machine therefore brings a fundamental misunderstanding to the topic of weaving and computation which is very difficult to work around.

Yes, weaving is computational, and yes, the Jacquard machine allowed data to be fed into that computation. But the same computational nature is present in all weaving, including traditions of hand weaving developed over millennia (Harlizius-Klück, 2017). The computation was already there before Jacquard, and by helping automate the weaving process, his device only takes humans further away from that computation. So while Jacquard's machine is often described in terms of the beginning of the relationship between weaving and computing, the opposite is true - it was an end.

So let's try to wipe the Jacquard machine from our minds in the following discussion, not because the technology isn't interesting and useful, but because the discussion around it is so full of misunderstanding. Once we do that, we are able to see that such machinery did not introduce any computation to weaving - the computation was there already. As will become evident in this paper, the computation is not in the machine, but in the weave.

Introducing the Live Loom

Having set one mechanism aside, I introduce another. The Live Loom is a warp-weighted loom, with solenoids attached so that warp threads may be individually picked from software. First, I explain the technology of the warp-weighted loom, and later explain the electro-mechanical attachments on the Live Loom.

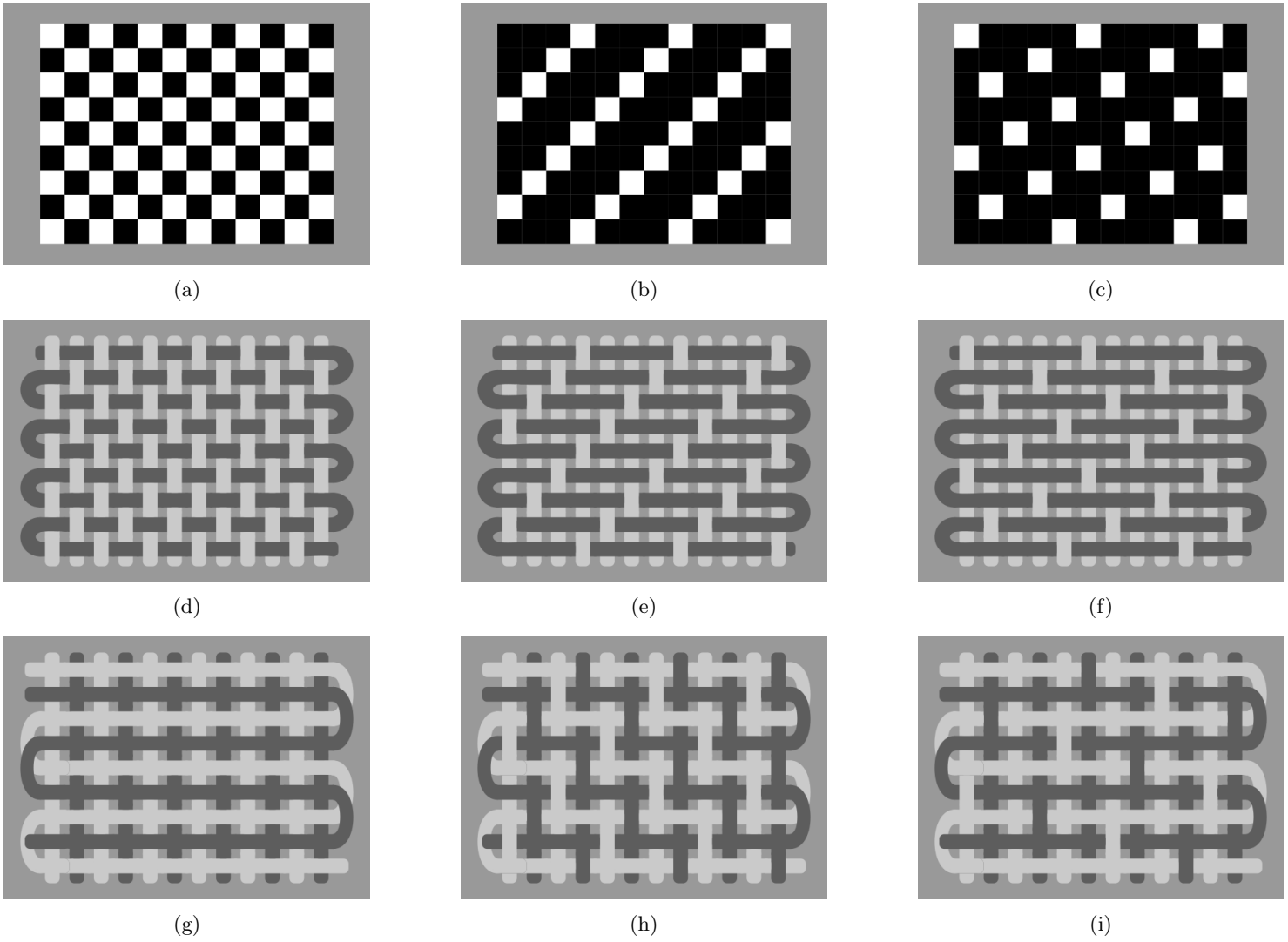
The primary purpose of any loom is to hold a group of parallel threads, the warp, in parallel and under tension, allowing weft threads to be woven over and under the warp threads. The warp-weighted loom is an ancient technology, where tension comes from the effects of gravity, by attaching weights to the bottom of the warp threads. By contrast, on modern looms warp threads are generally horizontal, and held in tension through mechanical means. The essential components of a warp-weighted loom are therefore very simple, consisting of a frame holding two horizontal bars in place, one to hang the warp from, and another below separating alternate threads, keeping them in order, and creating a potential gap (using weaving terminology, the natural shed) for the weft to pass through by default. The simplicity of the loom is also its advantage - the simpler the loom, the fewer constraints and therefore more possibilities there are to weave complex structures on it.

The weaving process involves a weft thread going over and under the warp threads, following one of a very large range of possible patterns, for example creating tabby, twill or satin structures (Emery, 2009, see also Fig. 1). Selective warp threads are pulled forward, creating a new gap or shed between the pulled and non-pulled warp threads, through which the weft travels in a straight line. When the warp threads are returned, the weft is trapped inside, and the next shed is prepared.

Weaving technology

The Live Loom is shown in Figure 2. Although it carries a contemporary 'maker' aesthetic due to its laser cut plywood construction, at its core, it is a hand-loom following an ancient warp-weighted loom design. The additional electro-mechanical parts do not replace the core functions of the loom, but rather augment them in order to allow threads to be selected using a computer language as well as directly by hand. The hardware and software designs are available as open hardware/free software (McLean, 2019).

The Live Loom is fitted with a number of solenoids (currently sixteen), mounted on two axes to double the number that could otherwise fit in a given space. The solenoids are controlled by an arduino micro-controller, via a bank of relays. When activated, each solenoid will push against a stick, which pulls its corresponding warp thread forward via a string. In this paper we refer to the wooden stick and string collectively as the heddle. With each solenoid controlling one



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 1: Fundamental weave structures, shown with binary 'draft' structure (top), simulated weave with light warp and dark weft (middle), and simulated weave with alternating light and dark warp and weft (bottom). These different structures lead to different physical properties and therefore uses (Emery, 2009).

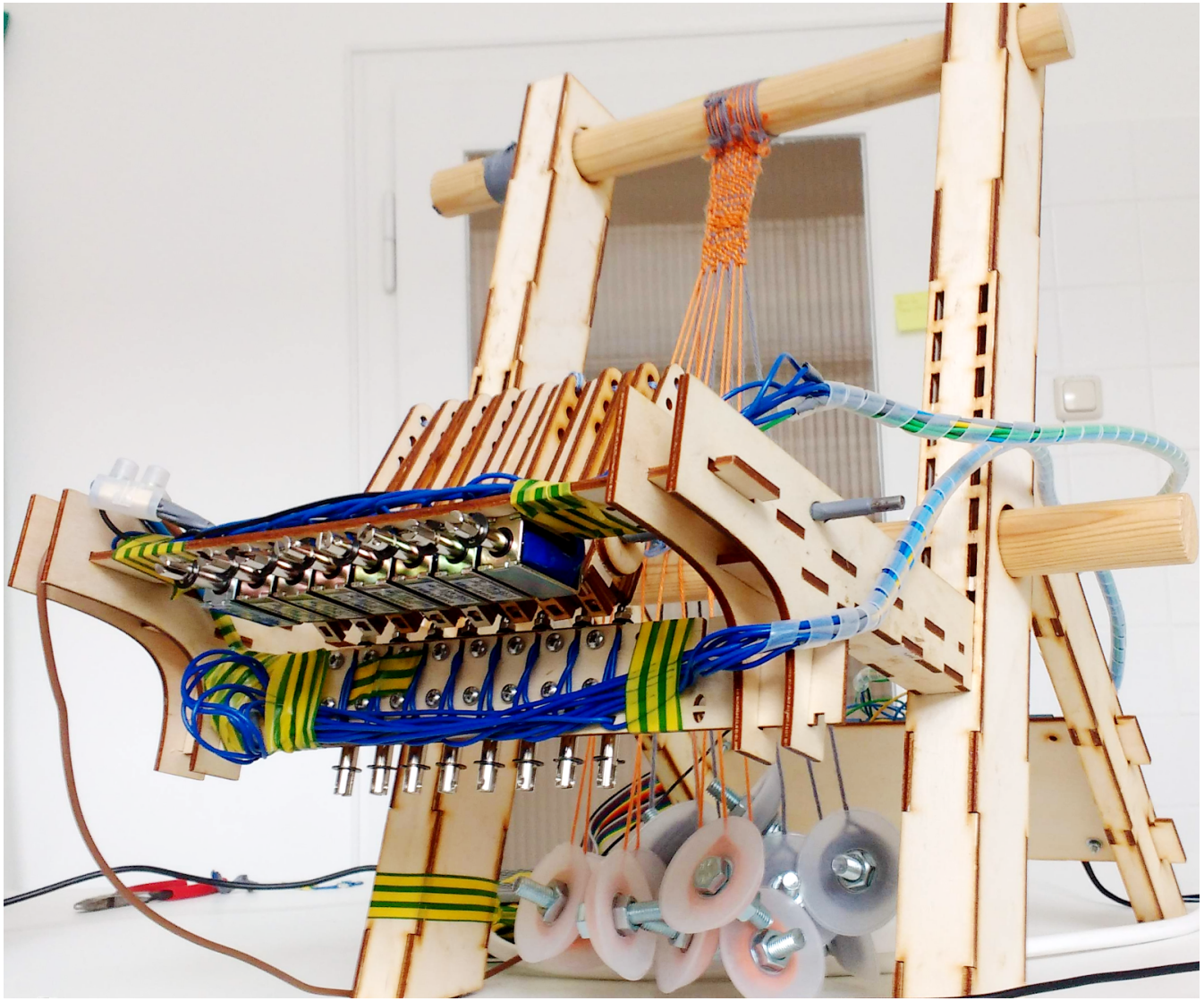


Figure 2: The Live Loom, a warp-weighted loom, with live-codeable heddles via solenoid actuators.

warp thread, the resulting weave is currently constrained to sixteen threads across.

Crucially, these solenoid movements are not designed to fully create a shed. Instead this movement only 'offers up' warp threads to the human weaver-coder, who then pulls the threads further by hand. This seems like a deficiency of power and leverage, but is not; this 'offering up' means the weaver can choose whether or not to pull each thread. This is particularly useful at the edges of the weave, where adjustments are often required to produce a good fabric. The suggestive nature of instructions sent via the solenoids reminds of the live coding choreographic work of Sicchio (2014). We can think of this process as not directly live coding a textile weave, but instead suggesting bodily movements to produce the weave. When live coding people rather than computers, it is humane to respect their ability to exercise creativity and agency in the way they interpret instructions given to them.

Computing a weave

Before introducing a language for live coding the loom, lets first look closer at the computational nature of weaving itself, focussing on colour and weave effects. Such effects bring together different dimensions or systems. Firstly, the structure of the weave - the arrangement of ups and downs in the grid created by the meeting points of warp and weft. Secondly, the colour patterning of warp and weft threads. The visible colour at a particular point of the weave then depends on two things - whether the warp or weft thread is visible (i.e. whether the weft is under or over the warp) and what colour that thread is. The result is an interference pattern between these two systems, creating a deterministic, logical outcome that is nonetheless very difficult for the layperson to predict.

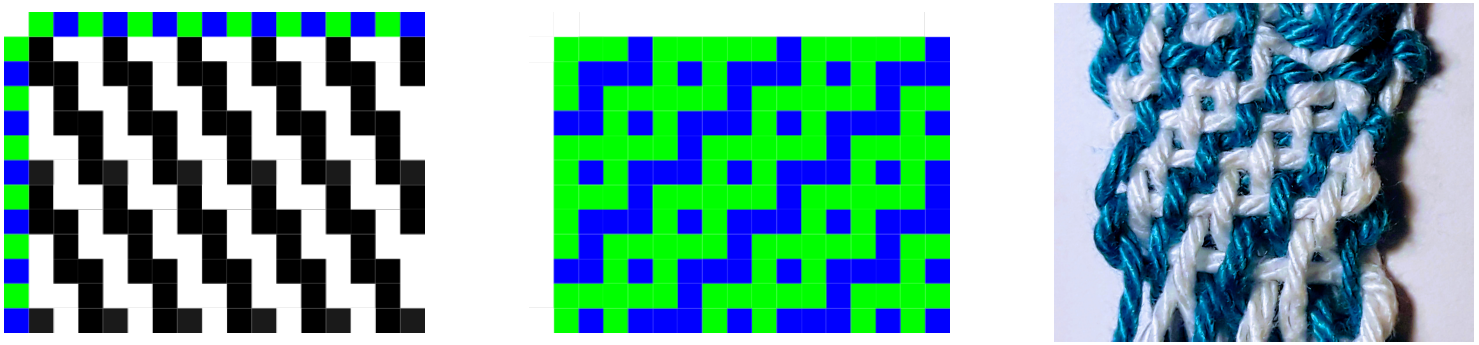
As a simple example of this, consider the weave structure shown in Fig. 3a, known as a draft pattern. The black-and-white grid shows the pattern of weft ups and downs represented as white and black squares respectively. For example, the first row shows a weft thread going under one warp, over two warps, and repeat. The second row shows a weft thread going under two warps, over one warp, and then repeating.

There are also coloured squares at the top and bottom, showing the pattern of warp and weft thread colours respectively, in this case both alternating between light green and dark blue. In order to find what colour will be visible, we look at the weave structure. For black squares, we know the warp colour is shown, so follow the column up to find its colour, otherwise we follow the row to the left[2]. From this we can see that where warp and weft meet with a matching colour (in this case, every other cell in a checkerboard pattern), the visible colour cannot be changed by the structure. This is analogous to the Moire effect seen by placing one net over another, with the visible result being the interference of the upper and lower structures.

If we plot out the result of this interference between thread colour and weave structure, we arrive at the image shown in Fig. 3b. This result will be surprising to a layperson, not only is the vertical and horizontal stripe of warp and weft not visible, but the diagonal runs in a different direction to the underlying weave structure. This experience will be familiar to those who have explored algorithmic interference patterns in livecoding software such as TidalCycles or Hydra, simple inputs often create unexpected, more complex results.

Finally, Fig. 3c shows a fabric woven using this structure and alternating white and blue threads, created by a workshop visitor on the Live Loom. The same features hold in the weave itself, although are not too well defined, due to interaction between the threads, and variation in density. The left and right edges are a mess, because in practice such a structure simply cannot be woven at the edges. Weft threads generally travel from left to right for one row, and from right to left on the next. Therefore, if a weft ends a row over a warp, and begins the next row also over a warp, then it will not be woven at that

[2]In practice, there are other variables which change which colour thread is visible, for example if weft threads are tightly packed, warp threads are hidden completely.



(a) Draft Pattern (b) Result (c) Actual result

Figure 3: A draft pattern and the result of virtually and actually weaving it.

point. A more experienced weaver would make consistent changes at the edges (known as the selvage), such as adding plain weave border, to ensure a coherent result. The above description of colour and weave effects should give us pause for thought. Weaving predates computer programming and indeed discrete mathematics in general, but nonetheless is a discrete, logical and therefore digital computational system. Furthermore, any hand-loom affords exploration of this system. When considering the computational nature of weaving then, we must be careful not to be dazzled by the machinery or electronics of industrial and contemporary weaving technology, when it is the ancient technology of the threads themselves that provides the environment for computation.

Coding the draft

We have already seen that weaving drafts are a form of code, which can compute unexpected results when interpreted. Such weaving drafts are themselves binary, digital images, developed well before electronic digital computers. It is therefore straightforward to add a further level of abstraction by using a programming language to create a draft. The purpose of doing so is to create patterns from patterns, making a rich space to explore weaving and gain tacit knowledge both about how it works, and its relation to computation as it is more conventionally understood in the context of programming languages. Each such layer of abstraction takes us further away from the material, but just as live coding of music brings together the experience of coding and listening, the Live Loom brings together coding with seeing and touching.

Figure 4a shows the current version of the Live Loom coding interface. The code is shown on the left, using the visual live coding interface Texture (McLean and Wiggins, 2011), originally designed as an exploratory interface for the TidalCycles environment, but here re-purposed for a system designed for discrete, binary draft patterns. The set of available symbols and keywords are on the top right, which may be dragged into the code using a mouse. On the bottom right a window into the draft pattern is shown, with the row most recently being sent to the Live Loom marked with blue squares on either side. Finally the row number is written below, which in this case is higher

than the number of rows shown, as the previous rows have scrolled off the top. The weaver-coder can manipulate the code with a mouse, while using arrow keys on a keyboard to step forwards (or backwards) through the draft, sending each warp lift to the loom to be actuated by the solenoids and woven by the weaver.

Perhaps most notable is what is not shown in the software interface. In particular, simulation of thread colour (such as that shown in Fig 3) could easily be included, but is not, indeed thread colour is not dealt with at all in the software, only on the loom. Keeping colour on the loom takes focus away from any simulation on screen and places it in the 'ground truth' of the material. After all, colour is only one quality of thread, alongside thickness, material, ply, tightness and direction of twist, tension and density of warp and weft, and so on. Trying to simulate all of these continuous variables on-screen would be an insurmountable task, and focussing the software on the singular task of planning the discrete structure of ups and downs works very well.

It is important to recognise that although the binary grid of a weaving draft is contained in two dimensions, the structure it describes is very much three dimensional. Indeed certain two-dimensional patterns will result in more than one fabric, one on top of the other, creating the possibilities of double weave structures.3

Live Loom language

The language currently used by the Live Loom is the pure functional programming language Haskell, using its list datatype. Standard Haskell lists are 'lazily evaluated', which means that infinitely long lists can be represented and calculated on demand. A weave structure is simply represented by a one dimensional list of Boolean values, where true and false stands for up and down (or if you prefer, over and under) respectively.

Listed in Table 1.1 below, the current number of functions for composing draft patterns on the Live Loom is small but already provides a very rich space of possibility. The weaver-coder begins with a list of ups and downs, then applies functions to transform that list and/or combine it with other lists. The result is a language interface that produces surprisingly complex results from simple elements.

name	**description**
[]	An empty list
:	Adds a value to a list
up / down	Keywords representing the boolean values of up (over) and down (under)
cycle	Repeats a list forever
backforth	Reverses every other row
offset n	Offsets each row from the last, by the given number of threads
shift	Shifts each row by one thread
rev	Reverses each rows
every n f	Selectively applies function f to every nth row
invert	Turns all ups to downs, and vice versa
zipAnd a b	Combines two lists, resulting in up when both lists have an up
zipOr a b	Combines two lists, resulting in up when one or both lists has an up
zipXOr a b	Combines two lists, resulting in up when only one list has an up

Table 1.1: The values, functions and operators available in the Live Loom code interface.

Working at the Live Loom

Figure 4 shows the Live Loom software interface next to the woven outcome. This starkly shows the perceptual gap between code, draft and weave, with little visual correspondence despite the structures of the draft being a logical outcome from the code, and the weave being that of the draft. The code is represented as a branching tree, the

visual interface directly showing the branching normally represented by parenthesis (McLean and Wiggins, 2011). This particular code creates the draft pattern shown, which perhaps has the appearance of vines growing up a wall. When this structure interferes with the alternating colours of warp and weft, the final result appears in the weave as (to my eyes) legs leaping into the air (Fig. 4b).

It is humbling that this leap from draft to weave constitutes ancient knowledge, demonstrating mathematical logic that predates our conventional view of mathematics. This brings historical grounding to the analogous logical leap from code to draft, shown alongside.

It is worth noting that adding an additional level of abstraction to the drafting of weaves is not novel. Indeed, it is very common in weaving for patterns of threads to be grouped into a number of shafts, where weave structure is created by patterns for lifting these shafts. The binary grids we consider in the present paper are in such cases drawn down from a draft composed of of threadings (how the threads are grouped into shafts) and treadlings (how the shafts are lifted over time). In a sense, the live coding language introduced in this paper provides a flexible, interactive alternative to lift plans.

Music of the loom

The solenoids are not triggered at once but in sequence, to even out the use of electrical power, with less needed to hold a solenoid than to move it. The most time-efficient way to do this would be to trigger the 'up' threads, pulling the warps one after the other, evenly spaced in time. However I have found it much more useful to include 'down' threads in the timing, so each row takes the same amount of time to actuate, no matter how many warps are being pulled forward. This gives a clear rhythm to each row, where the 'clunk' of a solenoid is heard for an up, and a silent pause is heard for downs. This rhythm breathes life into the weaving process, making it easier to orient myself in the pattern and spot errors, as I compare the rhythm I hear with the threads I see. It also brings rhythmic enjoyment to the repetitive nature of weaving, compelling me forward into the next row.

The solenoids have a particular 'duty cycle', meaning that it is best not to keep them activated for too long, otherwise they may overheat. Once a solenoid is activated, the micro-controller holds it in place, giving enough time for the weaver to place a hand on the heddles and pull selected warp threads forward. Although born from a technical need, these few seconds add an additional sense of regulated timing to the process of weaving. However if the heddles are not caught in time, the weaver-coder can repeat the lift with a quick press of the up arrow key. The weaver can also unweave by stepping backwards through the structure with the left arrow, removing rather than adding the weft by hand, for each step.

Live Coding

So far we have discussed action, but not live reaction. We have looked at coding the loom with a draft, and coding the coding of the loom by introducing a language for composing a draft, but we haven't discussed live coding - the changing of code in response. Let's do that now.

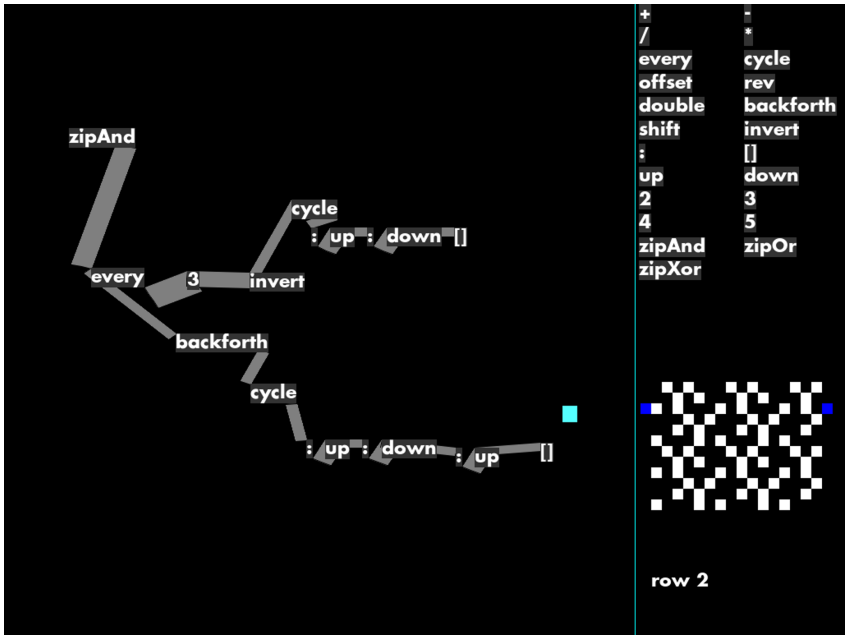
Changing patterns

Live coding of music is often characterised by comparatively slow, continuous changes. Changes are heard immediately, but the complexity of music grows with the code. The experience of the Live Loom is rather different, where a small change tends to have a large, global effect, but each change takes time to become apparent; rows are only produced at a rate of a few per minute, and it might take two or three repeats of a pattern before its nature can really be felt. These big differences from small edits are due to multiple levels of interference, between code, draft and weave.

A change from one pattern to the next also presents a problem of transition, where one pattern might not sit well with the next, potentially creating a physically uneven structure, with undesirable floats (see below). It can take a disturbed row or two before the weave set-

tles into the next structure. There are certainly parallels here with live coding music and indeed music in general, where a sudden change can be jarring, without a careful transition. Managing this transition is probably best done at the loom, adjusting each shed at the heddles by hand.

At this slow pace of change though, we are in the domain analo-



(a) Code (left) and resulting draft structure (bottom right)

(b) Resulting weave

Figure 4: Live Loom software interface and the woven result

gous to slow coding rather than frenzy of an algorave. Where each decision has long term consequences, there is a need for careful consideration. Furthermore in weaving we work with physical thread, rather than with the metaphorical thread of time as with live coding of music. This means that we are able to undo a weave in a way that we cannot undo music, and change our minds. By unweaving, the weaver, like the mythological figure of Penelope, resists external forces.

Embracing error

Live coders are known for embracing error, and so it is fortunate that it is so easy to produce a draft which is unweavable. For example, there is the problem of 'floats', lengths of unwoven fibre created wherever there is a contiguous series of either ups or downs in the warp or weft direction. Indeed, where there are only either ups or downs in a given row or column, that thread will not be woven into the fabric at all. In response to a problematic draft, the weaver can do one of three things – change the code to look for a more weavable draft, ignore activated heddles or pull additional ones to change the weave directly, or just attempt to weave the pattern anyway.

In the draft shown in Figure 5a, the draft looked unweavable to my naive eyes, due to the pairs of identical rows within it. Where this happens, pairs of consecutive wefts are passed through the same shed. I thought this would result in a mess, but out of curiosity went ahead anyway to produce the weave shown in Fig. 5b. I found that with care the wefts would still run parallel and stay in order, largely maintaining the 'correct' structure on-screen. Furthermore, because the repeat in the draft consists of an odd number of rows, and I was weaving with two different wefts, the wefts would alternate between either travelling through the same shed from one side to the other together, or in opposite directions. By embracing this 'error' I arrived at a (to me) surprising, pleasing, and subtle result, although there are undoubtedly many such surprises on the way to becoming an experienced weaver, and I have far to go.

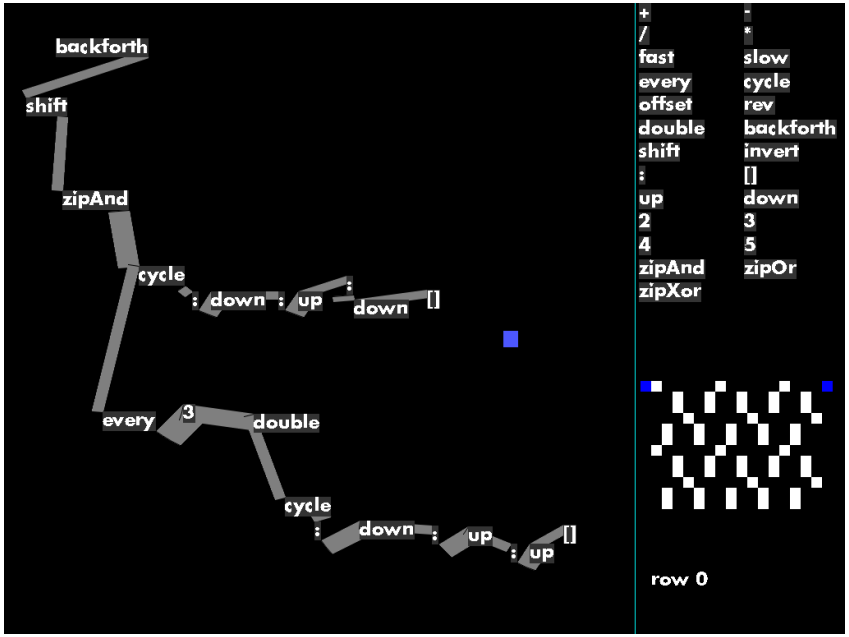
Weaving the edit

Decisions at the Live Loom are taken slowly, responding to problems and opportunities as they arise in the weave. Figure 6a shows the starting point for another improvised weave, a draft appearing to be a kind of hatched vertical pattern, drifting downwards to the left, with lines sometimes joining or breaking. When it came to weaving this structure (see Fig. 7), two features slowly became apparent – the pervasive pairs of ups and downs on the weft, offset from one row to the next, seemed to result in the warp spreading out vertically, and therefore partially hiding the warp at points where I expected it to be visible. This created an a partly weft-faced weave. However, some long floating threads were present on the warp direction, and the weft-facing only accentuated the presence of these long warps lying on top.

After weaving 20 rows of this pattern (Fig. 7), I hit a snag - the pattern of repeating warp floats drifted until they sat at both edges of the fabric, seen in Fig. 6b. I realised that having floats at the selvage would cause the textile to lose its otherwise uniform width, and I decided I neither wanted this effect or to change it by hand; I had been enjoying working the two wefts together at the selvage, and felt that having a warp float there would create a mess. So instead I changed the structure to that seen in 6c, adding code to invert every other row, as an effort to break up warp floats. However, after a few rows of weaving the edit (Fig. 7) to the point in the interface shown in Fig. 6d, I realised that by breaking up some floats, I had only created new ones. Another tweak shown in Fig. 6e, this time changing a number from 3 to 1, seemed to fix it. However once I started weaving I realised the floats were still there, but now so long that they took up the whole edge and so were no longer visible on-screen!

This time I decided having such long floats was an interesting enough challenge to pursue, and embraced this compounded error as an opportunity to experiment more with creating extra binding points at the selvage by hand. I continued with this structure for 53 rows, up until the point seen in Fig. 6f. The resulting weave shown in 7c

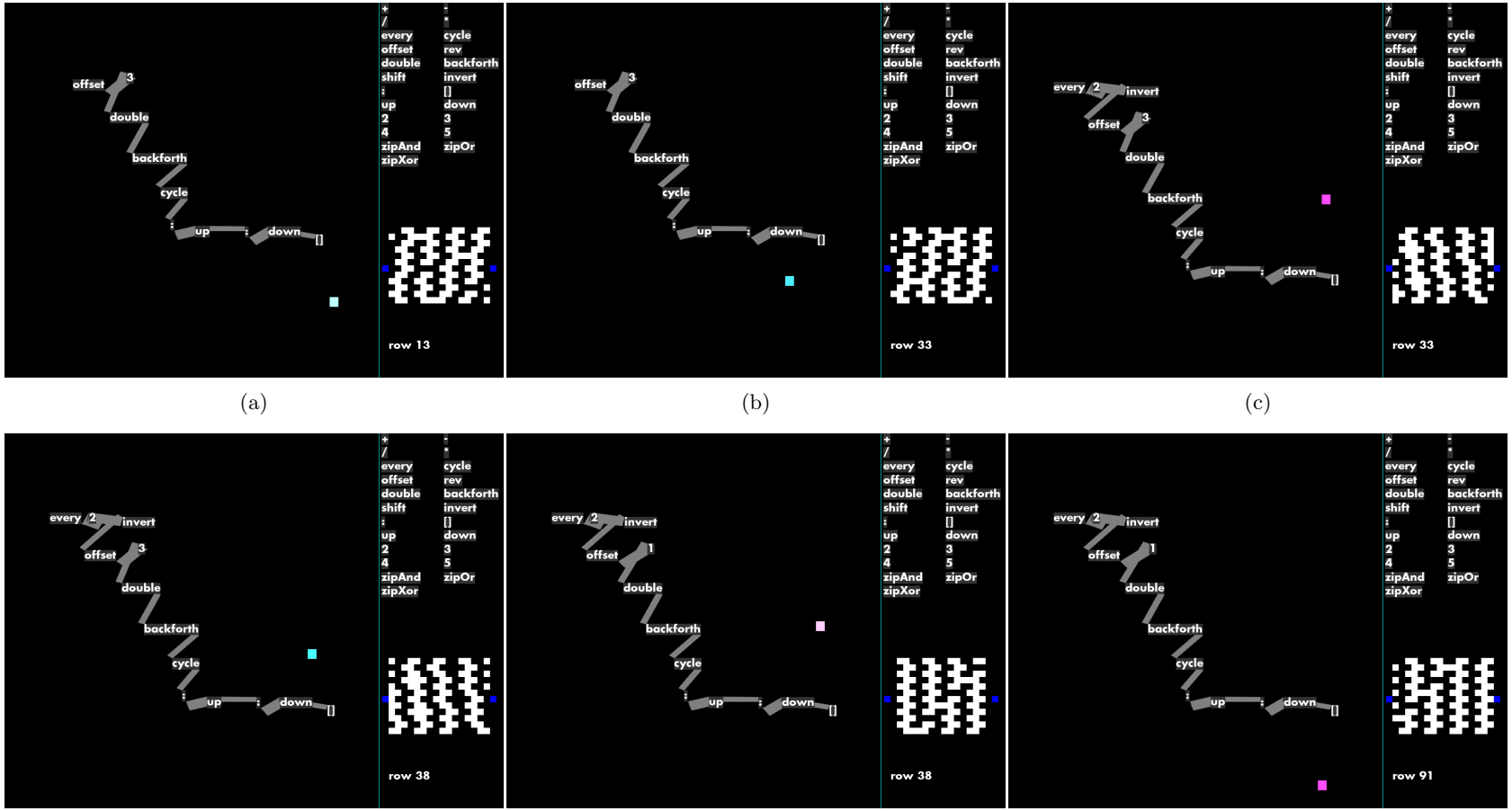
did indeed turn out to be interesting, the resulting weave curiously appearing to be much more coherent than the draft pattern. As the long floating warp threads stepped one warp to the left, they cycled between white and blue, over a steep diagonal. This time, the resulting motif reminded me of Quipu knots. The resulting experimental weave we have seen in Fig. 7 charts an experiment in three stages.



(a) Live Loom code and draft showing pairs of identical rows.

(b) The resulting weave, showing pairs of wefts that have travelled through the same shed.

Figure 5



(a) (b) (c)

(d) (e) (f)

Figure 6: Screenshots of Live Loom interface at six different points in the weaving of fabric (see Fig. 7)

First, the initial serendipitous discovery of a) a weft-faced structure with warp floats. Then transition b) as I searched for a solution to a perceived problem at the selvage. Finally a longer section c), with some manual experimentation at the selvage. The resulting fabric tells a story of its making, from a starting point, to prevarication and decision, with further learning points charted along the edge as I learned to deal with the selvage.

Conclusion

This paper has explored how the principles of live coding may apply to the warp-weighted loom. However, in connecting a live coding pattern language to the practice of weaving, we find that weaving is already abundant with computational patterns, and in particular that historical drafting techniques already demonstrate a similar computational abstraction from the resulting woven textile, as code does from media in the live coded performing arts. Nonetheless by adding another layer of abstraction to that which has been present in weaving since ancient times, and using solenoids in communicating movement from the code to the weaver, the Live Loom allows creative exploration of woven patterns in a way that is sympathetic to the repetitive, yet cognitive nature of hand-weaving. There is much to follow the preliminary work introduced here. This paper has purposefully focussed on understanding of weave from the perspective of live coding, taking care to have respect for this technological craft that has developed since Ancient times. It could be however that weaving practice could benefit from such a computer language interface, for example replacing the current relatively time-consuming process of uploading bitmap images whenever the pattern is changed on a TC2 loom. Introducing 'real' trained weavers to the Live Loom would undoubtedly also turn up valuable criticism of its design. Furthermore while some hands-on workshops have already been conducted, more involved long-form work with workshop participants are needed to explore the possibilities of the loom in helping people explore the complexities and possibilities of hand-weaving.

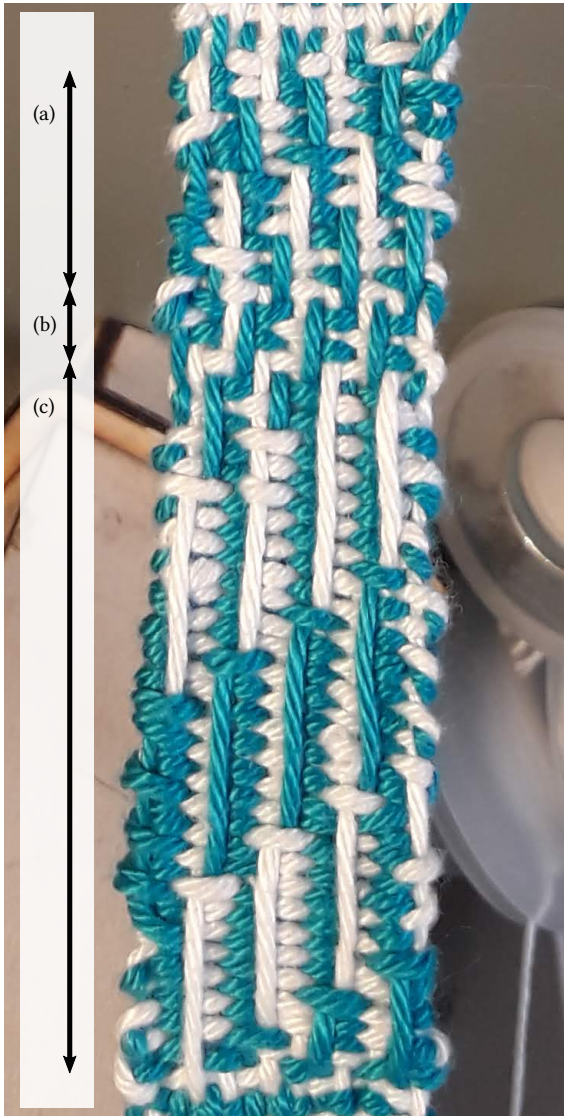


Figure 7: Result of improvised weave edits shown in Figure 6

Acknowledgements

This research is conducted by the PENELOPE project, with funding from the European Research Council (ERC) under the Horizon 2020 research and innovation programme of the European Union, grant agreement No 682711.

References

Armitage, J., 2018. Spaces to Fail in: Negotiating Gender, Community and Technology in Algorave. Dancecult: Journal of Electronic Dance Music Culture 10, 31–45. https://doi.org/10.12801/1947-5403.2018.10.01.02

Emery, I., 2009. The Primary Structures of Fabrics: An Illustrated Classification, 01 edition. ed. Thames; Hudson Ltd, New York, N.Y.

Harlizius-Klück, E., 2017. Weaving as binary art and the algebra of patterns. TEXTILE Cloth and Culture 15, 2017, 176–197. https://doi.org/10.5281/zenodo.3342554

Harlizius-Klück, E., Fanfani, G., 2017. (B)orders in Ancient Weaving and Archaic Greek Poetry. https://doi.org/10.5281/zenodo.840005

Hicks, M., Aspray, W., Misa, T.J., 2017. Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing, 1 edition. ed. MIT Press, Cambridge, MA.

Kurbak, E. (Ed.), 2018. Stitching Worlds: Exploring Textiles and Electronics. Revolver Publishing, Berlin.

McCallum, D.N.G., 2018. Glitching the Fabric: Strategies of new media art applied to the codes of knitting and weaving.

McLean, A., 2019. Kairotic/liveloom: Solenoid two. https://doi.org/10.5281/zenodo.3346032

McLean, A., Harlizius-Klück, E., 2018. Fabricating Algorithmic Art, in: Parsing Digital. Austrian Cultural Forum, London, UK, pp. 10–21. https://doi.org/10.5281/zenodo.2155745

McLean, A., Wiggins, G., 2011. Texture: Visual notation for the live coding of pattern, in: Proceedings of the International Computer Music Conference 2011. pp. 612–628.

Plant, S., 1998. Zeros and Ones: Digital Women and the New Technoculture, New Ed edition. ed. Fourth Estate, London.

Sicchio, K., 2014. Data management part III: An artistic framework for understanding technology without technology. Media-N: Journal of the New Media Caucus 10.

Turkle, S., Papert, S., 1990. Epistemological pluralism: Styles and voices within the computer culture. Signs 16, 128–157.

Cibo v2: Realtime Livecoding A.I. Agent

Jeremy Stewart
Rensselaer Polytechnic Institute, Troy, NY
stewaj5@rpi.edu

Shawn Lawson
Rensselaer Polytechnic Institute, Troy, NY
lawsos2@rpi.edu

Mike Hodnick
mike@kindohm.com

Ben Gold
bgold.cosmos@gmail.com

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

Cibo v2 is a live-coding artificial intelligence (AI) agent that performs TidalCycles and is trained on recorded performances by several TidalCycles performers. This paper presents an entirely new architecture from the original Cibo agent for realizing autonomous performing agents.

Introduction

Cibo v2 represents an entirely new architecture from the previously presented Cibo live-coding agent (Stewart and Lawson, 2019; Stewart, 2019, Cibo: Safeguard II). Cibo v2 performs TidalCycles (McLean and Wiggins, 2010; McLean et al, 2019) code in a live-coding setting for musical and sound performance. Cibo v2 is constructed with autoencoder and variational autoencoder architectures as its foundations, with additional neural network modules governing performance progression and variable production. Cibo v2 trains faster and is trained in a number of steps, allowing for a greater degree of flexibility during the development process. The resulting performance agent produces TidalCycles code that is highly reminiscent of the provided training material, while offering a unique, non-human interpretation of TidalCycles performances. Cibo v2 is trained using recordings of performances by several human performers, including Mike Hodnick, Ben Gold, and Jeremy Stewart. For the first time, the agent in performance hints at each of these influences. Furthermore, the manner by which the agent is constructed allows for visualizing the organization of training material, allowing us to peer into the learned AI-representation of these recordings. Finally, because each human contributor performs with a unique sample corpus, a sample analysis tool set is offered to allow the Cibo v2 agent to effectively substitute samples based on sonic features.

Pre-Agent Technical Overview

The first step to creating the Cibo agent include recording TidalCycles performances by human performers and tokenizing the TidalCycles code to prepare it for training purposes. The following section outlines the metholodies and software developed to satisfy these functions.

Recording Tidalcycles(Sublime JENSAARAI)

Jensaarai, a custom text editor written in NodeJS with Electron, was integral to the initial version of Cibo. Jensaarai created text recordings of all edits and code executions that a human performer would make. These recorders were fed into Cibo for its training process. When Cibo performs, Cibo's edits and code executions are made visible in the Jensaarai editor. Cibo v2 uses exactly the same process, with several differences in the text editor. Human performers making text recordings and Cibo v2 performances use Sublime Text with the Sublime Jensaarai plugin (Lawson, 2019). Using this plugin format reduced numerous issues that were occurring when the userbase of Jensaarai grew beyond the primary developers and Cibo. The functionalities of the Sublime Jensaarai plugin are nearly identical to the standalone Jensaarai application.

LEXER/TOKENIZATION

Tokenization of TidalCycles code is carried out using the PLY library (Beazley, 2018), much like was described in "Cibo: An Autonomous TidalCycles Performer" (Stewart and Lawson, 2019). The lexer dictionary contains 248 discrete tokens, including all of those available in the TidalCycles documentation (All the Functions - TidalCycles), as well as additional tokens for custom function definitions. Variable values are stored in a second vector (represented as the "values" vector in Figure 1), and are replaced with INTEGER, FLOAT, and STRING tokens.

We then convert this single token representation into an n-gram encoding, combining up to four tokens into a single integer representation, as seen in Figure 2, where the tokens [0, 30, 17, 28] are combined into a single token: 40. Only n-gram tokens occurring in the training material are preserved, resulting in a dictionary of 11267 discrete tokens—much fewer than the possible 2484 tokens. These n-gram tokens are used as the input and output of the AutoEncoder architectures described below.

Agent Architecture

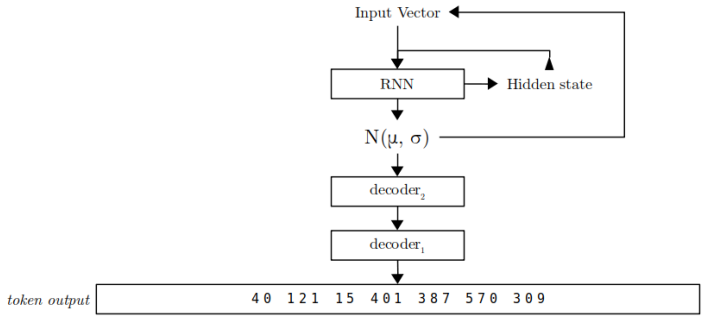


Figure 3: CiboV2 performance-ready architecture. Input vector is a three-dimensional vector denoting the latent space coordinates which are to be decoded by two stages of decoder module, resulting in n-gram output

The Cibo agent is trained, first, to construct compact encodings of TidalCycles code via an autoencoder and a variational autoencoder. The training of these neural networks produces a three-dimensional latent space: TidalCycles code can be converted into a point in this latent space, and, inversely, coordinates in this latent space can be decoded into usable TidalCycles code. The Cibo agent performs by traversing the three-dimensional latent space using a recurrent neural network (RNN), producing TidalCycles code at each step. The

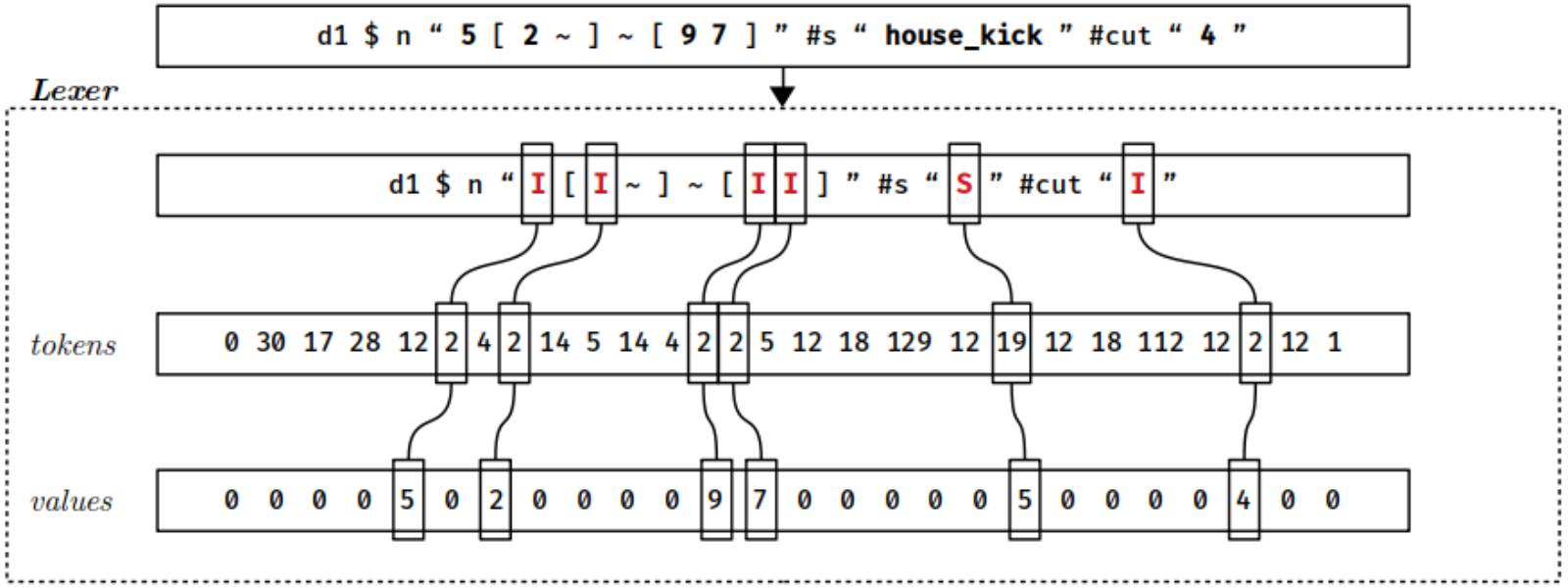


Figure 1: TidalCycles code is input into the lexer, which lexes the code in discrete tokens. Integer, Float, and String tokens are indicated as such, while preserving the values associated with these tokens in a second vector.

RNN module, seen at the top of Figure 3, is trained to traverse the resultant latent space in a manner consistent with the TidalCycles recordings used for training. Importantly, this module outputs normal distributions which are sampled, resulting in non-deterministic behavior. The basic performance-ready architecture is represented in Figure 3. "Decoder1", seen at bottom, is the first component to be trained, followed by "decoder2". These two modules together convert a 3-dimensional latent space vector into an n-gram token sequence which can be converted to TidalCycles code. The following sections will detail each of the neural network components of the Cibo agent.

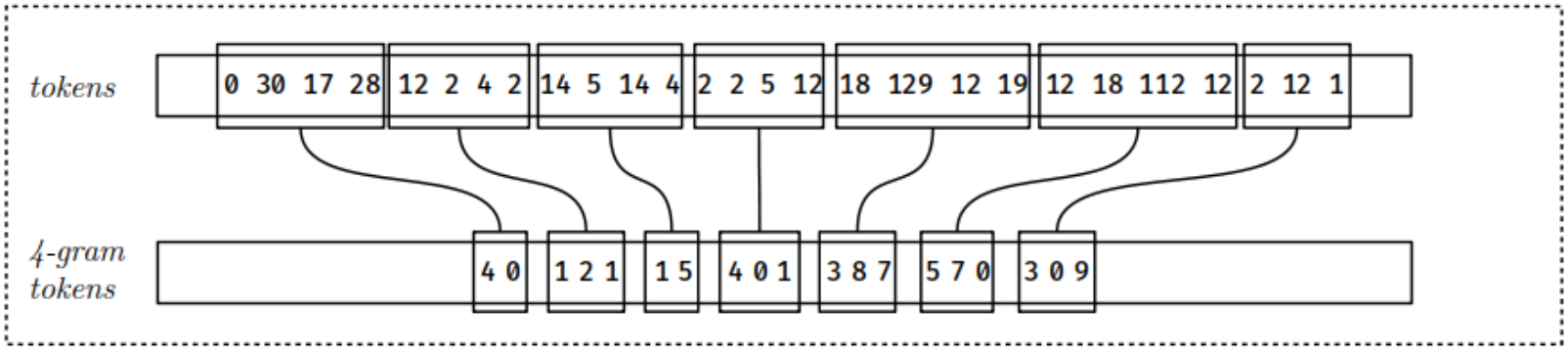


Figure 2: Tokens are combined into 4-gram representations.

Autoencoder

Once n-gram token sequences are produced by the lexer, the first stage of the Cibo agent must convert these variable-length sequences into a fixed-length representation. In order to accomplish these, we employ an autoencoder architecture, seen in Figure 4.

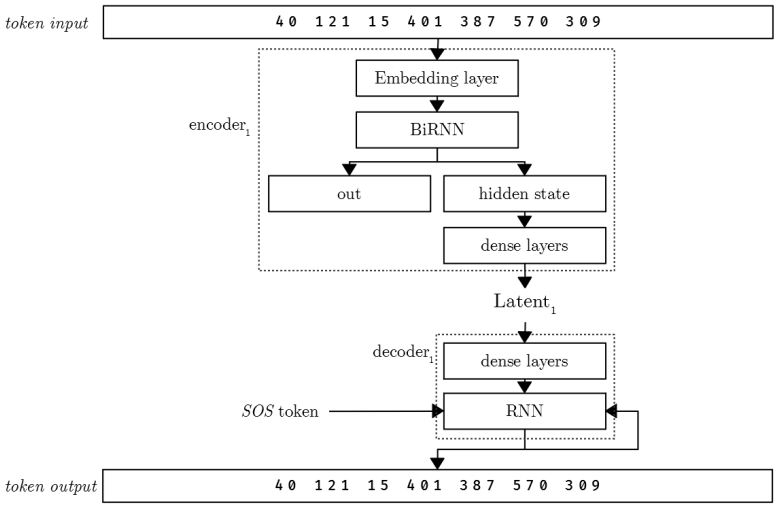


Figure 4: Training the first stage autoencoder neural network. Variable length input (n-gram token input, at top) is converted to a fixed-length latent representation (Latent1), and then decoded back to a variable length output (token output, bottom).

The token input passes, first, through an embedding layer, which converts the discrete, integer token notation into a floating-point vector in a higher-dimensional space. The output from the embedding layer is sent to a bidirectional recurrent neural network (BiRNN) which reads the sequence both forward and backward, producing two outputs: a variable-length out vector and a fixed-length hidden state vector. The out vector is discarded. The hidden state vector is passed through a fully connected (dense) layer in order compress the encoding further, resulting in a vector that contains 500 values: Latent_1 (This Latent_1 vector will be used to train the subsequent VAE module, discussed in the next section). The latent vector is then fed into the decoder module, which is trained to convert it back to the original input sequence, thus token input and token output are the same sequence.

Variational Autoencoder

With the first stage autoencoder complete and trained, we construct a variational autoencoder (Kingma et al, 2013) to further compress the latent representations of TidalCycles code. The variational autoencoder (VAE) is constructed by alternating linear (fully connected) layers with ReLU (Rectified Linear Unit) activation functions. The encoder, labeled encoder_2 in Figure 5, outputs two vectors which are used as the mean and standard deviation of a normal distribution (seen at center of Figure 5). This normal distribution is then sampled and input into the decoder (decoder2) which reconstructs the Latent_1 vector.

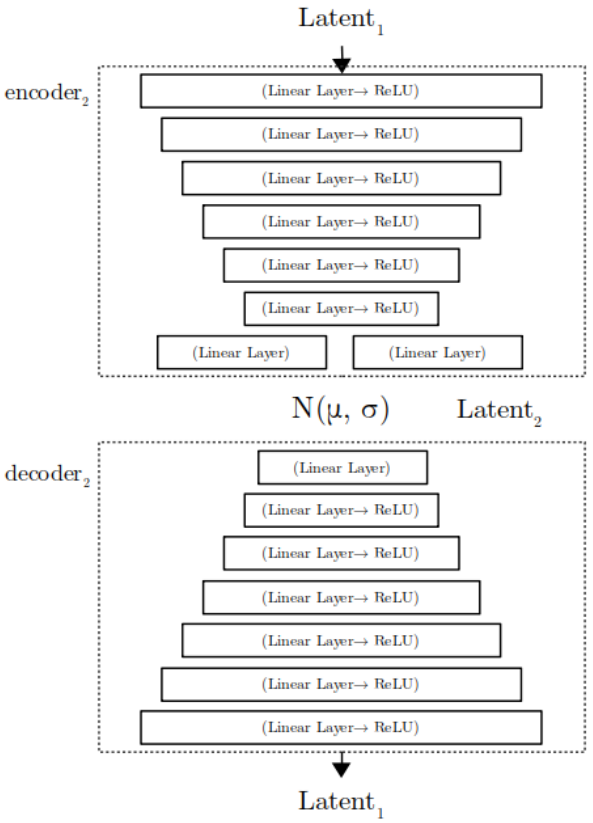


Figure 5: Training the second stage variational autoencoder (VAE). The learned latent representation from stage 1 (seen in Figure 4) is used as input and output here. The encoder module produces a normal distribution which is sampled before being sent to the decoder, resulting in non-deterministic behavior.

During the development of the Cibo agent and many training experiments, many different hyperparameter settings we tested, includ-

ing different numbers of layers, different numbers of hidden features between each layer, and different activation functions. Figure 5 accurately represents the architecture as it is currently in use, with 7 layers in each the encoder and decoder. The Latent1 input/target vector contains 500 features, while the innermost latent representation—the result of sampling the normal distribution (and the input into decoder2)—contains 3 features.

The variational autoencoder provides two strengths in this implementation. First, because the encoder (encoder2) produces a normal distribution which is then sampled, the VAE is non-deterministic and retains a degree of stochasticity in its final execution. Additionally, the resulting latent space produces gradual changes in the output, i.e., the latent space, while highly complex and knotted, can be smoothly traversed for continuous subtle changes in output.

Latent Space Traversal

*

Once the latent space has been constructed by training the autoencoder and subsequent VAE (discussed above), all training recordings are passed through the encoders, producing a normal distribution for each training sample. Once these latent normal distributions are generated, we can visualize the latent space (see Figure 6). The three TidalCycles contributors—blindelephants (Jeremy Stewart), kindohm (Mike Hodnick), and bgold (Ben Gold)—are signified through different colors, with each point representing an execution of a line or block of TidalCycles code. Figure 6 was produced via t-SNE using the latent-space vectors produced by the encoders (Maaten et al, 2008). There are clear regions in this visualization, with groupings of the same color in some areas, and mixing in others.

With this latent space constructed by the two encoder neural networks, all training recordings were encoded, resulting in a sequence of normal distributions for each recording. More simply put, with the latent space constructed as in Figure 6, it is possible to trace pathways through this space which represent a given performance. These sequences were used to train a recurrent neural network (using a LSTM architecture; Hochreiter, 1997) in an autoregressive fashion. The resulting model will trace a pathway through the latent space (in Figure 6) in a manner based on the training recordings, at each step producing a vector which can be sent to the two decoder modules and output as TidalCycles code.

Generating Token Values

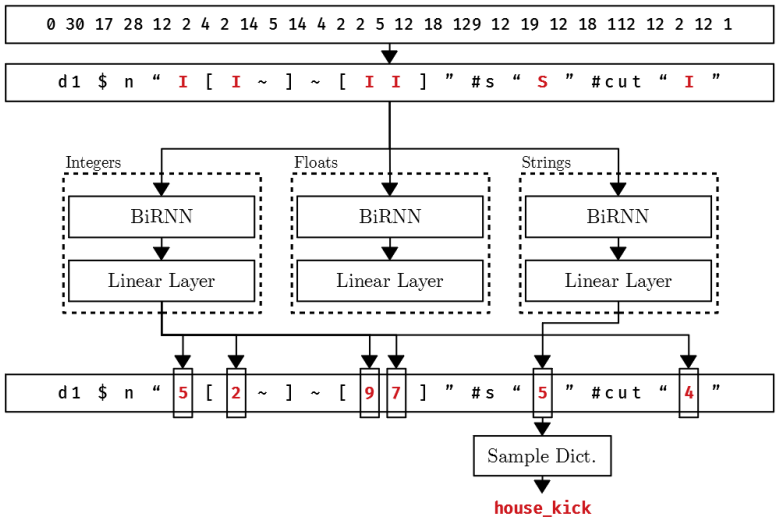


Figure 7: Generation of token values takes place by 3 neural networks, one for each variable type: Integer, Float, and String

The token sequence that is generated by the agent modules discussed above will contain Integer, Float, and String tokens, without yet specifying their values. The last step in the Cibo agent's generative process is to produce these variables. It does so with three neural networks, each dedicated to one of the variable types. The

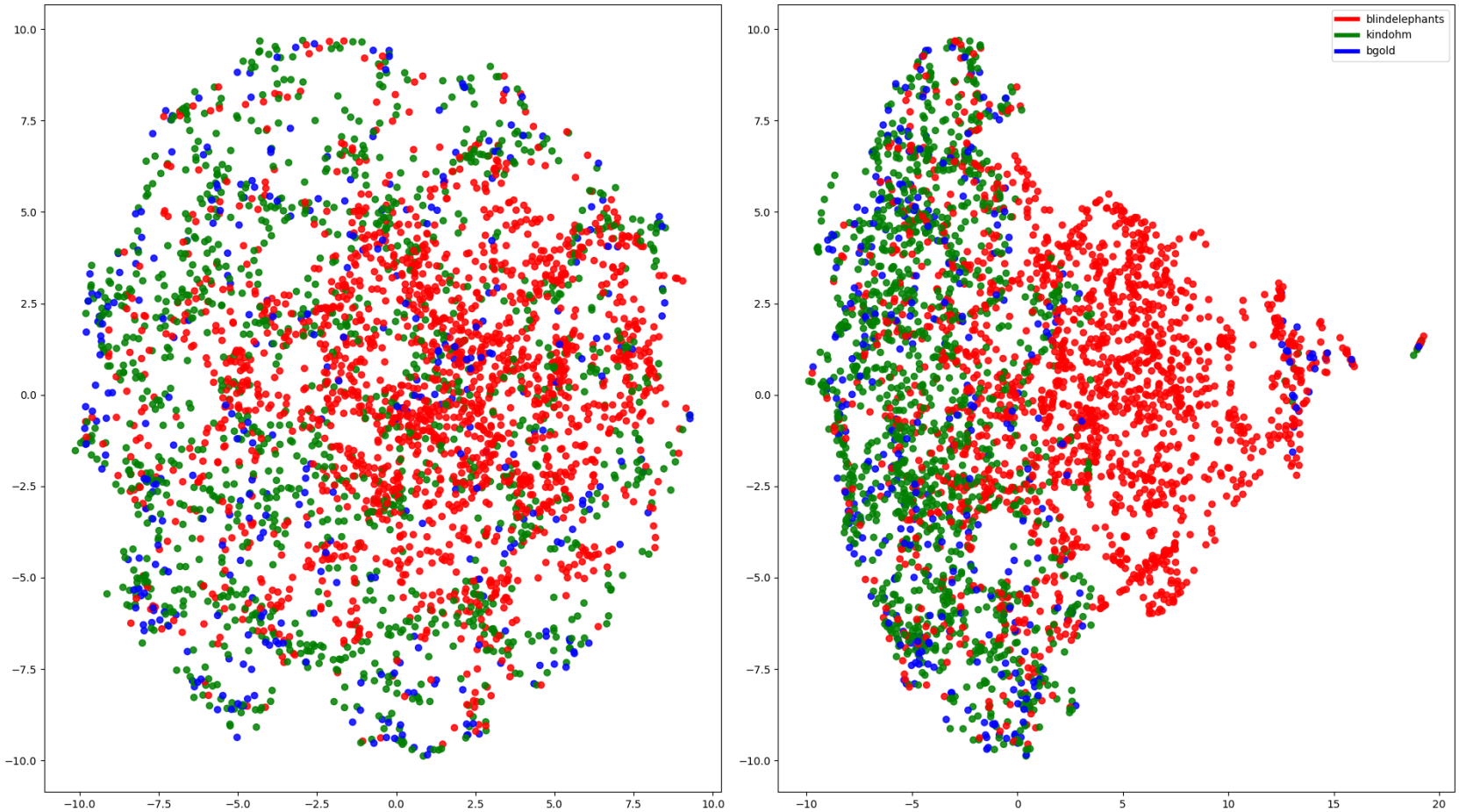


Figure 6: The latent space produced by training, visualization aided by t-SNE processing. Each point represents a single execution of a TidalCycles line or block. The two subfigures represent the same space, rotated by 90 degrees around the y-axis.

TidalCycles token sequence is read into a bidirectional recurrent neural network (BiRNN) which generates a variable length output (equal to the sequence length). Indices of the given variable type are then passed through a linear layer which outputs a single value. If the variable type is float, the output is used directly; integer type will result in the output being truncated and used. If the variable is of string type, the value will be used to look up a sample name from an available dictionary which contains all currently available samples.

Sound Analysis-Based Substitution

TidalCycles and SuperDirt make using one's own sample library very straightforward. While recording performances to use for training the Cibo agent, each contributor used a unique sample library—potentially containing a combination of publicly available sound samples (such as those included with SuperDirt) and samples created by each individual artist. While the agent is trained in a manner that preserves all of these unique sample names and possibilities, there is no guarantee that all of these samples will be available at the time of performance. Therefore, we developed a small tool set (Stewart, 2019, Sample Corpus Sound Analysis) to analyze each sample, producing a fixed-size vector that represents various spectral features, including: centroid, bandwidth, contrast, flatness, and rolloff; as well as a full STFT (short time fourier transform) and CQT (constant Q transform). This analysis tool uses the Librosa audio analysis library. From each of these analyses (the length of which would be based on the duration of the sample) a mean, standard deviation, and sum vector was calculated. Thus, each sample is represented by a vector containing 3360 values.

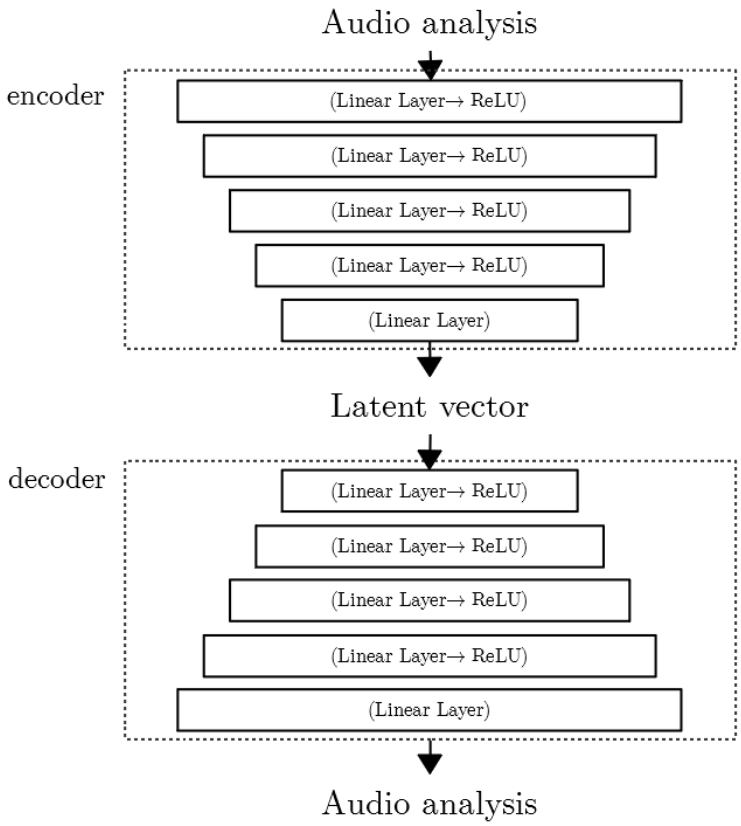


Figure 8: Dimensionality reduction performed on sample audio analyses with an autoencoder.

An autoencoder neural network was constructed for the sake of dimensionality reduction, allowing us to effectively reduce each sample's representation from 3360 values to 300 values. This neural network, seen in Figure 8, contains five layers in each the encoder and decoder.

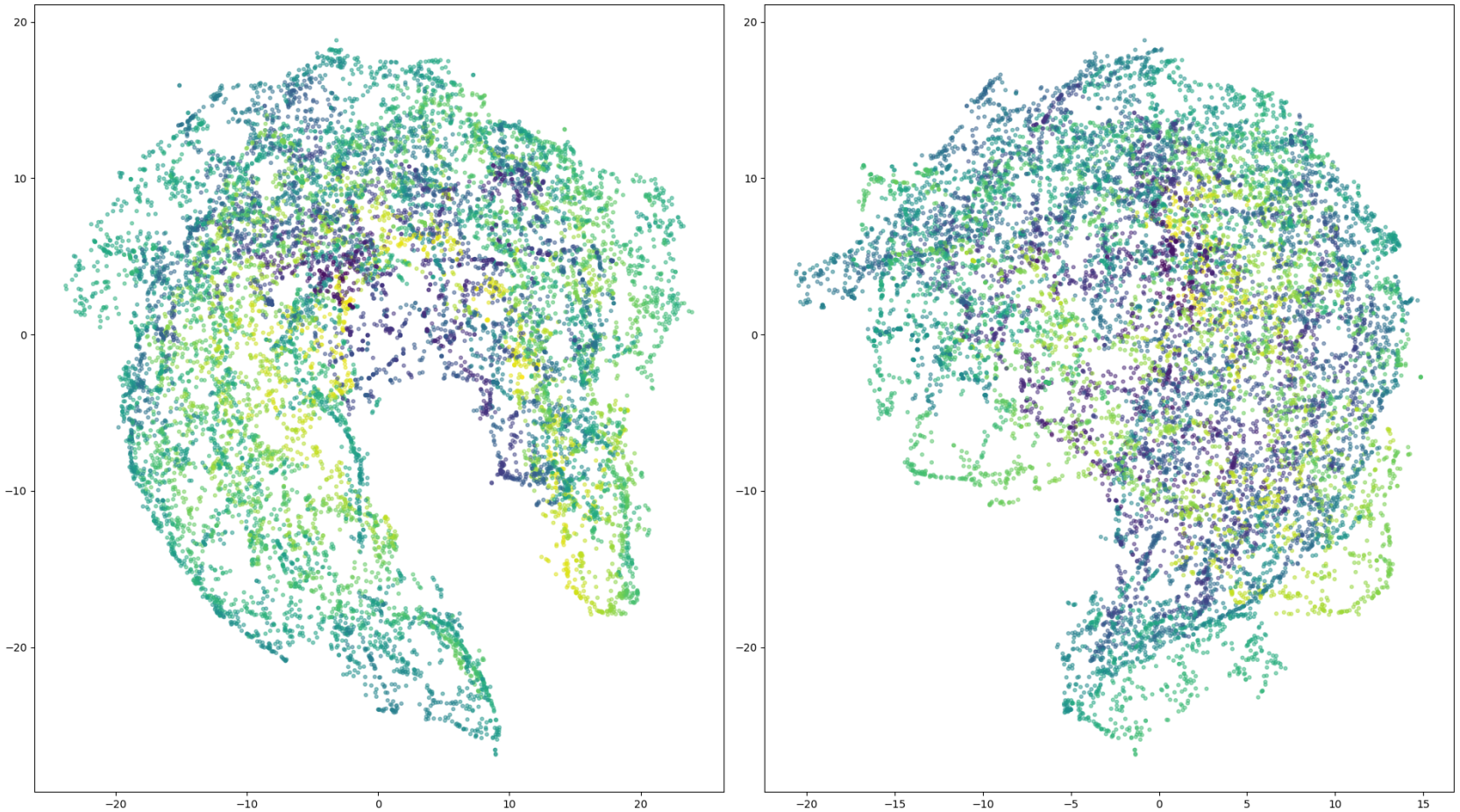


Figure 9: t-SNE visualization of the latent space produced by the autoencoder. Each point represents a single sample, with 13340 in all. The two subfigures are of the same space, rotated 90 degrees on the y-axis. Z-depth is represented by color.

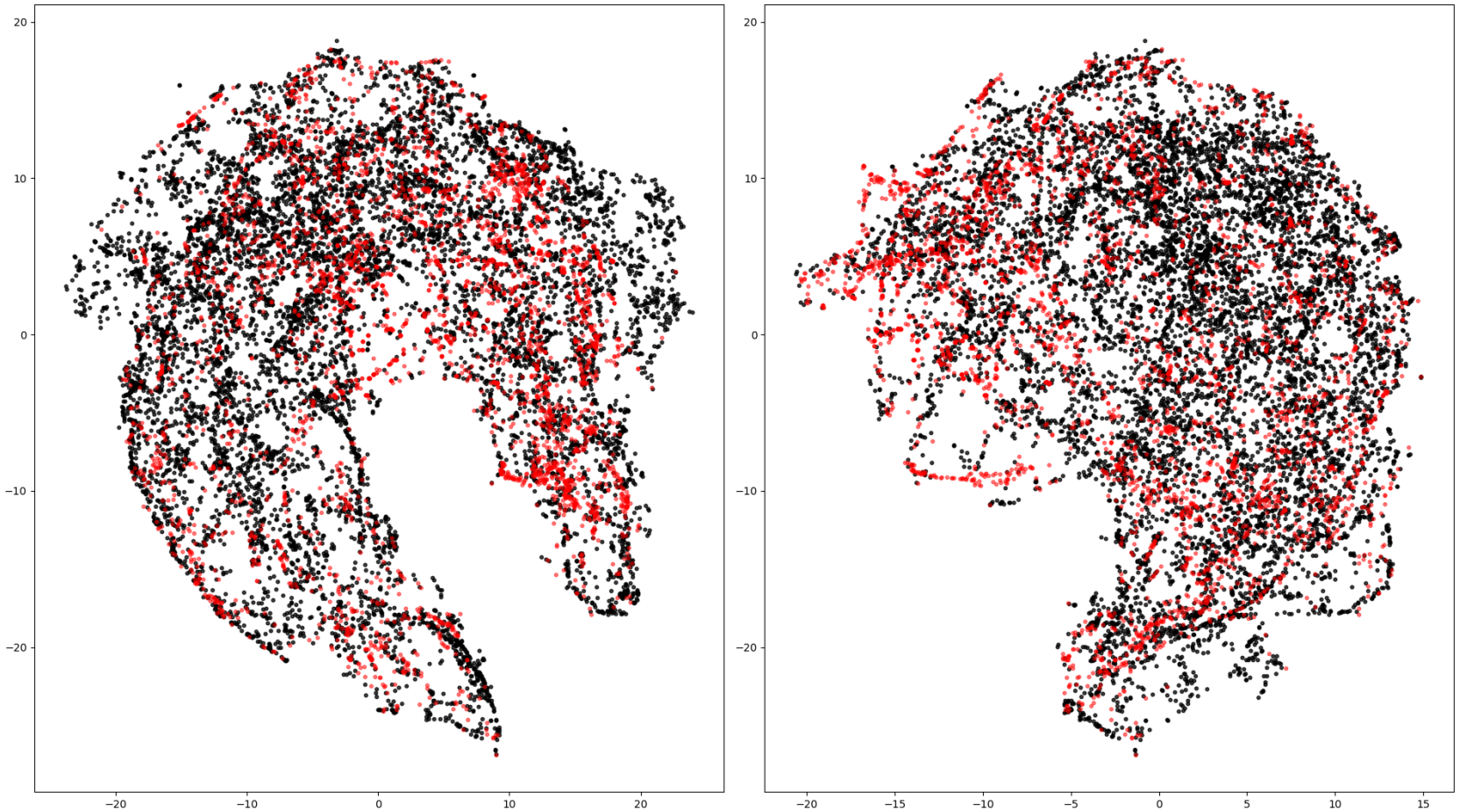


Figure 10: Here, the mapping is made more apparent. Red points indicate currently unavailable samples, while black points are those sample currently available. The substitution method described here will map each red point to the nearest black point.

*

From here, a table is produced that maps all unavailable samples to available samples, based on proximity of latent vectors. This table is stored for use by the agent during performance. If the agent produces a sample name which is not in the currently available sample library, this table is used for a simple name substitution.

*

All sample analyses (those containing 3360 values each) are preserved, allowing for recalculation of the substitution tables if the available sample library changes or if new training recordings, referencing additional samples, become available.

Results

The Cibo v2 agent, as described above, performs as a live-coding performer using TidalCycles code in a self-directing manner based on learned characteristics from training material. Video documentation of the agent's performance (Stewart, 2019, Cibo V2 Demo) demonstrates Cibo v2's ability to produce valid TidalCycles code in way that is highly reminiscent of the contributing performers, while also producing novel output that effectively blends between these influences.

Conclusions

Cibo v2 represents a novel implementation of artificial intelligence (AI) and machine learning (ML) techniques to create an autonomous live-coding performance agent. The construction of Cibo v2 around autoencoder and variational autoencoder modules results in a faster training process (than previous versions of the Cibo agent, specifically) and more robust sequence generation. Additionally, the ability to visualize the learned latent space (Figure 6) allows us to begin to develop a better understanding and conceptualization of how TidalCycles performances might be organized and correlated.

References

All the functions – TidalCycles, viewed 25 September 2019, `https://tidalcycles.org/index.php/All_the_functions`

Beazley, D., 2018. PLY (Python-Lex-Yacc), viewed 25 September 2019, `http://www.dabeaz.com/ply/`

Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R.,

Szerlip, P., Horsfall, P. and Goodman, N.D., 2019. Pyro: Deep universal probabilistic programming. The Journal of Machine Learning Research, 20(1), pp.973-978.

Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. Neural computation, 9(8), pp.1735-1780.

Kingma, D.P. and Welling, M., 2013. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114.

Lawson, S., 2019. Sublime Jensaarai, viewed 26 September 2019, `https://github.com/shawnlawson/SublimeJensaarai`

LibROSA, viewed 25 September 2019, `https://librosa.github.io/librosa/`

Loshchilov, I. and Hutter, F., 2017. Fixing weight decay regularization in adam. arXiv preprint arXiv:1711.05101.

Maaten, L.V.D. and Hinton, G., 2008. Visualizing data using t-SNE. Journal of machine learning research, 9(Nov), pp.2579-2605.

McLean, A., et al, 2019. Tidal, viewed 6 December, 2019, `https://github.com/tidalcycles/Tidal`

McLean, A., and Wiggins, G., 2010. Tidal—pattern language for the live coding of music. Proceedings of the 7th Sound and Music Computing Conference.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A.,

Antiga, L. and Lerer, A., 2017. Automatic differentiation in pytorch.

Stewart, J., 2019. Cibo: Safeguard II, viewed 26 September 2019, https://vimeo.com/288889990/6d92fd1a55

Stewart, J., 2019. Cibo V2 Demo, viewed 26 September 2019, https://vimeo.com/361567860/abac29e10f

Stewart, J., 2019. Sample Corpus Sound Analysis, viewed 26 September 2019, https://github.com/BlindElephants/SampleCorpus_SoundAnalysis

Stewart, J. and Lawson, S., 2019. Cibo: An Autonomous TidalCyles Performer.

Williams, R.J. and Zipser, D., 1989. A learning algorithm for continually running fully recurrent neural networks. Neural computation, 1(2), pp.270-280.

Re-coding the Musical Cyborg

Jacob Witz
golmechanics@gmail.com

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

Where does the line between musician and instrument exist, and can we redraw it? This paper defines the concept of the "musical cyborg:" a model for synthesizing human creativity and digital algorithms to create sounds that would not otherwise be possible through human production alone, rooted in the theories of thinkers like Juan Atkins, Kodwo Eshun and Donna Haraway. It then explores several different constructions of musical cyborgs, each demonstrating various degrees of opacity and accessibility: Kindohm, digital influencer-turned-musician Lil Miquela and Spotify-focused bands all rewrite the boundaries between human and machine in ways which effect how their work is consumed and replicated across digital networks. Most notably, the musical cyborg of the live coder is invoked in order to provide a potential model for future constructions of art-generating cyborgs. The paper assesses the impact of various open-source live coding projects in order to determine how these design choices can be replicated across a broader network of artistic practices. It concludes with the understanding that, while current technologies for open-source collaboration only presently exist through generous external funding, they nevertheless provide an imaginative and achievable vision for future projects, software and communities both within and beyond music.

Introduction

The guidelines for hosting an Algorave, posted on GitHub as a README file by British musician and academic Alex McLean, cautions users against framing the event as a revolutionary practice. After defining an Algorave as a live music show in which "most performances/acts should involve code-based and algorithmic generation of music and visuals, in a way that makes the code or algorithmic process visible," the guide advises newcomers that "Algorave is not 'the future of dance music', we're just trying things out as part of a much longer history" (McLean 2019). This longer history, as noted by

Roger T. Dean and McLean in the Oxford Handbook of Algorithmic Music, starts as early as the word "algorithm" itself around 900 AD (McLean & Dean, 2018, p. 5). More recently however, and perhaps unavoidably, algorithmic music - now commonly practiced through the digital medium of computers - has collided with the history of the digital algorithms which process data and people on a global scale. This is in part because algorithms, coupled with computational power, have the ability to execute processes far beyond the capacity of an individual person. In this way, algorithms have served to revolutionize the processing of data in the same way that mechanical tools once revolutionized the processing of raw elements. Algorithms undergird our financial systems and media landscapes, creating new paradigms for understanding the world. As such, whether by processing sound, consumer data or social interactions, the digital algorithm has become a notable cultural medium for artists.

The nature of how artists interact with algorithms, however, varies sharply between communities. Some musicians, like those who work within Digital Audio Workstations (or DAWs), use static algorithms as tools to produce a single desired output. Others use the algorithms created by streaming services like Spotify so as to better understand how they can shape their sound to increase audience engagement. Algorithms themselves now play a role in music as an aesthetic in and of themselves, as in the case of live coding: a musical practice which "foregrounds the human authorship of algorithms as the fundamental musical activity at play" (McLean & Dean, 2018, p. 4). This often takes the form of writing algorithms in real-time as a performance takes place, with the written algorithmic language itself being projected to audiences. "Algorithmic" has also become an identity: pop stars such as Lil Miquela and Hatsune Miku use the aesthetic of digital algorithms to generate a mystique surrounding their humanity or lack thereof. I want to suggest that all of these aesthetic relations to algorithms can be understood as differently-programmed constructions of the "musical cyborg:" a figure which combines human creativity and digital algorithms to create sounds and moments that would not otherwise be possible through human production alone. In calling forth the metaphor of the cyborg, I do so according to Donna Haraway's reading of it in the Cyborg Manifesto as "a cybernetic organism, a hybrid of machine and organism, a creature of social reality as well as a creature of fiction." (Haraway, 2016, p. 5) The cyborg musician is both a creative and social construction; its wiring produces sound while also providing insight into the "social reality" that produced it. Some musical cyborgs are a direct byproduct of our increasingly algorithmic society at-large: in a world shaped by black box algorithms developed by mega-corporations like Google and Facebook, musicians are presented with the challenge of navigating a hyper-reality in which information processing is entirely opaque; this results in a cyborg relationship that fosters an antagonism between its respective human and machine halves. Moreover, albeit on a smaller scale, black box musical cyborgs can conceal the humanity in their construction through further black boxing. Lil Miquela, a "digital influencer" based primarily on Instagram (Khan, 2018), bills herself as "not a human being" despite a lack of evidence for her behavior being produced by non-human processes (@lilmiquela, 2018). Human producers, then, are left competing with an invisible network of actors which manifest in these singular, fictional forms. This overall relationship elucidates an antagonistic relationship between humans and algorithms, despite their constant effect on each other.

This seemingly bleak path where algorithms are either the masters or slaves of humanity is not the end: Part of the reason the boundaries between human and computer production have been collapsing is because they have always been ripe for reconstruction. As Donna Haraway notes in the Cyborg Manifesto: "It is not clear who makes and who is made in the relation between human and machine. It is not clear what is mind and what is body in machines that resolve into coding practices[...] we find ourselves to be cyborgs, hybrids, mosaics, chimeras[...] There is no fundamental, ontological separation in our formal knowledge of machine and organism, of technical and organic" (2016, p. 60). Just as a cyborg's construction can represent the world from which it was born, so too can it gestures towards not-yet-realized futures through its fictive components. Because "the ma-

chine is us, our processes, an aspect of our embodiment," as Haraway continues, "We are responsible for boundaries; we are they" (2016, p. 65). This "we", however, has become a case of contested ownership in the 30 years since Haraway published the Cyborg Manifesto between those steering the cyborg network towards optimized profiteering and those seeking to generate life within it. As Mackenzie Wark notes in her retrospective of Haraway's work, "it was prescient of Haraway to notice, and early on, that 'the new communications technologies are fundamental to the eradication of 'public life' for everyone.' The reduction of a wide range of processes, and not just labor, to a thing, or in this case to code, supports a vast extension of private property relations" (Wark 2015). The algorithms that undergird musical cyborg network, with its musicians and instruments, now bear patents and code that make true interoperability a distant dream. Later in this paper, I explore how these components, found installed in artists that are optimized for the streaming economy, make it increasingly difficult to build communal knowledge. The more we accept this mode of algorithmic cyborging, both in and outside of music, the more hard-coded the network will become against any redistribution of power. The guidelines McLean lists for an Algorave construct an alternate model dependent on the transparency and experimentation of algorithms, which in turn proposes a new model for the musical cyborg. Though algorithmic music production takes many forms which vary in human agency, for this paper I want to focus specifically on musicians within the practice of live coding because live coders have the most deliberate and intimate relationship with algorithms among musicians. Moreover, it is a practice in which algorithms are equally visible to both the performer and the audience. Live coders, through this intimate relationship with their tools, offer a vision of what the cyborg musician could look like in an age of digital automation and reproduction, as realized through a few key practices, such as open sourcing and human-computer dialogue. These practices define protocols for discourse in the live coding community. As such, live coders can encode their own desired futures: "The boundary is permeable between tool and myth, instrument and concept, historical systems of social relations and historical anatomies of possible bodies, including objects of knowledge. Indeed, myth and tool mutually constitute each other (Haraway, 2016, p. 33). Thus, the live coding community, through the development of their own agreed-upon tools and practices, proposes a future for all human computer relations.

Redefining the terms of algorithmic culture

In order for a musician to be able to author their own algorithms, the tools for algorithm authorship must be accessible at an individual level. Naturally, open source software has become one of the tenets of live coding: programs like TidalCycles and SuperCollider have dozens of contributors that continue to make changes to the software to this day. This accessibility makes the live coded cyborg both highly visible and mutable.

This creative paradigm stands in stark contrast to the music industry's relationship to algorithms at large. At present moment, the influence of black-box technologies in popular music culture is simultaneously opaque and exploitable. Perhaps most notably within the context of creative development, Spotify offers artists the ability to understand themselves through the data the platform generates so as to appeal to a wider demographic. In a video titled "How to Read Your Data," Spotify executives note that "Data can help you learn things you hadn't even thought about before." To demonstrate this, they feature a testimony from musical group slenderbodies [sic]. "Through the Spotify data we've come to realize that a lot of people consider us as electronic artists... it's surprising to see that considering how organic our sound is... maybe there's, like, a future for us doing some DJ sets on the side rather than just playing full live musician sets with a live band" (Youtube, 2018).

The band came to this conclusion through Spotify's artist tools, which allow artists to see their most popular songs for given times and locations. Though they existed as musicians before Spotify, the data processing nudged slenderbodies towards new behaviors and creative decisions in pursuit of further quantifiable audience engagement.

There is a clear difference in slenderbodies' use of algorithms compared to a live coder's use, though, which is that a live coder's algorithm generates sound while the other generates consumer data. What this example is meant to show, rather than product of the algorithms, is the artist's relationship to algorithmic processes as a whole. The Spotify cyborg is blind to the large-scale processes which inevitably shape self-image, whereas the live coded cyborg actively shapes the algorithms they use to optimize their output on an individual level.

There has been another curious effect of widespread black box algorithm use: a new aesthetic has developed wherein human-produced artworks present as if they were algorithmically produced. This l cyborg, unlike the live coder, has their circuitry concealed in order to generate tension and confusion as to which parts of their construction are human and which are machine. With over 1 million followers on Instagram, Lil Miquela presents as a CGI generated model with feminine features, dressed in the latest high fashion and streetwear. Whether Miquela's actions are those of a robot or those of a human is unclear: the company behind her, however, is made of human actors. "Brud," the creator of Miquela, "[is] a mysterious L.A.-based startup of 'engineers, storytellers, and dreamers' who claim to specialize in artificial intelligence and robotics" (Petrarca, 2018). Here, a black box conceals the efforts of human labor rather than those of an algorithm. That Miquela chooses to present as a robotic being shows that there is social capital at stake in the establishment of creative entities which, authentically or not, traffic in the aesthetics of computation and algorithmic production.

She's released several singles and an entire album, for which the credits are not publicly available. As perhaps alluded to by the title of one of her singles with electronic musician Baauer "Hate Me" (LuckyMe, 2018), the reception to Lil Miquela has been somewhat antagonistic. Nora Khan notes that fans "dissect her photos online and IRL, expressing disgust, disbelief, annoyance, loyalty, and adoration" (Khan, 2018). In her public announcement of being a robot, Lil Miquela acknowledges reading comments such as "You're Fake," "You're CGI" and "Show us your ACTUAL face" (@lilmiquela, 2018). Even large institutions have participated in Miquela's dissection; one video by New York Magazine walks viewers through the process of "Hijacking" Lil Miquela in order to produce the digital figure in-house. The designer in the video superimposes an image of Lil Miquela over a featureless 3D model and subsequently distorts the image to lay over the model like a tight cloth (New York Magazine, 2018). This practice of "hijacking" the cyborg is only possible because of the gates constructed by Miquela's creators. With more visible circuitry, one could construct a similar influencer avatar without the invasive practices demonstrated in the video.

If people want to access the underlying circuitry of a digital art work, why not hand them the keys to the gate instead of forcing them to break in? Live coders also wield digital aesthetics for their projects, but do so in a way which makes the creative processes at play transparent. While it is not necessarily radical or new to publish the code of an art project on GitHub, live coding musician Mike Hodnick, aka Kindohm, uploaded the repository of his RISC Chip release on Conditional Records when it was released in 2017. Through the GitHub repository alone, one could navigate to the software, samples and code used for the project and theoretically recreate the entire album themselves from scratch. Hodnick even went as far as to include a Readme file and inline comments for each of the songs, as well as links to downloads for samples used in the project (Hodnick, 2017).

It took only a few minutes to boot up my copy of TidalCycles and paste in Hodnick's code to run it for myself. From there, I began fiddling with the numeric parameters for his functions, which in some cases noticeably changed the nature of the sound being produced and in other cases made no perceivable change. This moment presented an alternate vision of remix and sampling culture, in which the source was not degraded or otherwise wrested from the hands of its originator. In The Work of Art in the Age of Its Technological Reproducibility, Walter Benjamin posits that "to an ever-increasing degree, the work [of art] reproduced becomes the reproduction of a work designed for reproducibility" (2010, p. 17). With RISC Chip as evidence, it seems as if open-sourced albums are works of art designed

for not just reproducibility, but also mutability.

This careful documentation, however, ends up further blurring the lines between artist and artwork, producer and consumer, and even one work of art from another. At what point did my substitution of variables and functions in the code for RISC Chip become my own creative product, rather than Hodnick's? What is clear is that such art does not belong to any one person or entity. Without the inherent black box of an exported MP3 file in which all instrumentation, production and human labor are condensed into a single file, RISC Chip explicitly eliminates the possibility of constructing what George E. Lewis refers to as the "superperson:" "The better the machines played—and for my money, they do play quite well now—the greater the threat to the mystery, and to an artist's strategic self-fashioning as one of a select band of designated superpeople with powers and abilities far beyond those of mere Mortals" (2018, p. 4). The blurred lines here are not the same which conceal Lil Miquela's true construction; they serve to erase the preconceptions of an artwork's true identity, but in a way that can be rewritten by more than just the initial creator.

To reveal the circuitry of one's creative process, rather than diminish the role of the human, however, recasts them as an essential organic component of a networked sonic machine. As live coders use each others' code, they engage in mechanic processing at a different pace and scale than their digital instruments, but nonetheless contribute to the global computation of algorithmic noise. In this way, live coding is the materialization of the fictional techno cyborg proposed by Kodwo Eshun: "To cyborg yourself you name yourself after a piece of technical equipment, become an energy generator, a channel, a medium for transmitting emotions electric" (1998, p. 106).

Live coders "cyborg" themselves whenever they create GitHub accounts under their artist names, or reveal the code on their screens to show which processes are human and which are machine. This cyborg, in contrast to Lil Miquela, lays its circuits visible for rewiring and recycling by others. As the role of the individual musician is diminished, the contours of a larger, alien structure begin to take shape. Eshun repurposes the words of Norman Mailer to further his definition of cyborging, quoting that the process "'takes the immediate experiences of any man, magnifies the dynamic of his movements, not specifically but abstract so that he is seen as a vector in a network of forces.'" (1998, p. 106) The ideal live coder then, becomes a transparent processing unit which processes the ideas that pass through it into a form that's both simultaneously legible by an individual, yet illegible without the additional processing power of the live coding cyborg's computational organs.

Machines are some of the most important collaborators in a live coded work. To quote Andrew R. Brown, live coding creates "a potentially enhanced sense of agency or otherness brought about by automated processes and their unpredictability or apparent intentionality" (2016, p. 184). The communication between human and computer is amplified through live coding, but has been identified in other artistic practices as well. Hatsune Miku is a Japanese virtual pop idol owned by the company Crypton. She performs via projection for physical audiences and is relevant to discussions of live coding in that, along with being a musical cyborg, her interactions with fans take place through open-source software and licensing, thus allowing for her to be copied and pasted, just as text-based algorithms would, into vastly different contexts.

The Miku cyborg has a healthier relationship to her fans than Lil Miquela, evidenced primarily by said fans accepting her cyborg status as opposed to frustratingly prying at a black box. But Miku's aura is nevertheless dependent on the unifying force of an idolized avatar projected onto stages and laptop screens. The labor of many fans, or "prosumers," funnels directly into a single entity that is, decidedly, a reflection of ourselves. Jelena Guga writes that Miku's "new aura projects the intensities of our own bodies into new holographic humanoid forms."(2015, p. 43)

As Wark notes in her essay on Haraway, cyborgs can also give us insight into "the point of view of the apparatus itself, of the electrons in our circuits, the pharmaecuticals in our bloodstreams, the machines that mesh with our flesh. The machinic enters the frame

not as the good or the bad other, but as an intimate stranger." From this perspective, the Miku cyborg construction is incapable of fully embodying our second halves, our machinic side, in order to restructure the musical cyborg as a whole. The live coding performance in its most simple form - the coder, the computer and the visual of the screen - could be seen as simulating this same projection onto the coder's body itself. But as was the case with a live-streamed performance, the body of the coder need not be visible for their presence to be felt. Instead, the aura of the live coding performance could be seen as projecting the intensities of our cognitive processes on to a new hybrid structure that is as machine as it is human. This object is a network of computers and humans which make up the global processor of live coding, where humans are both the instrument and the producer; nodes in a larger, shapeless network available for restructuring. Rather than clinging to humanoid idolatry, the live coding network distributes its resources more evenly across a constantly reshaped landscape of coders and software, allowing for a more heterogeneous soup of cross-collaboration and creation.

This form of the cyborg was foretold by the fictions realized by the techno producers of Detroit. As Kodwo Eshun notes of Juan Atkin's Model 500 project, "The producer is now the modular input, willingly absorbed into McLuhan's 'medium which processes its users, who are its content.' Tapping into the energy flow of the machine, the Futurist becomes an energy generator" (1998, p. 106). The live coding realization of this is slightly less strong-handed in its posthumanism, as human input is still an essential component of performance. But this compromise, held together by open source code, has the potential to be reconstructed. Perhaps then, the producer-as-instrument as foretold by Eshun, rather than the individual live coder, is the entire network of live coding musicians, hyperlinked by code and wire for the purpose of networked sonic computation.

Conclusions

In this last section, I wanted to outline some of the essential qualities of the live coder as a potential model for the musical cyborg. Their use of legible, personal algorithms and open source software envision a relationship between human and machine which is symbiotic, mutable and transparent. Whether using live coding software such as TidalCycles or SuperCollider, as well as performance specific softwares as in Sonic Biking, the live coder has the potential to reshape the boundaries between their human and machine counterparts. It was difficult pinning down what wasn't a live coded cyborg precisely because the definition by McLean, which simply calls for the human authorship of algorithms to be a primary creative goal, allows for endless constructions and reconstructions. One can be a live coder and still use hardware instruments, collaborate with other humans as well as non-algorithmic entities, so long as there is some degree of processing the world through an algorithmic process.

Even the politics of live coding, due to their open source, mutable nature, have been iterated on as one would an iterate an algorithm. In charting the development of the TOPLAP manifesto, Christopher Haworth explains that "it outlined the conceptual, performative, technological, and philosophical conditions live coders should meet or engage with, performing the dual function of materializing and speculatively positing an idea of authentic live computer music in the form of ten short commandments" (2016, p. 13-14). Over time, though, these commandments changed - notably, demands about what specific languages or tools could be used by live coders disappeared. What remained was a focus on transparency that, according to Haworth:

> "(. . .)conveys an ontological politics of live computer music, one that is positioned against two dominant tendencies in electroacoustic and computer music: one, electroacoustic art music, where fixed-media music is played back in concert halls over loudspeakers; and two, the club-based laptop performance of the early 2000s, where audiences watched performers from behind their laptop screens, and

the performativity of the spectacle was largely taken on faith" (2016, p. 14).

The examples Haworth picks are telling, in that they are instances of performance specific to a time and culture of reproducible music. The TOPLAP manifesto, then, both reacts to and affects the development of computer music, much like a live coder's algorithms are affected by and subsequently affect the live coder themselves. The manifesto reveals that the design ethos of live coding relates not just to the individual live performance, but also at a meta level in terms of how live coders interact with each other and develop cultural standards for these practices. Having these standards laid out in precise terms, available to all but always subject to change, reinforces the open source accessibility, productive interaction and vision of an algorithmic future that live coding generates. Whether the practice generated the politics, or the politics generated the practice, is perhaps irrelevant; both aspects components feed off each other as smaller parts in a larger cyborg system. Nevertheless, there are lines to be drawn between the live coder and other algorithmic cyborgs. Though Lil Miquela, Hatsune Miku, and Spotify's artists exist within an algorithmic culture, the nature of their relationship does not qualify for the continued authorship essential to McLean's definition of live coding because their algorithms have been authored by third parties. In addition, their black box nature leaves them immutable, and thus unable to be reconfigured so as to connect with the human in a symbiotic relationship. These intentional distinctions posit the live coding cyborg as a cultural figure whose construction could be applied beyond the world of music. Haraway might see the live coded cyborg as but one piece of a larger potential restructuring of society; "Taking responsibility for the social relations of science and technology means [...] embracing the skilful task of reconstructing the boundaries of daily life, in partial connection with others, in communication with all of our parts." (2016, p. 67) As algorithms pervade everything from industry to culture to politics, the utopian model of the live coded cyborg can serve as a model for further rewiring in order to generate empathy and symbiosis between human and machine. Though the live coded cyborg exists today, it is still an unattainable fiction for many: I would be remiss not to mention what Haworth discovered to be fundamental fuel in the development of live coding: the institutional support of universities. "Being subsidized by arts and engineering grants," he notes, "the earlier-cited emphasis on novelty and formal experimentation (to the detriment of questions of musical style and genre) emerges as an institutional and economic mediation as much as a performative genealogy" (2018, p. 21). The only places where these generative ideas can survive are those that have yet to be completely recoded by platforms of privatization. As it stands, the reality for many musicians is that black boxing technique or influence is essential to generating profit from their work. But, with Eno as inspiration, planting seeds now can lead to great and unexpected things later. If the barrier between human and machine is now more malleable than ever, we must work towards the betterment and interoperability between machines as much as we do between humans, lest we ignore a vital organ simply because it bears circuitry. The systems which govern current human relations are composed of the same unnatural, ideological code that govern copyright and creative relations. The cyborg is an ideal, and thus it can be an aspirational goal at every level of the stack, from government to culture. But the same can be said of a network recoded from the bottom up; it too can generate systems beyond those which they are conceived in. Building our networks towards the live coding cyborg brings us closer to building a protocol of trust between human and nonhuman, nature and machine, if such a boundary still exists.

Acknowledgments

This paper could not have been written without the support and guidance of Dr. Reginold Royston from the University of Wisconsin, Madison.

References

Bauuer (2019). Hate Me. [Online] LuckyMe. Available at: https://www.youtube.com/watch?v=hYRD0OYSL3w [Accessed 13 Aug. 2019].

Benjamin, W. and Jennings, M. (2010). The Work of Art in the Age of Its Technological Reproducibility. Grey Room, [online] 39, pp.11-38. Available at: http://www.jstor.org/stable/27809424 [Accessed 13 Aug. 2019].

Brown, A.R., 2016. Performing with the other: the relationship of musician and machine in live coding. International Journal of Performance Arts and Digital Media, 12(2), pp.179–186.

Eno, B., Composers As Gardeners. Edge. Available at: https://www.edge.org/conversation/brian_eno-composers-as-gardeners.

Eshun, K. More Brilliant Than The Sun: Adventures in Sonic Fiction. Quartet Books, London, 1998.

Guga, J., 2015. Virtual Idol Hatsune Miku. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering Arts and Technology, 145, pp.36–44.

Haraway, D. (2016). The Cyborg Manifesto. In: Manifestly Haraway. Minneapolis: University of Minnesota Press.

Haworth, C., Technology, Creativity, and the Social in Algorithmic Music. In The Oxford Handbook of Algorithmic Music. Oxford: Oxford University Press. Available at: http://www.oxfordhandbooks.com/view/10.1093/oxfordhb/9780190226992.001.0001/oxfordhb-9780190226992-e-29 [Accessed November 11, 2018].

Hodnick, M. (2019). kindohm/risc-chip. [online] GitHub. Available at: https://github.com/kindohm/risc-chip [Accessed 13 Aug. 2019].

Instagram. (2018). Instagram post by Miquela Apr 19, 2018 at 5:59pm UTC. [online] Available at: https://www.instagram.com/p/BhwuJcmlWh8/?utm_source=ig_embed&utm_campaign=embed_loading_state_control [Accessed 13 Aug. 2019].

Khan, N. (2018). Lil Miquela Shows Us the Future of Fame. [online] Garage. Available at: https://garage.vice.com/en_us/article/wjkbex/lil-miquela-interview [Accessed 13 Aug. 2019].

Lewis, George E. Why Do We Want Our Computers to Improvise?.The Oxford Handbook of Algorithmic Music. : Oxford University Press, February 05, 2018. Oxford Handbooks Online. Date Accessed 11 Nov. 2018 http://www.oxfordhandbooks.com/view/10.1093/oxfordhb/9780190226992.001.0001/oxfordhb-9780190226992-e-29.

Matthews, K. Beyond Me. The Oxford Handbook of Algorithmic Music. : Oxford University Press, February 05, 2018. Oxford Handbooks Online. Date Accessed 11 Nov. 2018 http://www.oxfordhandbooks.com/view/10.1093/oxfordhb/9780190226992.001.0001/oxfordhb-9780190226992-e-34 .

McLean, A. (2019). Algorave/guidelines. [online] GitHub. Available at: https://github.com/Algorave/guidelines/blob/master/README_en.md [Accessed 13 Aug. 2019].

McLean, A. and Dean, R. (2018). The Oxford Handbook of Algorithmic Music. 1st ed. Oxford: Oxford University Press.

Ong, W.J., 1988. Orality and literacy: the technologizing of the word, London ; New York: Routledge.

Petrarca, E. (2018). Lil Miquela's Body Con Job. [online] The Cut. Available at: https://www.thecut.com/2018/05/lil-miquela-digital-avatar-instagram-influencer.html [Accessed 13 Aug. 2019].

Seyfert, R. & Roberge, J., 2016. What Are Algorithmic Cultures? In Algorithmic Cultures: Essays on Meaning, Performance and New Technologies. Routledge, pp. 1–25.

Spotify for Artists (2018), How to Read Your Data. Available at: https://youtu.be/686C0VucG54. (Accessed: October 12 2018)

Striphas, T., 2015. Algorithmic culture. European Journal of Cultural Studies, 18(4-5), pp.395–412. DOI: 10.1177/1367549415577392

Wark, M (2015). Blog-Post for Cyborgs—McKenzie Wark on Donna Haraway's 'Manifesto for Cyborgs' 30 years later. [online] Verso. Available at: https://www.versobooks.com/blogs/2254-blog-post-for-cyborgs-mckenzie-wark-on-donna-haraway-s-manifesto-for-cyborgs-30-years-later [Accessed 3 Dec. 2019]

Yashari, L., 2019. Lil Miquela Is A Virtual Artist Who Is Blurring The Boundaries Of Identity. NYLON. Available at: https://nylon.com/articles/lil-miquela-interview [Accessed October 12, 2018].

YouTube. (2019). How We Hijacked Lil Miquela (and Created our Own CGI Influencer). [online] Available at: https://youtu.be/AEXjFjwq3uU [Accessed 13 Aug. 2019].

Designing for a Pluralist and User-Friendly Live Code Language Ecosystem with Sema

Francisco Bernardo
Emute Lab, School of Music, University of Sussex
f.bernardo@sussex.ac.uk

Chris Kiefer
Emute Lab, School of Music, University of Sussex
c.kiefer@sussex.ac.uk

Thor Magnusson
Emute Lab, School of Music, University of Sussex
t.magnusson@sussex.ac.uk

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

[1]Project MIMIC, `https://mimicproject.com`, accessed: 2019-09-15

Abstract

The growing popularity of the live coding and algorave scenes has inspired incentive and support for accessible, diverse and innovative approaches in expressing art through code. With live coding, the real-time composition of music and other art becomes a performance art by centering on the language of the composition itself, the code. Sema is a new open source system which aims to support user-friendly approaches to language design and machine learning in live coding practice. This paper reports on the latest technical advances and user research with Sema. We provide an overview and design rationale for the early technical implementation of Sema, including technology stack, architecture, user interface, integration of machine learning, and documentation and community resources. We also describe the activities of the MIMIC Artist Summer workshop, a full-week workshop with a group of 12 participants, which we designed and delivered to gather user feedback about the first design iteration of Sema. Findings from our workshop corroborate that language design and machine learning are advanced topics in computer science which may be challenging to users without such a background. Nevertheless, we found that such topics can inform the design of systems which may be both useful and usable to the live coding community.

Keywords: Programming Language Design, Web Live Coding, Machine Learning, User-Centred Design, Coding Ecosystems

Introduction

This paper presents Sema, a new Web-based, open-source, live coding language design and performance playground. Sema is aimed for real-time signal generation and processing, machine listening and machine learning. We are developing it as part of the AHRC-funded project MIMIC[1] ("Musically Intelligent Machines Interacting Creatively"), a three-year AHRC-funded project, run by teams at Goldsmiths Col-

lege, Durham University and the University of Sussex. MIMIC explores how to design and communicate machine learning and machine listening tools in simple and accessible ways for composers, instrument makers and performers. It does this through the design and adoption of new web-based computational tools that leverage on the internet as substrate for a live software ecosystem.

We are interested in the symbiosis of creative machine learning (Grierson et al. 2018) and live coding (Magnusson 2014) approaches to music. Live coding can facilitate pedagogical approaches to computational thinking in the context of creative and artistic practices (Roberts et al. 2016) and STEAM (Yee-King et al. 2017). We seek to understand how well new users from creative areas-i.e. as opposed to more technical backgrounds such as computer science and engineering are able to grasp and apply computational processes of considerable complexity, such as real-time interactive signal processing, machine learning model-building (Bernardo et al. 2017), and language design and grammar specification. We are employing user-centered techniques to leverage the design of new software development tools and evaluate progress through open-ended and creative processes (Bernardo et al., 2018). The paper is structured as follows: this section introduces the paper and presents background research around live coding systems and practices and machine learning. Section 2 presents an overview of the early technical implementation of our new system Sema and elements of our design strategy and rationale. Section 3 describes the MIMIC Artist Summer workshop and the activities for gathering user feedback about the first design iteration of Sema. In Section 4, we discuss the main findings and emerging themes of the workshop. Section 5 concludes with the main takeaways and future work.

Background

Live coding in the arts has existed as an exciting field of activity since the early 2000s, with seeding work and experimentation from previous decades. Live coding practitioners typically engage simultaneously in programming with a domain specific language (DSL) and other modalities, including audio and visual synthesis, instrument design, algorithmic creation, composition and performance (Magnusson, 2014). Early practitioners would typically invent their own systems for musical and other types of performance (e.g. McLean, 2004), often developing systems that were inspirational, humorous and highly effective in real-time performance under stress. With the growing popularity of live coding and algoraves (Armitage 2018), the live coding community appears to be consolidating their practices around a few systems-e.g. SuperCollider (McCartney, 2002), ixi lang (Magnusson, 2011), Gibber (Roberts and Kuchera-Morin, 2012), Sonic Pi and Overtone (Aaron and Blackwell, 2013), TidalCycles (McLean, 2014), ChucK (Wang et al., 2015) and Extempore (Sorensen, 2018). While such systems are excellent examples of established tools for live coding which help to build the live coding community and attract new beginners in, we have lost some of the variation and diversity which existed before.

Wakefield and Roberts (2017) have conducted previous research in language design for live coding on the Web. Their browser-based environment, which leverages a virtual machine, the Parsing Expression Grammar formalism, and an interactive online tutorial and documentation, aimed at enabling users to define custom DSLs syntax and semantic actions. Wakefield and Roberts also described the results deploying the system on an ICLC 2016 workshop, where a few participants were able employ it to develop their own mini-languages.

This paper follows up on this research and on a previous survey on the design of languages and environments for live coding presented at ICLC 2019 (Kiefer and Magnusson 2019) and the "Live Coding Machine Learning" workshop conducted at the ICLC 20191. In this engagement with communities of practitioners, we asked which features they envisioned for future live coding environments and languages that could integrate machine listening and machine learning. The findings indicated a wide space of possibilities, including support for hybrid approaches and multi-paradigm languages (i.e. OOP and functional), flexible, expressive and extensible languages and proto-

typing environments, good quality documentation and examples, as well as a clear and informative error report system. The emphasis that survey respondents placed on potential qualities for a new live coding language, e.g. brevity, simplicity, expressivity, flexibility, adaptability and plurality, pushed us to reconsider the idea of trying to satisfy everyone with one general live coding language design. Instead, we considered designing a new system which could enable and empower users to create and refine their own idiosyncratic languages for musical expression. Considering the history and tradition of live coders building their own systems (Magnusson 2014), this decision would contribute further to a plurality of systems in a field teeming with inventive solutions. We believe that the recent innovation in Web technologies can afford the evolution of an ecosystem of real-time, user-defined live coding languages which combines interactive machine learning, machine listening and audio threads. In this paper we account for the early stage of our design exploration aimed at fulfilling this vision.

SEMA, A Live Coding Language Design Playground

Our previous findings (Bernardo et al., 2019; Kiefer and Magnusson, 2019) inspired us to build a modern Web-based system to support rapid prototyping of live coding languages, which we titled Sema. We are engaged in a design exploration process pursuing the following principles:

- Integrated signal engine – no conceptual split between the language and signal engine. Everything is a signal
- Single sample signal processing – per-sample sound processing to support techniques that use feedback loops, such as physical modelling, reverberation and IIR filtering
- Sample rate transduction – signal processing with one principal sample rate-i.e. the audio rate-is simpler. Different sample rate requirements of dependent objects can be resolved by up-sampling and down-sampling. We use the 'transducer' analogy to enable and accommodate a variety of processes with varying sample rates (video, spectral rate, sensors, ML model inference) within a single engine.
- Minimal abstractions – no high-level abstractions such as buses, synths, nodes, servers, or any language scaffolding in our signal engine. Such abstractions sit within the end-user language design space.
- Striving for an adequate compromise between simplicity and flexibility – support the different user needs in the continuum which comprises beginner and expert live coders.
- Prioritizing usability and learnability – support a smooth and gradual learning curve, ease of use, and straightforward applicability.
- Balancing performance trade-offs with an efficient implementation – considered the constraints above and the performance overhead they may entail, we build upon an efficient implementation to optimize the utility of our system for live coding performance.

In this section, we provide a technical overview of the first design iteration of Sema. Figure 1 below illustrates the general architecture and main elements of our solution.

Machine Learning

Machine learning (ML) has been integrated into Sema as a first-class citizen and core component. ML processes have computationally intensive stages which can undermine the user experience of an interactive application. Previously, we described the critical usability issues of Web-based applications with interactive machine learning

(IML) workflows and audio, where end-users build custom ML models from small, lightweight, user-created data sets (Bernardo et al. 2018). In simple IML implementations, the ML model-training stage can have a thread-hogging behaviour which results in a freeze of the main JS thread. Furthermore, the ML-inference stage competes with the DOM and audio rendering, which may cause audio clicks and dropouts. These processes are therefore better suited for execution on a dedicated thread. This motivated the design of a multi-thread and loosely-coupled architecture for Sema, based on JS workers for ML and AudioWorklet for audio signal processing and rendering (further detailed in Section 2.2). Sema imports the latest version of Tensorflow.js (TFJS) into a JS Web worker and where it is used in the dynamic evaluation of the JS code for bespoke ML pipelines. This enables the user to enact parts of a ML workflow through partial evaluations of TFJS code related to different parts of the ML workflow, such as the set up the training datasets, inputs and outputs, the creation of a model architecture, the definition and configuration the models' hyperparameters, and communication with the user-defined live coding language context. The mechanism is similar to Jupiter Notebooks, where the user can evaluate different code blocks or regions in a non-linear fashion. ML processes in Sema adhere to our transducer concept, in that the sample rates from the event streams they receive from and generate to the live code language context, are converted to and from the sample rate of the audio context.

Signal Engine

The critical usability issues described in the previous section motivated the first step in our design strategy: to implement a signal engine which could run client-side in the browser, in a dedicated thread. Bernardo et al. (2019) provide a more detailed treatment of how we accomplished our innovative design pattern for an WAAPI AudioWorklet-based signal engine and of the performance tests conducted. In a nutshell, we refactored the C++ DSP library Maximilian (Grierson and Kiefer, 2011) and transpiled it into a WebAssembly[2] (WASM) module using Emscripten (Zakai, 2011). Our signal engine loads the Maximilian WASM module into a custom Web Audio API (WAAPI) AudioWorklet processor (AWP) (Choi, 2018). In the audio rendering loop, the AWP (Figure 1) evaluates dynamic DSP code which is injected through an the AudioWorklet asynchronous messaging system. One trade-off of our scalable and high-performance signal engine is that Sema inherits the current WAAPI AudioWorklet limitations and only runs in Chromium-based browsers (e.g. Chrome, Brave, Microsoft Edge, Opera).

Live Code Language Parser

In Sema's first-iteration implementation, which was used on the MIMIC Artist Summer workshop, users were required to employ and manually execute a Nearley.js[3] shell script to generate a new parser for a their user-defined live code language. The Nearley.js toolkit and library implements the Earley algorithm (Earley, 1970). Users needed to define and write a grammar specification in the Backus-Naur Form (BNF) and compile it against Nearley to generate a JS parser. The resulting parser would then be included in Sema's source code and the solution rebuilt. In comparison with other parsing formalisms (e.g. parsing expression grammars), the Earley algorithm supports a broader set of grammars, including ambiguous grammars with left-side recursion. The trade-off for the versatility and flexibility of Nearley[4] is performance. This is shown by comparisons with

[2] WebAssembly, `https://webassembly.org/`, accessed: 2019-09-15

[3] Nearley.js, `http://nearley.js.org/`, accessed: 2019-09-15

[4] Parsing Libraries Benchmark, `https://sap.github.io/chevrotain/performance/`, accessed: 2019-09-18

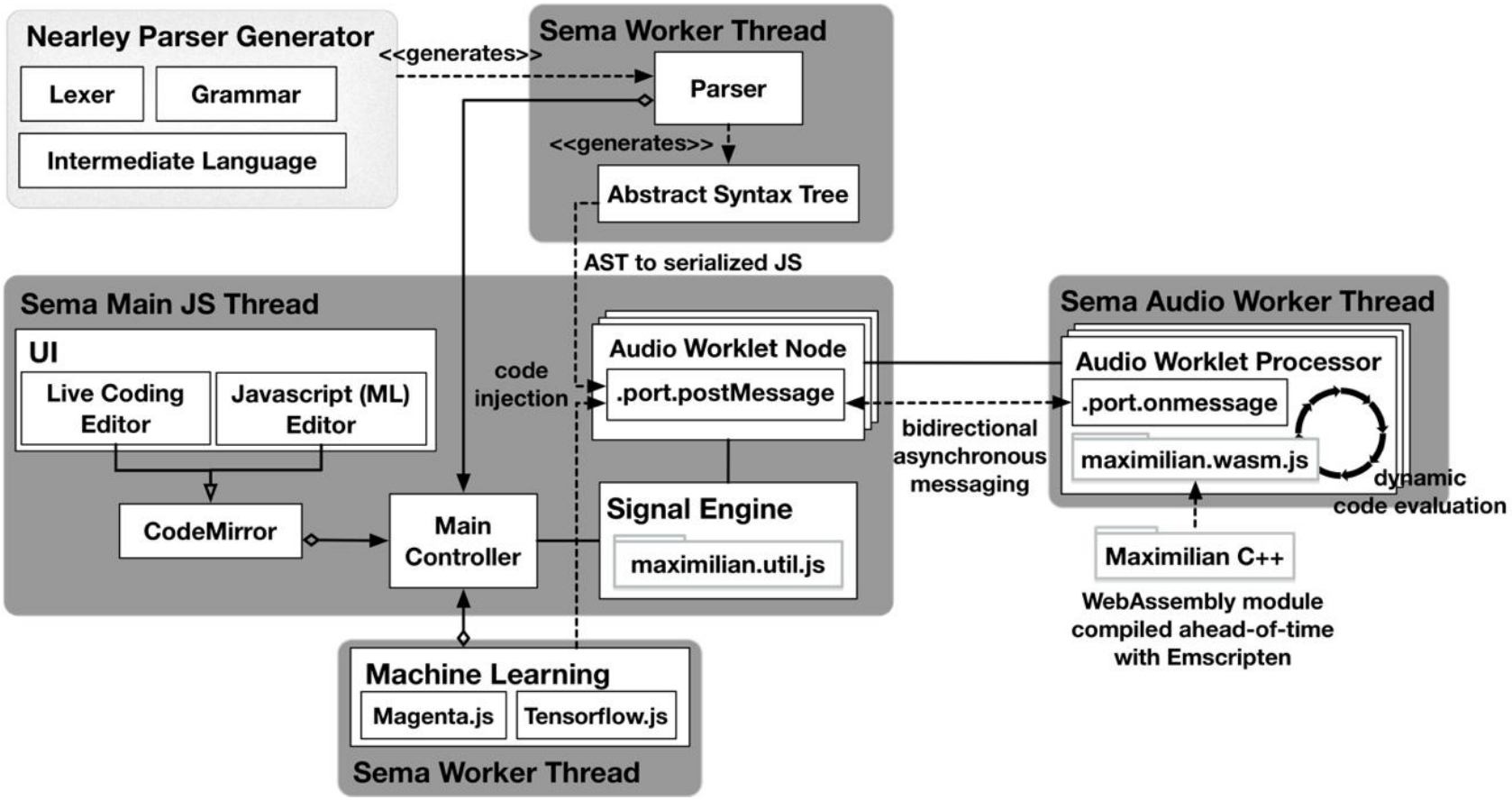


Figure 1: Sema's first-iteration architecture

parsing libraries, DSL and custom-written parser implementations, and other parsing approaches. However, the results from our previous performance tests (Bernardo et al., 2019) show that Nearley, even if slower than other parsers, performs in sub-perceptual time, which is, therefore, adequate for live coding performance.

Graphical User Interface and Code Editors

We experimented with different code editors while considering criteria such as component architecture, community adoption, maintenance and support, and ease of integration with webpack[5]. We opted for CodeMirror[6] to power the two user-facing editor instances in our web-based live coding environment. One CodeMirror instance runs a responsive live coding editor (Figure 2, top, dark background) which provides users with general code editing capacities and manual code evaluation using keyboard shortcuts (CMD-Enter). The other instance runs a second editor (Figure 2, white background) where the user can inspect, customize, or program TFJS-based ML-model pipelines from scratch, as well as define the communication bindings between the user-defined language and the ML worker threads. Sema's first GUI iteration is minimalist and provides a few command buttons, (Figure 2, bottom) for pausing and resuming audio rendering and downloading code from the editors to the local file system. Sema also provides a combo box button with a selection of pre-defined TFJS code for populating the second editor with JS code. This selection consists of a selection of pipelines for building specific ML models built into our system—e.g. simple linear regression (hello-world), two-layer non-linear regression, binary classification, Long-Short-Term-Memory (LSTM) for text generation, echo state networks, and transfer learning with a pre-trained Music Recursive Neural Network (RNN) from Google Magenta[7].

[5]Webpack, `https://webpack.js.org/`, accessed: 2019-09-18

[6]CodeMirror, `https://codemirror.net/`, accessed: 2019-09-18

[7] Google Magenta, `https://magenta.tensorflow.org/`, accessed: 2019-09-18

Workflows

```
@{%
const moo = require("moo");
const lexer = moo.compile({
  click: /click/,
  ws:    {match: /\s+/, lineBreaks: true},
});
%}

Statement -> %click
{% d => [{"@sigOut": {
              '@spawn': {
 '@sigp': {
   '@params':[{ '@num': { value: 1 }},
                  { '@string': 'click' }],
                          '@func': { value: 'loop'}
                  }
      }
        }
  }]
%}
```

Listing 1: Code Example 1

Section 2.3 introduced how the parser for the custom user-defined live code languages is created by the user. Listing 1 shows code of the first Sema tutorial to illustrate a minimal live coding language grammar, written in extended BNF and Sema's intermediate language. A

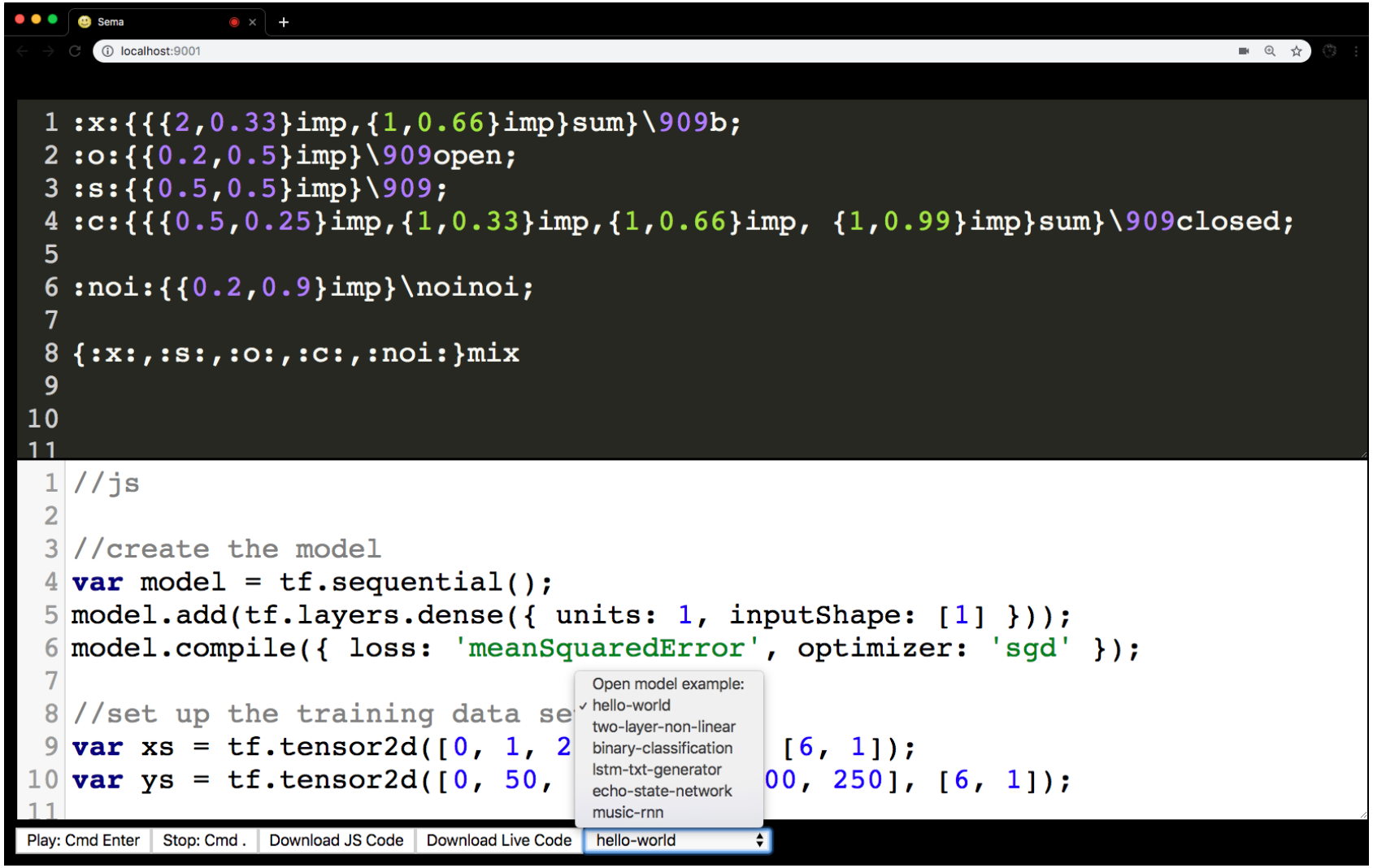


Figure 2: Snapshot of the GUI (pre-workshop version) with default language and machine learning model code

user compiles a file containing this grammar with Nearley to generate a parser for the 1-token language containing the expression "click". The parser is included in Sema source code and used when users evaluate an expression in the live coding editor — i.e. by pressing Cmd-Enter after selecting an expression or placing the cursor on a given line in the editor — and trigger the main workflow in Sema (Figure 1, dashed connectors). The user-evaluated expression is parsed by the Nearley-generated parser. If the expression is valid according to the language formally defined by the BNF grammar specification, the parser outputs an Abstract Syntax Tree (AST). The AST, a tree-like data structure which breaks down the user expression, is serialized to JS expressions that specify which Maximilian DSP objects are used and how they are assembled into DSP functions that will run in the AudioWorklet processor (AWP). These JS expressions are packed into a JS object which is posted through the AudioWorkletNode messaging port (Figure 1) and evaluated dynamically in the AWP audio loop .

Community and Learning Resources

Sema is hosted in a code repository on github.com[8] MIMIC-Sussex organisation, where it is published along other MIT-licensed satellite projects (e.g. osc2sema, sema.github.io). We are developing Sema using a modern web development stack based on node.js[9], webpack, and package managers such as yarn[10] or npm[11]. We are using this stack to leverage on automatic bundling workflows for code, assets, and integrating third-party code from the OSS ecosystem. The documentation for Sema comprises resources that assist the user in the described workflow. Currently, that includes:

- reference and code examples for the default demo language
- intermediate language representation for the signal engine
- reference for the DSP objects and methods of the Maximilan.js API
- data storage and loading functions

Other learning resources are tutorials (Listing 1) embedded in Sema's solution which aim to support a progressive learning curve to grammar editing and language design.

The Mimic Artist Summer Workshop

In this section we describe the elements, activities and results of the MIMIC Artist Summer workshop, which we designed and delivered to gather user feedback about the first design iteration of Sema.

Data Collection

We used an array of data collection methods before, during and after the workshop. We ran a pre-workshop survey to help us understand the background knowledge and skills, motivation, and project proposals of workshop candidates. Data collection during the workshop included participant interactions in the workshop in Slack channel, photos, video and sound recordings of participants' live coding performances with their customized environments, observational notes from the workshop, and notes from the final group discussion. Participants' forks and pull requests during the workshop are also part of the primary data set and publicly available from Sema's github

[8] MIMIC-Sussex/sema, `https://github.com/mimic-sussex/sema`, accessed: 2019-09-18

[9] Node.js, `https://nodejs.org/`, accessed: 2019-09-18

[10] `https://yarnpkg.com/`, accessed: 2019-09-18

[11] `https://www.npmjs.com/` accessed: 2019-09-18

repository. We also ran a post-workshop survey with questions on four main categories: live coding language design, the Sema system, machine learning and community.

Participants and Pre-Workshop Survey

The call for participation for the MIMIC Artist Summer Workshop[12] was released on May 24, 2019. The call addressed artists interested in participating in the workshop and using Sema to build their own live coding languages for live performances and composition using machine learning. The call presented a workshop week-long programme and introduced. We received 16 responses to our pre-workshop survey from which we selected 12 workshop participants (9 males, 3 females). Participants came from the UK (6 participants), Netherlands (2), Norway (1), Germany (1), Sweden (1) and Spain (1). With only one exception, a participant who reported having beginner coding skills, most participants reported being very experienced coders in multiple languages (e.g. JavaScript, Python, C++), including CS graduates, PhD students and teachers of programming. Most participants reported being experienced live coders (e.g. SuperCollider, TidalCycles, ixi lang) also with skills in data-flow languages (e.g. Pure data and Max/MSP). One participant mentioned never having live coded, two participants mentioned not having performed live coding in public. In relation to machine learning skills, the group was more diverse. Half of the group reported having little to no experience in machine learning. Other participants reported having tinkered with a ML toolkit—e.g. Wekinator, GRT, Keras, ml.lib for Max/MSP, and SuperCollider ML tools. Three participants reported having advanced knowledge in ML, two with publications or artworks in the area.

Some of the reasons and motivation that users expressed for attending the workshop included:

- developing their practice in live coding and performing
- understanding the possibilities of machine learning in music
- enhancing their knowledge about machine listening and machine learning
- building new musical instruments and tools
- developing new methods of performance and interacting with audience
- expanding their social network in the machine learning for music and live coding community and meeting like-minded people, learn how to communicate with people in the field
- finding new teaching material
- creative JS coding and exploration

Participants' proposals for projects at pre-workshop stage included:

- clive coding environment with live acoustic audio inputs and algorithmic processing
- building a new instrument with relevant musical parameters for both the performer and the audience
- building tools for song writing
- expanding a personal live coding environment with generative algorithms and recommendations for composing melodic/rhythmic structures
- exploring the possibilities of real-time synthesis using machine learning to create new sounds
- controlling feedback systems
- a live coding system for 3D printing, a rule-based learning system for live coding

[12] MIMIC Artist Summer Workshop, `http://www.emutelab.org/blog/summerworkshop`, accessed: 2019-09-15

Overview of the Workshop Week and Sessions

Workshop sessions took place at the Sussex Humanities Lab (SHL) with the exception for the performance night. The first day of the workshop started with a contextualisation of the workshop within MIMIC research goals and outline of the workshop week activities (Table 4.1). Participants were invited to participate at EMUTE LAB live coding performance night "Musically Intelligent Machines" by the end of the week at the local venue Rose Hill.

Day	Topic
1	Induction session on Language Design with Sema & Induction session on Machine Learning and Sema
2	Counterpoint studio presentation and workshop session
3	Induction session on Machine Listening
4	(Aesth)et(h)ics and creative-AI & Live coding performances with participants' systems at music venue
5	Artist Residence Project showcases + Participant demos & Discussion about Sema (experience, requests, future path)

Table 4.1

The core sessions with Sema were delivered on the first day, one in the morning and the other in the afternoon. The remaining days had blocks of project work interweaved with inspirational and debate sessions, garden lunches, and social activities in Brighton.

The first day workshop sessions with Sema consisted of a practical crash-course and hands-on exploration on language design (Figure 1 a) and an introduction to machine learning and Tensorflow.js. These sessions were preceded by a demonstration of supporting tools, installation and forking of the Sema repository. We introduced Sema's tutorials for language design and grammar specification using extended BNF, Nearley and Sema's intermediate language. We went through simple examples (e.g. Listing 1) to gradually more complex while attempting to get everyone up to speed for them to proceed in autonomous exploration. The machine learning session provided an overview covering ML concepts and terminology, artistic examples and applications, and a walkthrough the Tensorflow.js examples provided with Sema.

Samuel Diggins and Tero Parviainen from Counterpoint[13] studio gave a presentation (Figure 4 a) about their projects with computational design with ML and music participated in the first two days of the workshop. Shelly Knotts from MIMIC-Durham presented some examples of the MMLL library for machine listening (Figure 4 b).

In the EMUTE LAB 4 performance evening, six participants performed along other artists in line-up (Figure 5 a). Three participants performed individually using Sema (Figure 5 a, b, c). Marije Baalman and Henrike Hurtado Mendieta, two artists who were invited to do a 2-week long MIMIC residency, participated in the workshop, performed at the EMUTE lab evening and also presented their work in a session (Figure 6 a and b). In the same morning workshop participant also presented their work in the workshop (e.g. Figure 6 c).

3.4 RESULTS There were two new languages created with Sema during the MIMIC Artist Summer Workshop. One workshop participant created a language titled MAIA[14] and performed with it. The artist-in-residence Marije Baalman created another new language. Two other participants customised Sema and performed with the default language. One of them augmented Sema with 3D graphics and animations, and sonified the machine learning training stage. The other performer developed a probabilistic system that communicated

[13]Counterpoint creative studio, `https://ctpt.co/`, accessed: 2019-09-19

[14]MAIA – Live coding mini-language build upon SEMA, `https://github.com/tmhglnd/maia`, accessed: 2019-09-26

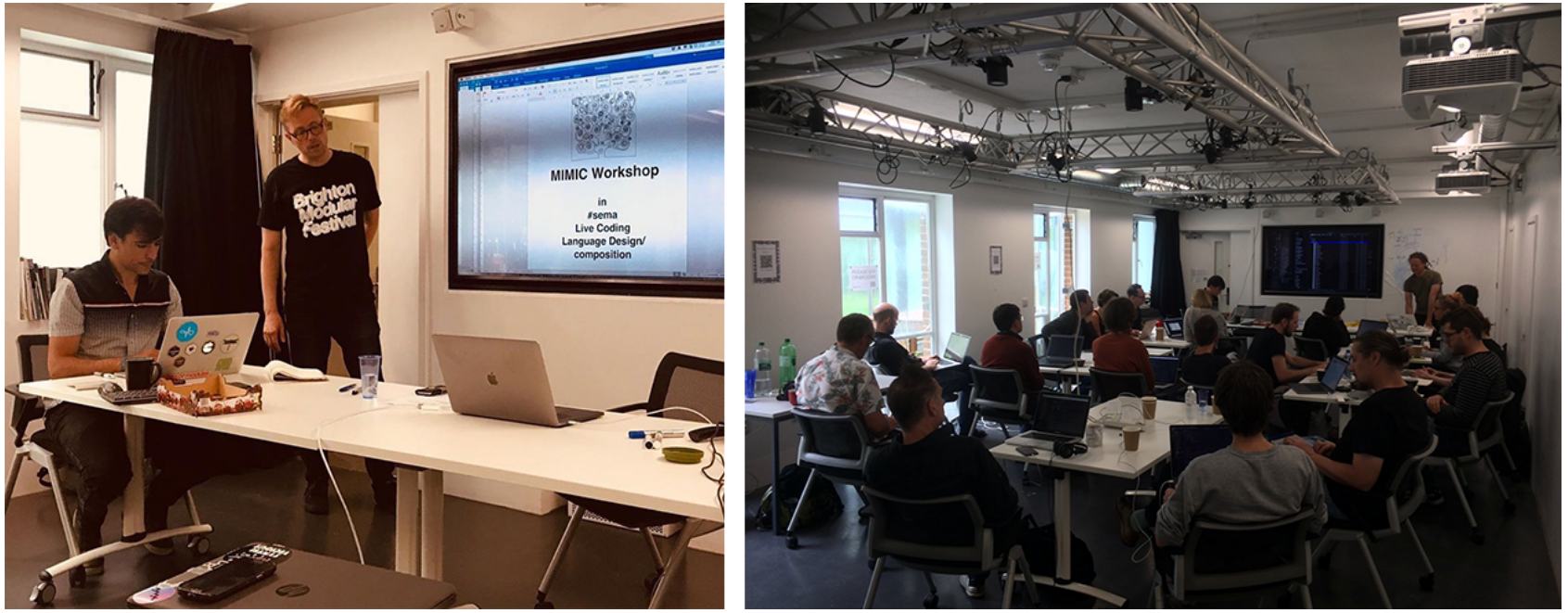


Figure 3: a) The MIMIC Artist Summer workshop opening session and b) the language design induction session with Sema at the Sussex Humanities Lab.

with the live code language to stochastically change tones of the musical sections. Interestingly, one other participant designed a grammar with Nearley playground and used it for generating new text in performance. Other contributions to Sema included extensions to the intermediate language —'amsynth', 'fmsynth', 'oscbank', i.e. three intermediate language constructs corresponding to an AM synthesizer, an FM synthesizer and a bank of oscillators. One contribution consisted of an integration of melody-rnn[15], a pre-trained model from Magenta. There were very meaningful contributions to the documentation. One participant refactored the tutorials with more complete and adequate comments for beginners. Another participant documented the intermediate language. We obtained 12 respondents to our post-workshop survey. We employed NVivo in a qualitative content analysis of participants responses. Participants' names were anonymized and en-

[15]Magenta Models, `https://github.com/tensorflow/magenta/tree/master/magenta/models`, accessed: 2019-09-26

Figure 4: a) Counterpoint presentation on designing with Music and AI, and the b) machine listening session at SHL

coded with labels from the range [MP01-MP12]. The codes employed to classify textual content included: sema, audio engine, machine learning, machine listening, grammars, regexp, documentation, tutorials, workshop, language design, knowledge, experience, programming language paradigms, community, contributing, goals, challenges, suggestions, functionality, understandability, learnability, utility, performance, limitations, negative feedback, positive feedback.

Discussion

In this section, we discuss themes synthesized from the analysis of the main results and primary data.

Signal Engine: Good Audio Quality and Reliability, but more Flexibility and Transparency Required

The audio quality of Sema's signal engine was considered good in general, and in one case, surprisingly solid and reliable for an earlier implementation. Participants who used Sema in their performances also had technically good sounding performances. This confirms the quality and reliability of our signal engine prototype (Bernardo et al., 2019) in particular for use in a live performance setting. These results also show that our strategy for implementing a browser-based signal engine running on its own thread was sound. In one case, our signal engine implementation enabled the sonification of the ma-

Figure 5: a) Poster for the EMUTE LAB 4 night, where b) c) d) three participants performed with Sema L

chine learning model training stage. This shows important improvements over previous technical challenges in integrating machine learning with audio in web-based applications (Grierson et al., 2018), such as the thread-competing behavior of machine learning, as well as audio clicks and drop-outs. There were a few concerns the recurred among participants. One other concern was about the limitations of a browser-based signal engine in terms of processing capacity. This remains an open question for further research with load tests and experimentation in live performance scenarios. One possibility is to explore the trade-offs of an Electron-based build of Sema. There were also concerns was about the investment required to learn yet another audio engine implementation when participants took previous effort with another language—e.g. SuperCollider (McCartney, 2002). In two cases, there were remarks about how the functional 'flavour' and signal-flow-oriented architecture of the audio engine could limit the musical outcomes and artistic expression with Sema. Some suggestions included developing support in Sema for procedural and object-oriented approaches to the intermediate representation. On the other hand, some remarks pinned these limitations to the language design workflow. Such aspects are tied to both the musical affordances of future languages, to the usability of the different components of Sema and to the language-designer experience, all of which require further research and are discussed in the next section.

General Usability, Learning Resources and Documentation Require Improvement

The potential of a language design system for the live coding community was considered useful and appealing. However, several aspects of Sema were considered obscure and very challenging. There were general difficulties with understanding how to use the intermediate language in the grammar specification and how to build the AST; or, understanding how the mechanism of converting the AST with the intermediate language to audio DSP worked. On one hand, these difficulties were related to the specifics of the implementation of Sema,

Figure 6: Final day presentations with a) b) artists-in-residence showcase and c) participants demos

which lacked transparency and abstraction in certain areas and also failed to provide users with adequate documentation. On the other hand, these difficulties are intrinsic to language design, which is considered an advanced topic of computer science. Nevertheless, despite the cumbersomeness of the language design and grammar specification workflow, and of manual and external parser compilation, we observed that people were able to design valid grammars and languages. We got very positive feedback about the Nearley playground for rapid prototyping and exploration of throw-away grammars. The approach of using minimal tutorials to support the gradual exploration of custom-designed languages through adjustments, trial and error, was considered useful and helpful. However, tutorials were considered mostly incomplete and beginner un-friendly, with suggestions for supporting different entry points and skill levels. It is fundamental to improve the usability of Sema and the complex processes that it leverages with better learning resources on conceptual knowledge, system documentation, examples and code comments, as suggested. We found that there is little research on systematic approaches to language design workflows, documentation and learning resources, particularly for live coding languages. There is research sharing a common base of HCI and usability, and focusing on improving API usability (Myers and Stylos, 2016), on design guidelines (Karsai et al., 2009) and usability (Barišić et al., 2013) of DSLs, which we are looking forward to explore with Sema.

Finding the Adequate Approaches, Models, Uses and Data for Machine Learning in Live Coding Practice

There are compelling opportunities for empowering the live coding community with new artistic processes which may arise from the inte-

gration of real-time interactive signal processing and machine learning technologies. Particularly, if such processes are provided in a scalable and accessible environment such as the Web. This was reflected, for instance, by the general appreciation for the knowledge improvement that Sema and our workshop facilitated around different aspects and layers of ML—e.g. from ML concepts and terminology through to specific implementations with Tensorflow.js; also, for how Sema and the workshop attempted to bridge domain-specific concepts of live-coding music performance and interactive audio, in a playful and accessible way.

In our surveys and during discussions, we noticed an overall uncertainty and ambivalence about the utility and use cases of machine learning in live coding. Further research with Sema will prioritize reaching an understanding of which ML-algorithms fulfil a specific live coding use case better. There were interesting workshop outcomes which can help to lead future research with this question. For instance, there were remarks acknowledging that the real-time nature of live coding performance and lack of extensive data sets should be considered. This hints to the future design of Sema to adhere to the live coding constraints pf real-time and small data sets, including curation of machine learning algorithms and probabilistic models (e.g. kNN, Markov Models, RNN), understanding which ML approaches are more suitable for these constraints, for instance, interactive machine learning (Bernardo et al., 2017), and other which may be simultaneously valuable for generation, such as transfer learning (Oore et al., 2018) or reinforcement learning (Jaques et al., 2016).

Design Decisions

The findings and user feedback which we obtained in the workshop helped us to consider, define and prioritize the main development goals for the subsequent iterations of our user-centered design exploration. They are as follows:

- Integrate grammar design and parser compilation in Sema's workflow
- Define and clarify entry points into Sema to improve learning resources and documentation
- Explore the links and adequate abstractions for designing workflows, GUI, AST, intermediate language, machine learning and machine listening
- Explore the adequate ML approaches, models and use cases for live coding
- Design the OSS community strategy and prepare for contributions

Conclusions

In this paper, we presented Sema, a new Web-based OSS system for live coding language design and performance with real-time signal generation and processing, machine listening and machine learning. We contextualised Sema within the research activities of MIMIC, presented the underlying motivation for its development, and presented an overview of the latest technical advances and user research with Sema. We also discussed the main findings and themes which emerged from this work. One group of main findings relate to the quality of the current signal engine implementation and how it enabled to overcome previous challenges in the integration of ML and audio in Web-based applications. Another group findings concerned the challenges and difficulties which users found with the workflows in Sema and general usability issues. A third group of findings concerns the usefulness of Sema as a resource for leveraging pedagogical approaches and learning experiences with ML learning, and which requires further exploration around the provision of the adequate approaches, models and use cases for live coding practice. Sema promises utility and value for the live coding community by filling the gap of systems that support new language design. Whilst the potential to enable users to create their own

languages in a simple web-based playground is strong, Sema needs extensive work to become more usable and welcoming to novices. Future work includes writing better learning resources, making the signal engine more flexible and transparent, and identifying and implementing adequate ML approaches, modes and use cases. We also noticed how people were not be able to grasp the full potential of ML in live coding practice, and this is one of MIMIC's key project objectives. Finally, in this paper, we used workshop findings to motivate and present design and development goals for the next design iteration of Sema. Acknowledgments We would like to thank the participants of the workshop for their participation in workshop and contributions to Sema. We would like to thank Marije Baalman and Enrike Hurtado Mendieta for their inspiring work at the MIMIC residency, participation in the workshop sessions and contributions to Sema. We would like thank Samuel Diggins and Tero Parviainen for their inspiring workshop sessions and contributions to Sema. Finally, we would like to thank Paul McConnell and Alex Peverett for the documentation of the workshop. The research leading to these results has received funding from AHRC through the MIMIC project, ref: AH/R002657/1 `https://gtr.ukri.org/projects?ref=AH/R002657/1`.

References

Aaron, S., 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, Modeling – FARM'13. pp. 35–46. https://doi.org/10.1145/2505341.2505346

Armitage, J., 2018. Spaces To Fail In: Negotiating Gender, Community and Technology in Algorave. Danc. J. Electron. Danc. Music Cult. 10, 31–45.

Barišić, A., Goulão, M., Amaral, V., Barroca, B., 2013. Evaluating the usability of domain-specific languages, in: Software Design and Development: Concepts, Methodologies, Tools, and Applications. pp. 2120–2141. https://doi.org/10.4018/978-1-4666-4301-7.ch098

Bernardo, F., Grierson, M., Fiebrink, R., 2018. User-Centred Design Actions for Lightweight Evaluation of an Interactive Machine Learning Toolkit. J. Sci. Technol. Arts 10, 2. https://doi.org/10.7559/citarj.v10i2.509

Bernardo, F., Kiefer, C., Magnusson, T., 2019. An AudioWorklet-based Signal Engine for a Live Coding Language Ecosystem, in: Web Audio Conference. Trondheim.

Bernardo, F., Zbyszyński, M., Fiebrink, R., Grierson, M., 2017. Interactive Machine Learning for End-User Innovation, in: Proceedings of the Association for Advancement of ArtificiaI Intelligence

Symposium Series: Designing the User Experience of Machine Learning Systems. pp. 369–375.

Choi, H., 2018. AudioWorklet: The future of web audio, in: Web Audio Conference.

Dannenberg, R.B., Mercer, C.W., 1992. Real-Time Software Synthesis on Superscalar Architectures, in: International Computer Music Conference, International. pp. 174–177. https://doi.org/10.2307/3681016

Earley, J., 1970. An efficient context-free parsing algorithm. Commun. ACM 13, 94–102. https://doi.org/10.1145/362007.362035

Grierson, M., Kiefer, C., 2011. Maximilian: An Easy to Use, Cross Platform C++ Toolkit for Interactive Audio and Synthesis Applications, in: Proceedings of The International Computer Music Conference. pp. 276–279.

Grierson, M., Yee-king, M., McCallum, L., Kiefer, C., Zbyszyński, M., 2018. Contemporary Machine Learning for Audio and Music Generation on the Web: Current Challenges and Potential

Solutions, in: Proceedings of The International Computer Music Conference.

Jaques, N., Gu, S., Turner, R.E., Eck, D., 2016. Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning. Thesis 410–420.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S., 2009. Design Guidelines for Domain Specific Languages. Proc. 9th OOPSLA Work. Domain-Specific Model.

Kiefer, C., Magnusson, T., 2019. Live Coding Machine Learning and Machine Listening: A Survey on the Design of Languages and Environments for Live Coding, in: Proceedings of the International Conference on Live Coding. Madrid.

Magnusson, T., 2014. Herding Cats: Observing Live Coding in the Wild. Comput. Music J. 38, 91–101. https://doi.org/10.1162/COMJ

Magnusson, T., 2011. The IXI Lang: A Supercollider Parasite For Live Coding, in: Proceedings of the International Computer Music Conference. pp. 5–8.

McCartney, J., 2002. Rethinking the computer music language: SuperCollider. Comput. Music J. 26, 61–68. https://doi.org/10.1162/014892602320991383

McLean, A., 2014. Making Programming Languages to Dance to: Live Coding with Tidal, in: Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling –
FARM'14. pp. 63–70. https://doi.org/10.1145/2633638.2633647

McLean, A., 2004. Hacking Perl in Nightclubs [WWW Document]. Perl.com.

Myers, B.A., Stylos, J., 2016. Improving API usability. Commun. ACM 59, 62–69. https://doi.org/10.1145/2896587

Oore, S., Simon, I., Dieleman, S., Eck, D., Simonyan, K., 2018. This Time with Feeling: Learning Expressive Musical Performance. Neural Comput. Appl. 1–24. https://doi.org/10.1007/s00521-018-3758-9

Roberts, C., Allison, J., Holmes, D., Taylor, B., Wright, M., Kuchera-Morin, J., 2016. Educational design of live coding environments for the browser. J. Music. Technol. Educ. 9, 95–116. https://doi.org/10.1386/jmte.9.1.95_1

Roberts, C., Kuchera-Morin, J.A., 2012. Gibber: Live coding audio in the browser. ICMC 2012 Non-Cochlear Sound - Proc. Int. Comput. Music Conf. 2012 64–69.

Sorensen, A.C., 2018. Extempore: The design, implementation and application of a cyber-physical programming language. Australian National University.

Wakefield, G., Roberts, C., 2017. A Virtual Machine for Live Coding Language Design. Proc. New Interfaces Music. Expr. 2017 275–278.

Wang, G., Cook, P.R., Salazar, S., 2015. ChucK: A Strongly Timed Computer Music Language. Comput. Music J. 39, 91–101. https://doi.org/10.1162/COMJ

Yee-King, M., Grierson, M., D'Inverno, M., 2017. STEAM WORKS: Student coders experiment more and experimenters gain higher grades, in: IEEE Global Engineering Education Conference, EDUCON. IEEE, pp. 359–366. https://doi.org/10.1109/EDUCON.2017.7942873

Zakai, A., 2011. Emscripten: An LLVM-to-JavaScript Compiler, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. ACM.

Live Coding From Scratch: The Cases of Practice in Mexico City and Barcelona

Hernani Villaseñor-Ramírez
Graduate Music Program, UNAM
hernani.vr@gmail.com

Iván Paz
Computer Science Department BarcelonaTech
ivanpaz@cs.upc.edu

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

Live coding performance starting from a white screen is commonly termed from scratch. In this contribution, we present some thoughts about the possibilities and consequences of this technique, based on our experience as members of the live coding communities of Mexico City and Barcelona. For that, we described from scratch rules and definitions of each community and then, we comment on how this practice began at each place. We finish the text discussing why we consider that live coding from scratch is a technique whose practice can be used as an epistemic tool, through which new creative spaces and limits are explored.

Introduction

Writing code in a blank document against time is a well known live coding practice called from scratch. For example, the one hour practice of Fredrik Olofsson during a hack pact [1], the nine minutes live coding sessions in Mexico City[2] and Barcelona[3] or some networked performances by Cybernetic Orchestra[4]. This practice can be observed either in a concert or as an exploratory process. In this paper, we reflect on the from scratch live coding technique used in the live coding scenes of Mexico City and Barcelona, where the 9 minutes challenge shapes the way to approach it. We also discuss how the from scratch practice developed a sense of meaning of live coding at the beginning of the Mexican scene, how this practice have been developed since then, as well as how the from scratch practice have been adapted and developed in live coding sessions in Barcelona.

[1] It is possible to see here: `https://fredrikolofsson.com/f0blog/node/7`

[2] As you can see in the first videos of the audiocmmcenart Vimeo player: `https://vimeo.com/audiocmmcenart`

[3] `https://toplapbarcelona.hangar.org/index.php/live-coding-sessions/`

[4] For example, `https://youtu.be/vBJeK1wDRJU`

Motivation

This text is written collaboratively; the ideas presented here about the from scratch technique and its practice come from our experience as organizers and participants of the live coding communities of Mexico City and Barcelona. The text covers different periods of time and space: the live coding sessions organized by the Taller de Audio of the Centro Multimedia (CMM) in Mexico City between 2010 and 2014 and the sessions currently organized (since May 2018) by Toplap Barcelona in the Hangar Center for artistic research of Barcelona [5]

From Scratch

We begin by stating two possible definitions which ponder to some extent on the practice in the aforementioned communities and how a sense of meaning about the live coding practice was built from those definitions.

Definition and Meaning

A proposed definition of from scratch is to write code in a blank document against time. From this proposal these questions arise: What kind of aesthetics produce this practice? and What it means to write code as an artistic practice? Back in 2010, in CMM's Mexico City community there was not a clear definition of the term live coding beyond some references to Toplap website and the SuperCollider JITLib Help file. So, from scratch worked as a synonym of live coding, that is to say, in order to consider a computer music or visual performance as live coding it must began from a blank screen. As Dave Griffiths (2012) observed during his participation in the /vivo/ Symposium "the from-scratch technique is considered important in Mexico, with most performances using this creative restriction to great effect", or as Eduardo Ledesma (2015) points, when he analyzes the visual work of Mitzi Olvera, "Mexican-style live coding is considered to be a slightly rougher variety which begins from scratch, not relying on a pre-programmed material, and slowly builds to greater visual and aural complexity as the performance intensifies" (p. 114). The definition of live coding in Centro Multimedia can be observed in its web site, this definition assumed the origin of live coding as sonorous with an extension to the visual: "live coding is the practice of programming in real time, usually linked to computer music and with a paragon to music improvisation, this activity has also extended to video-animation" (Centro Multimedia CNA n.d.). Later, this definition changed assuming that music and visuals are the aesthetic discourses of live coding: "live coding is the practice of writing code live to generate aesthetic discourses as music and visuals in real time" (Centro Multimedia CNA n.d.). The definition used by TopLap_Barcelona is: 'The from scratch technique consists of playing live for 9 minutes starting from an empty screen. In this way, it allows to visualize the high or low level of the languages (try to play from scratch using, for example, Tidal, SuperCollider or Csound), making transparent the tools (classes, functions, data structures, etc.) that allow the live coder to carry out the different musical tasks within the performance. These sessions seek to take advantage of the empty screen restriction and the 9 minutes to explore new possibilities' (TOPLAP_BARCELONA, n.d.). These definitions created common spaces that narrowed the practice through which the communities identified themselves.

Live Coding from Scratch Experiences

In Mexico City

We have discussed some definitions and the construction of meaning of the term from scratch, but How live coding started in Mexico? Live coding sessions in Mexico City began in Taller de Audio del Centro Multimedia CENART at the end of 2010 after a series of artistic and educational events. The first antecedent can be seen in the work of

[5] https://hangar.org

Figure 1: First live coding session, Galería Manuel Felguérez del Centro Multimedia, Mexico City, December 2010. (By Hernani Villaseñor)

multimedia collective mU, formed by Ernesto Romero, Ezequiel Netri and Eduardo Meléndez who made a few live coding concerts during 2006[6]. Later, in July 2009, the concert Prácticas con código vivo[7] by Ezequiel Netri. Education was important to conform the live coding practice in CMM. We can mention the series of SuperCollider courses that took place from 2007 to 2014, which included a discussion of the live coding topic defined by the JITLib and ProxySpace. We can also mention the courses of Processing, Pure Data, VVVV, OpenFrameworks and Fluxus that were strong influenced by a FLOSS discourse. After a few years, regular assistants to these courses had a creative programming background. The Taller de Audio members thought how to invite them to take part in the concerts of the CMM. One proposed activity was to organize computer music concerts that include the live coding practice using SuperCollider ProxySpace as well as other techniques and programs[8]. The first concert came true after a Fluxus workshop, taught by Luis N. Del Angel in 2010, to show the works of the students in public alongside regular students of other courses who figured out how to live coding sound for the first time. The idea was to program from scratch visuals or sound with Fluxus and SuperCollider[9].

After this concert the Taller de Audio continued organizing live coding sessions in the CMM with the same dynamic: around 12 participants from an open call organized in pairs, one participant coding sound while another coding image in a limited period of time established in 9 minutes after a few sessions[10]. This is described by Jessica Rodríguez (2014) who points out the construction of a community during these sessions alongside the ephemeral and changing nature of live coding improvisation, what Carolina Di Próspero (2015) refers as the construction of sociability and subjectivity in practices as live coding. The first year the Taller de Audio organized one session per month in the installations of CMM, and in 2012 started to collaborate with different institutions. That year the First International Symposium /*vivo*/ was organized dedicated to the topic of live coding. From late 2010 to 2014, the Taller de Audio organized around 30 live coding sessions from scratch including CMM and different institutions.

After that period, live coding in Mexico has taken different paths and modes of production oriented mainly to dance music[11] and visual live coding, as well as different ways to organize events, outside and inside official institutions. Also, it is possible to observe a strong commitment to visibility and inclusion[12]., the development of own tools and the reflection of the practice inside academic contexts[13]

[6] `https://toplap.org/wiki/ToplapEvents`

[7] `http://cmm.cenart.gob.mx/cartelera/2009/julio.html`, scroll down to Sinescenia

[8] This was described in the talk 9 minutes from scratch: a story of live coding in Mexico by Hernani Villaseñor and Alexandra Cárdenas, in the Symposium of Live.Code.Festival, Karlsruhe 2013 (Hutchins 2013).

[9] `http://cmm.cenart.gob.mx/cartelera/2010/diciembre.html#livecoding`

[10] `http://cmm.cenart.gob.mx/cartelera/2011/enero.html#livecoding`

[11] For example, Ocelotl et al. (2018) mention that the band RGGTRN "has evolved into a collective that engages in algorithmic dance music and audiovisual improvisation informed by the Latinx context that revolves around its members"

[12] For instance, `https://hbrdsyqmrs.wordpress.com/`

[13] For instance, `https://piranhalab.github.io/comunidad.html`

Figure 2: Arinoise (left) and Mitzi Olvera (right), 18th Live Coding Session in Sciences Faculty UNAM, Mexico City, August 2012. (By Hernani Villasñeor)

From Scratch Rules in the Centro Multimedia Live Coding Session

- Write code from scratch in order to generate sound or image in the preferred programming language.
- In nine minutes.
- Stop when the audience applauds.

From Scratch in Barcelona

Ever since May of 2018, from scratch sessions have been carried out once a month by the Barcelona TOPLAP community[14]. By the time the first session was organized, Barcelona has had three Algoraves, live coding was known, but the from scratch technique was not a common and well defined practice. The idea was to start the TOPLAP node with a live coding session using the same rules of the Mexico City sessions, to emphasize the writing of the code in real time (Barcelona already had a tradition with languages such as SuperCollider or PD. However, performances were conducted mostly using prepared systems). In subsequent events, as in the case of Mexico, modifying pre-written code was also allowed. Nonetheless, the community continuously practices from scratch performances in the sessions. Proof of this are the two from scratch sessions held in January

of 2019 during the /* VIU */ festival[15], a moment that also brought together the live coding communities of Mexico City and Barcelona. Unlike the Mexican scene of the late 2010, that with some exceptions was quite homogeneous in the tools, since the first session the polyglotism of the Barcelona scene was clear (SuperCollider, Tidal, Sonic pi, Max MSP, and C++ were the languages used)[16]. This immediately brought to the discussion the next questions: How the differences in design and abstraction level of the programming language, impact its sonority, complexity (in terms of the number of lines needed) and readability? Also, what kind of sonority and form do the performances have, according with the tool, when using only the original resources that the program has? Since the first session, the from scratch restriction has been used to explore these questions, up to the limit of using bash, and C++ to emphasize the possibilities

[14]TOPLAP Barcelona started as a resident collective in Hangar on May 2018 https://hangar.org/en/residents/collective-residents/toplap/

[15]`https://toplapbarcelona.hangar.org/index.php/viu-en/`

[16]You can see a from scratch session in Barcelona: `https://youtu.be/IJPKeKZ6bv0`

and aesthetics of working with low level languages[17], or packing the code into classes to increase the readability and compactness.

From Scratch Rules in the Hangar Live Coding Session

For the public:

1. Live coding sessions are gatherings to practice with public, either from scratch or re-writing/modifying written code.

2. All attendees should applaud at the end of 9 minutes (remember that this is just for fun)

From scratch rules for the coders:

Each live coder (audio or visuals).

1. Start with the blank screen

2. You have 9 minutes to play

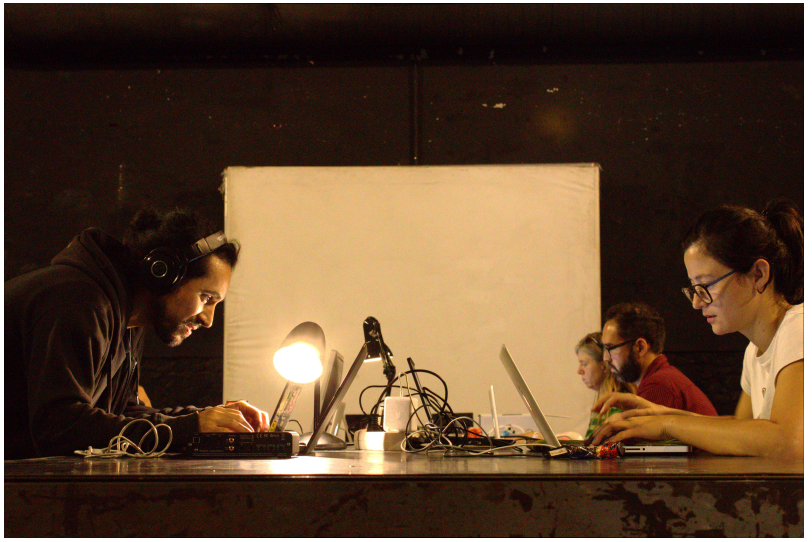


Figure 4: Iván Paz, Lina Bautista, Gabriel Millán ang Alicia Champlin, live coding form scratch session, Hangar, Mayo 2019 (By Silvia Miranda Arana)

Ideas Beyond From Scratch

Consequences for the Performance

In a from scratch performance the live coder demonstrates writing skills and the unfolding of the piece, as it is constructed in real time, gives insight to the composer's mind. The public, who does not know this practice, tries to understand what is happening during the events. In this sense, very clear and simple explanations are required. On this point, Alexandra Cárdenas (EFEAV 2019, 00:01:27) comments that

[17] In a personal conversation with Niklas Reppel he said that he decided to use C++ in the first from scratch session of Barcelona to emphasize the idea of "not using anything prepared" after that, in a session during the III International Conference on Live Coding, he received comments saying that he had started with code on the screen (using his live coding language Megra)

Figure 3: Lina Bautista and Citlali Hernández live coding from scratch, Sala Ricson, Hangar Barcelona, Marzo 2019 (By Iván Paz)

it is not necessary to know how to play the piano to enjoy a concert, since certain structures are familiar to the public, for example where the high and low pitches are located. Little by little, this coarse structures are being created as the public becomes familiar with the code. The error that happens during the code writing has been widely discussed (Collins et al. 2003). Besides coding errors, some failures appear during from scratch sessions due to constant changes of computers; sometimes no soundcheck or projector test are carried out, then the input of the projector is different, the sound card is not configured, the screen size is modified or sound does not come. All types of errors, as they are part of the performance, are embraced by the live coders and integrated into the aesthetics.

Tools Implications

Live coding from scratch shows the code to the public with the intention of sharing, being open, and to give access to the performers's mind (TOPLAP n.d.). Restricting from scratch practice to "vanilla" distributions of the programming languages (using the programs as they are downloaded) as some people suggest, also guarantees that the tools used during the performance can be studied/used by anyone that downloads the program. In this way the idea of accessibility is reinforced. But this maybe doesn't apply for audience, such as Herrera Machuaca et al. (2016) observe, they say that after years of performing in live coding sessions and concerts as Colectivo Radiador "we began to notice that only the programmers in our audiences received the performances well. Programmers were interested in the code we projected on screen, but the rest of the public was less interested" (p.118). Their solution was to translate their live coding language into one more related to audience understanding[18]. So the open and access to code can be questioned in relation to the knowledge of programming.

Poetic Implications

As we mentioned, the temporary restriction during from scratch performance pushes the performer to find more efficient code structures and syntax (e.g. concise, compact, succinct), to, with the least amount of characters, achieve to develop a complete piece. Time constraints and starting with a blank screen make the technique from scratch ideal for visualizing the role of the "level" (e.g. high or low) of the programming language in performance. This, most of the times, imprints a specific sonority in the resulting performances. For example, the first live coding sessions in CMM Mexico were characterized to figure out how to perform in short time writing code; sometimes in nine minutes a SynthDef was written with no sound at the end of the time, so the question of What means to write code in front of an audience arise at some point? Other times participants used strategies such as transcribe, memorize, or write minimal lines of code. These approaches pushed participants to write fast or search for functions of the programs that do a lot of image and sound with few code; in reference to Marije Baalman (2015) they were adapting mind and body to the code as "code embodied by the human" (p.36).

Machine Learning From Scratch: On-The-Fly Training

During the MIMIC artistic summer workshop held at the University of Sussex during July of 2019[19], the current approaches for using artificial intelligence (machine learning algorithms) in Live Coding were discussed. One of the current discussions on this issue is whether

[18] This can be observed in the next video: `https://youtu.be/m03pay7A44k`

[19] `http://www.emutelab.org/blog/summerworkshop`

[20] Including the data collection of the training set

the training process of the models should be done offline or on-the-fly[20]. In the second case, the model training can be included as part of the performance, however the time required depends on the size of the training data set. One of the performances presented at the end of the workshop was carried out by Marije Baalman, who trained the model on stage, so that the training process became part of the performance. In contrast to some other performances that used pre-trained models (from which model instances were taken during the performance) this from-scratch-training had something that remained the live coding from scratch performances. In this case, the algorithm response, the data being collected, etc. can be inferred by the public while the time restriction shapes the performance. From this perspective, the idea that starting from scratch visualizes the limits and functioning of what is being written can be extended to the new live coding tools.

The Term of Live Coding as Creative Constraint

On-the-fly writing of code through interactive programming is a way of approaching tasks where no clear specification of the problem to be solved can be given in advance, e.g. when the exact form of the solution is not known in advance or when only a draft of the idea is available; problems in which we only find the exact form of the solution when it is found. In these situations (which are not unusual in artistic research), the formal structures of the programming language provide the environment to explore, in real time, the different possibilities to conduct the search within the space. Even when we already tested certain structures, when performing from scratch, the restrictions imposed keep us practicing this type of search. The from scratch practice can be understood as an exploration tool (in specific conditions) that allows to find the first possibilities of a language, visualize the available abstraction levels, and glimpse different possible paths. Then, although the languages change, it is still a valid approach. As it could be the case of training a machine learning model in real time (as discussed in Section 5.4).

Conclusions

Live coding from scratch is a creative technique that emphasizes (visualizes) real-time code writing. When a temporary restriction is added, the abstraction levels contained in the programming language (is it low or high level), the design of the program (for example the cycles in the case of Tidal), and the virtue of the live coder to use these elements to create a composition or moving images are visualized. The sonority (timbre, structure, form) or the visual part of the performances exhibits the immediate use of the sound and image generators, sequencers, task definitions, etc. The tools that the live coder has more "at hand". In live coding, the gesture is expressed through the code, as it is the code (its writing and its execution) that conducts the performance. This is emphasized in a from scratch performance since the public can more easily follow the writing. The challenge engages the performers since, besides the performance, it is also a way to explore. It is a way to know the limits of the software, as well as its possibilities and response in specific conditions. That is, how far it can go, or what kind of pieces can be made with, for example, a vanilla version of SuperCollider in nine minutes. What audio generators work for tasks such as creating sound textures, synthesizing kicks, producing melodies, patterns, etc.? From this perspective, the nine minutes challenge is a way to find new things, to explore the limits of the software, to find ideas for other compositions. Zlatko Baracskai once said that if you find it difficult to eat pizza with your hands, you could try it without hands and then, you will see that when using only your hands again, it seems the easiest thing in the world. This analogy illustrates some of the sensations of the challenge. In the context of the scenes and communities of Mexico City and Barcelona, live coding from scratch have been practiced at different moments, from where different things can be observed. For example, in Barcelona, having C++ and Tidal Cycles in the same session clearly visualizes the levels of abduction in programming. In the case of the Mexican scene having more homogeneously distributed tools, such as SuperCollider and Fluxus, allowed the challenge to be a

shared exercise from which to learn from others (for example, the use of ProxySpace), thus generating a sense of meaning and sociability of the live coding practice. In this regard, the definition of a practice and the construction of a community through a technique.

Acknowledgments

There are many people involved in the organization and conformation of both scenes and communities. We want to thank all who participated in live coding sessions, the members of Taller de Audio including the people of social service, and all in CMM CENART who made design, documentation, dissemination, tech support and space security, also the institutions that hosted the live coding sessions. In the case of Hangar, we thank the whole live coding community involved and the support of the Hangar team. Hernani Villaseñor acknowledges the support provided by CONACyT through Music Master and Doctoral Program UNAM.

References

Baalman, M. (2015) 'Embodiment of code', in McLean, A., Magnusson, T., Ng, K., Knotts, S. and Armitage, J. International Conference on Live Coding 2015, Leeds, UK, 13-15 July, Leeds: ICSRiM, 35-40.

Centro Multimedia CNA (n.d.) Actividades, available: `http://cmm.cenart.gob.mx/cartelera/actividades.html` [accessed: 25 October 2019]

Collins, N., McLean, A., Rohrhuber, J., & Ward, A. (2003) 'Live coding in laptop performance', Organised sound, 8(3), 321-330.

EFEAV (2019) 'Escribir código en directo puede ser todo un espectáculo' [video], available: `https://youtu.be/c2I_v44ndUc` [accessed 04 Dic 2019].

Di Próspero, C. (2015) 'Live coding. Arte computacional en proceso', Contenido. Arte, Cultura y Ciencias Sociales, 5(2015), 44–62.

Griffiths, D. (2012) 'Mexican livecoding style', dave's blog of art and programming, 21 November, available: `http://www.pawfal.org/dave/blog/2012/11/mexican-livecoding-style/` [accessed: 25 September 2019]

Herrera Machuca, M., Lobato Cardoso, J. A., Torres Cerro, J. A. and Lomelí Bravo, F. J. (2016) 'Live coding for all: three creative approaches to live coding for non programmers', International Journal of Performance Arts and Digital Media, 12(2), 187-194.

Hutchins, Ch. C. (2013) 'Live Coding in Mexico', Les said, the better, 20 April, available: `http://celesteh.blogspot.com/2013/04/live-coding-in-mexico.html` [accessed: 25 September 2019]

Ledesma, E. (2015) 'The Poetics and Politics of Computer Code in Latin America', Revista de Estudios Hispánicos, 49, 91-120.

Ocelotl, E., Del Angel, L. N. and Teixido, M. (2018) 'Saboritmico: A Report from the Dance Floor in Mexico', Dancecult Journal of Electronic Dance Music Culture, 10(1), available: `https://dj.dancecult.net/index.php/dancecult/article/view/1066/962` [accessed: 25 September 2019]

Rodríguez Cabrera, A. J. (2014) El código de programación en la era Post-media: Análisis del proceso creativo en las producciones estéticas visuales y/o sonoras, thesis (B.A.), Universidad Michoacana de San Nicolás de Hidalgo.

TOPLAP_BARCELONA (n.d.) Live Coding Sessions, available: `https://toplapbarcelona.hangar.org/index.php/live-coding-sessions/` [accessed: 16 September 2019]

TOPLAP (n.d.) ManifestoDraft, available: `https://toplap.org/wiki/ManifestoDraft` [accessed 04 Dic 2019, 18:08]

Disabled Approaches to Live Coding, Cripping the Code

Amble Skuse
amble.skuse@plymouth.ac.uk

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

The project takes a Universal Design approach to exploring the possibility of creating a software platform to facilitate a Networked Ensemble for Disabled musicians. In accordance with the Nothing About Us Without Us (Charlton, 1998) principle I worked with a group of 15 professional musicians who are also disabled. The group gave interviews as to their perspectives and needs around networked music practices and this data was then analysed to look at how live coding software design could be developed to make it more accessible. We also identified key messages for the wider design of digital musical instrument makers, live coders and performers to improve practice around working with and for disabled musicians.

Definition of terms

In this paper I will use the terms Disabled and Non-Disabled in line with the social model of describing Disability (Oliver 1990). The social model of disability describes how disabled people are not disabled by our bodies, but by a society which creates environments in which we cannot function. This is the process of society and structures, actively disabling people. For example, the person who uses a wheelchair is disabled by the decision not to include lifts and ramps to a building. It is the design of the environment which disables them. Following this theory, a person with a fatigue condition is disabled from taking part in a project because of the long days scheduled or the distance travelled. The design of the project has disabled the person. A Non-Disabled person is one for whom the structure and design of society broadly works. I will also use the term d/Deaf which is a term which refers to two differing communities and political positions on deafness. Small d refers to those who have partial hearing, or use English as their first language, and capital D deaf referring to those who use sign language and consider themselves "culturally deaf" that is, to live in a more visually orientated culture, outside of spoken language (Woodward,1972).

I use the term Disabled Musicians to mean disabled people who have a professional or semi professional music practice, as distinct to using music as a therapeutic or community building practice with disabled people. It may be that some of our findings may be suitable and transferrable to the therapeutic and community music research environment.

Why we did it

Dreams and Possibilities - a manifesto

- We would like to see action from the Live Coding community to listen to the needs and concerns of disabled musicians around the making of live coding musics and platforms.
- We would like to see ensembles, rehearsals, conferences and performances which respond to the requirements of disabled people.
- We would also like to see the community provide learning opportunities for disabled musicians to learn live coding and software development.
- We would like to see disabled access built into the core of music technology software.

8

Disabled Access and Universal Design - Why disabled people must be at the heart of developing technology

The extension of the social model approach is the "nothing about us without us" concept which holds that those for whom a service, system or environment is designed, must have a contributing say as to its design. By incorporating diverse voices in the design of something, we can more reasonably design something which fits those it will serve. This work is extrapolated by Jutta Treviranus in her work on Universal Design.

The first principle of Universal Design, is Equitable Use, and is broken down into the following four categories.

1. "Provide the same means of use for all users: identical whenever possible; equivalent when not.
2. Avoid segregating or stigmatizing any users.
3. Provisions for privacy, security, and safety should be equally available to all users.
4. Make the design appealing to all users." (Shein, Treviranus, Brownlow, Milner, & Parnes, 1992).

The next logical step to Universal Design is the concept that everything that is made, should be made by a wide range of people for the full range of people. In this way we can ensure that we turn made for, into made by. By incorporating this into our design we can avoid cultural appropriation, perpetuating stereotypes, assumptions, supremacist perspectives, and oppression. For example, a recent development of gloves which interpret sign language into spoken text has met with a negative reaction from some d/Deaf communities (Errard, 2019). Academics from the d/Deaf community roundly criticized the gloves, which claim to turn sign language into spoken language, for a number of reasons. Firstly, that the gloves only interpreted alphabetic spelling, which is a tiny part of sign language, secondly that the hand and finger movements are just a small part of signing, which uses facial movements, mouth movements, and eye movements as a part of the language. Thirdly, philosophically, the gloves reinforce a notion that d/Deaf people's communication is a thing which needs to be 'solved' for non signers, and puts the responsibility for that back onto the signer, to buy, train and use the gloves for the benefit of the non-signer. This was seen as cultural appropriation and a colonialist approach to technology and disability. If d/Deaf people had been included in the conceptualisation of the project, and if the makers had read and understood disability culture and politics, this

could have been averted. Something more useful and equitable could have been created. From a development point of view then, it is vital that approaches to disabled technology come from the community, with an equitable approach, rather than to fix a problem perceived by the non-disabled community. One example of this kind of thinking comes from The Inclusive Design and Research Centre's project Co-designing Inclusive Cities which

> '...offers citizens a way to actively participate in the iterative design and growth of communities that meet their needs. Including the most unique and diverse needs—the "edges"—in the co-design process is an effective strategy to ensure our design stretches and responds to a broader range of needs. If we reach the edge, the design will also work better for the centre and will be more flexible and generous.' (IDRC 2019)

It is this co-design which we can import to music technology, and specifically Live Coding in order to make it more flexible, robust and inclusive.

Why Live Coding?

In this paper we specifically focus on live coding as a way of exploring disabled networked performance, building on a recent strand of live coding research that considers diversity from various angles (Skuse & Knotts, 2018; Ocelotl, N. del Angel, & Teixido, 2018; Armitage, 2018; N. del Angel, Teixido, Ocelotl, Cotrina, & Ogborn, 2019). By making this work about live coding, we have the opportunity to design things from scratch. Coding is Design. When we code something we make it new from scratch, and we can make it to suit ourselves. This allows us multiple levels of engagement with the Universal Design movement, which can support disabled people in making their own technology, and having access to the skills to adapt it as they see fit. This shift in power from the "made for" to the "maker" means that Disabled coders can be at the forefront of countering (potentially unintentional) ableist music making.

There are also a number of key connections between the concerns of live coding research and those of disabled studies. Firstly a focus on open source software potentially responds to a major concern for disabled artists: the costs of software purchase, maintenance and deployment to diverse computing platforms. Secondly, live coding can often be accomplished without a requirement for specific hardware such as MIDI control surfaces, whose buttons, faders and pots can cause difficulty for disabled people with motor control issues or upper limb difference. Thirdly, and conversely, live coding's enthusiastic embrace of alternative and diverse programming interfaces connects readily with the use of additional hardware, such as eyegaze and headmouse controllers, that might be required by people with particular needs. Fourthly, live coding often takes the form of networked performances using a shared media surface. Moreover, when such networked systems transmit code rather than sound, bandwidth requirements are reduced substantially, making them useful to home-based Disabled artists, for whom the majority of their work may then be performed over the network. Finally, the work of the Live Coding community in breaking down expectations about performance and audience (for example watching someone code is a relatively new progression in terms of gig expectation) can create an opportunity for us to rethink how performance can be made more Disabled. Networks and online presence continue to be a powerful and revolutionary tool for the Disabled community (Pearson & Trevisan,2015). There are obviously important concerns around surveillance, facial recognition technology, racist algorithms, and oppressive regimes (Umoja Noble, 2018; Eubanks, 2018). However, on a very simple level, Disabled people with limited mobility, or with limited income are able to meet, support each other and undertake activism online (Berghs et. al., 2019). We are still subject to these oppressive technologies of course, alongside trolling, hate speech and unfair representations in social and traditional media. But, we are also liberated from bodies and minds which do not easily meet and communicate in person. In this

way, networked activity offers a means of collaborating specifically useful to home based disabled people. In this case the question of the impact of networks can come down to ownership, authorship, and agency—whose algorithms are these? Who do they represent (and fail to represent)? How do users have control over changing how something works, understanding how their data is used, and governing its use and aggregation. So again, by shifting "made for" to "made by" we have the opportunity to address power imbalances, structural oppressions and bias.

What we did

In order to reflect the principles of Universal Design, and ensuring that the users had as much influence on the process as possible, we interviewed disabled musicians to find out their requirements and then attempted to design systems according to our findings. I began by contacting home-based disabled musicians worldwide using social media (disabled twitter, Facebook, Instagram). I conducted 15 online interviews with home-based disabled musicians to explore their approach to making music, their requirements from a music making applications (How should it be laid out? What platform works best? What controller adaptations may be useful? etc.) their requirements for learning and work-shopping ideas (How long can they concentrate for? How long can they control the software for? How best to communicate during the workshops?), and their requirements for performance (Can they do real time performances consistently? Would they need to pre-record aspects of the work? How long can they perform for? etc).

The interview process was granted Ethical Agreement by the University of Plymouth Arts and Humanities Research Ethics Committee, and all data is held in compliance with GDPR, the UK's Data Protection Act (2018).

The Interviewees

The interviewees represented a wide range of disabled musicians, and also included two non-disabled academics who work in the fields of adaptive music technology and live coding (although one of these has lived experience of being disabled, they do not currently identify as disabled). The group of disabled musicians included a range of backgrounds and experiences, from acoustic performers who had limited experience of working with music technology to those who have performed as live coders. The interviewees also represented a range of impairments, racial identities, genders and sexualities. This was explicitly chosen as a strategy as to not foreground a white male perspective. As the pool of 'disabled musicians who are interested in networked performances' is a particularly small group of people, we were unable to be as diverse as we had hoped. However our group were reasonably mixed. Disabled identities included limb difference, mobility issues, hypermobility, stroke recovery, d/Deaf, M.E., Diabetes, Schizophrenia, Autism, and ADHD. 2 Interviewees identified as black British, 2 as British Asian and the rest as White. 9 interviewees were male and 6 female.

What we learned

Findings

Findings from the interviews show a wide range of preferences, requirements and adaptations. Those requirements broadly fell into four categories, physical adaptations, communication preferences (partially sighted / d/Deaf etc), fatigue requirements, social interaction. We mention this because physical adaptations and communication preferences are often foregrounded in disabled access projects, whilst fatigue requirements and ways of structuring activities and social interaction are less commonly considered. The paragraphs below show the different responses to the questions which were put to the interviewees.

Hardware

There was a roughly even spread between those who favoured the use of a computer and those who preferred to use a tablet or phone, with some saying that they would often switch between the two. Preferences for using a computer included the larger screen size being easier to view, the keypad being easier to control than a touch screen, that there is more control over connecting different applications, and that it is easier to plug in adaptive hardware such as eyegaze or Headmouse. Preferences for tablets and phones included being easier and lighter to transport, that the touchscreen is less impactful on hands and wrists, that the touchscreen allows interaction for those with less motorcontrol, and that they are generally cheaper to purchase than a computer.

Design and Display

The majority of interviewees requested that an electronic music environment's colour scheme and display be customisable, with users able to choose their own background colour, font type, size and colour. In addition to this, interviewees requested that the different elements of a user interface or display are placed in separate windows which can be moved around the screen, resized, and zoomed in, or out. This reflects a networked community perspective where each individual is able to define the circumstances in which they work best. By providing customizable design and display we give each person (not just disabled people) the ability to create their own workspace. One of the interviewees controls his computer with a Headmouse, using his tongue to move the curser and blowing into a tube to click. The Headmouse has no double click or right click function, and in order to 'ctrl click' he has to request his assistant to hold down the ctrl button. For this user it is important that there are no double click or right click actions. In order to type he has to select a screen alphabet and click on each letter, this is a slow a laborious process. For this user, typing proves problematic and laborious.

Barriers to engaging with new music making software

One of the key barriers to engaging with music technology was cost, a place where the open source live coding community have already made significant inroads. There were also concerns around whether using computer coding to make music is a 'real' musical activity, and whether it would be seen as such in the community. Another barrier was the time it would take to learn a new approach to music making. Many disabled people have pain and fatigue conditions which means that they have less time to function on a daily basis. They may only be able to concentrate or cope with the display screen or controllers for 10-20 minutes a day. This means that it is important that results can be heard after a relatively low amount of time working with the software. In addition to this requirement, two of the interviewees told me that many disabled music technology applications have a limited progression, which does not allow for continued growth and learning. So although the research shows that the disabled musicians would like to see results fast, they would also like there to be the opportunity to grow with the software and increase complexity as they learn. Some of the musicians raised the issue of unnecessary technology being made for the disabled community without their consultation. As discussed above with the Sign Language Gloves. A further barrier was raised around the issue of updating and fixing problems. Much software is made specifically for disabled people by individual developers and university departments. However there is limited budget and timeframe for these projects once the software is built. This then leads to issues when the software is no longer compatible with the latest updates, or the disabled person accidentally changes a setting that requires in depth knowledge to fix. This issue of bespoke software requires bespoke support which is rarely available. Therefore, there is a preference for standardized software to be made disabled friendly so that it is regularly updated along with the mainstream aspects of the programme. Another barrier was the perceived difficulty of learning to code computer music, and a gap in knowledge around what live coding is. In some cases it was considered as an activity where the

musicians would have to start from scratch and was only suited to programmers or coders.

Preferred methods of learning new software

Methods showed a diversity of preferences, with the majority wanting to learn by trial and error, with help files to refer to as support. The quality and accessibility of help files was brought up by at least half of the interviewees as an issue for learning new software. Our interviewees told us that help files must be written in clear language, in an accessible font and be consistent throughout the programme. A small minority preferred to use video tutorials, but with the stipulation that these should be scripted, structured, captioned and given tags to enable the user to skip through to the section they require. Video tutorials should show the actions at the same time as the explanation, along with clear captions. The view of the screengrab must be clear and legible. Interestingly, those who preferred video tutorials were also the people who had the most experience of working with music technology and live coding. It is possible that this correlation shows that video tutorials are currently more accessible for those with background knowledge and / or confidence in the field (although more research would be necessary as this is a small sample size). Only one person said that they would prefer to have someone help them on a one to one basis, with two expressing that this would be their absolute last resort due to social anxiety or communication barriers. Three said that they would find an online workshop situation difficult due to social anxiety, attention span or communication issues. Accessing forums for support was also considered to be challenging for reasons of social anxiety and a fear of being dismissed or undermined for asking 'stupid questions'. It was also raised that often the responses received in forums can be either incorrect or overly complex, leading to frustration. Finally, there were comments about how forum environments can often be challenging for those for whom social interaction is difficult, and can lead to arguments or flaming due to confusion over the tone of a comment.

What would a disabled friendly gig look like?

The response to this question showed a range of requirements and preferences amongst the group. In a live environment where the performer was present, most welcomed a quiet or silent space where they could relax before, during and after the gig. It was also mentioned that many chill out rooms tend to become colonized by groups and their focus changes from quiet space to alternative music or VIP space. One of the musicians said a private space for people to go to if they needed space and quiet would be good for the general audience. There were also requests for healthy / diet appropriate food and drinks (for those who have allergies and intolerances) and sugary foods and drinks (for those with diabetes). A flexible approach to programming (the running order) was also considered desirable, with musicians having the ability to rearrange their performance order, or slightly adjust times based on how they were feeling. However, one of the musicians said they needed a clear structure and knowledge of what would happen in which section, and how long each section would be. Other musicians asked for "relaxed performances", where the generally accepted protocols of attending a concert were suspended. One of the musicians said that he would prefer to blend into something which had already started and then drop out when he felt he needed to. For him, arriving at a specific time and being ready to perform was a huge anxiety trigger. Four of the musicians liked the option of being able to pre record something and send it in advance in case of illness, and 3 of the musicians said they would prefer to perform in the venue, but would like the option of a networked performance if they were unable to make it in person. In addition to these specific needs, all interviewees agreed on wheelchair accessible venues, captioned performances, signing translation and remote access.

What issues are there around performing live in a networked ensemble?

Many of the performers cited a lack of eye contact and visible communication between players to be a major concern in playing in a networked ensemble. Others were concerned about computer latency, and internet speeds. Concerns around latency were also expressed in terms of hearing back the ensemble playing out of sync with your own playing.

Preference for a networked ensemble or live coding ensemble

Three musicians said that they would prefer an opportunity to perform via network with other disabled musicians using their existing musical set up. Four musicians said that they were currently interested in finding out more about live coding and being part of a live coding networked ensemble.

Implementing Findings

These responses lead to two specific channels of development, firstly, what changes could we make to live coding software, which would support disabled people in becoming part of the community? Secondly, how might we change our working practices to accommodate disabled people in ensembles and performances?

Software Design Messages

1. The need for complete flexibility of layout, design and display, allowing people to create a workspace which works for them.
2. The need for the software to deliver musical results quickly, but also allow for ongoing progression and development of skills and complexity.
3. The need for well documented, plain language, accessible help files.
4. The need for captioning and scanning through video tutorials.
5. The need for the software to be accessible on both computer and tablet, with the option of using assistive hardware such as eyegaze or Headmouse.
6. The need for disabled access to be fundamentally a part of the main programme for any software to reduce issues around updates.
7. The need for more disabled people to be involved in design and making of their own technology rather than acting as focus groups for non disabled makers.

Changes in Working Practices

1. Learning and development in the disabled musician community around Live Coding, and approaches to making music in this way.
2. The need for performers to be able to dip in and out of obligations depending on their circumstances, without this being seen as a negative by others.
3. The need for live performances to be flexible, relaxed, with appropriate rest spaces and nutrition available.

What we are doing next

I am working with the Universal Design Research Centre at OCAD (Ontario College of Art and Design) to make a software system which will allow networked performance from a wide range of disabled musicians on various platforms using various music programmes.

I will then invite the interviewees and a wider group of disabled musicians to explore the software and begin making music. There will be several one to one feedback sessions and a discussion group where the musicians can advise on further potential issues or adaptations.

The group will meet online and explore ways in which we can use the platform to make music together. Within this group we will create a "working with" document for each person to outline any needs they may have around rehearsal or performance etiquette, communication needs or group behaviours. As a group we will then work to find a way to work which supports and respects each of those needs equally. This working practice will reflect a non-hierarchical networked structure, allowing for flexibility and non-judgmental responses to requests.

References

Armitage, Joanne Spaces to Fail in: Negotiating Gender, Community and Technology in Algorave, Dancecult: Journal of Electronic Dance Music Culture, 2018

Berghs, Maria & Chataika, Tsitsi, The Routledge Handbook of Disability Activism, Routledge 2019

Centre For Excellence In Universal Design, What is universal design? The 7 Principles, http://universaldesign.ie/What-is-Universal-Design/The-7-Principles/ Accessed 16th September 2019

Charlton, J. I. 1998. Nothing about us without us : disability oppression and empowerment, Berkeley, Calif. ; London, University of California Press

Errard, Micheal, Why Sign Language Gloves Don't Help, The Atlantic, 9 November 2017. https://www.theatlantic.com/technology/archive/2017/11/why-sign-language-gloves-dont-help-deaf-people/545441/ Accessed September 16th 2019

Eubanks, V. 2017. Automating inequality : how high-tech tools profile, police, and punish the poor, New York, NY, St. Martin's Press

Inclusive Design Research Centre Co-designing Inclusive Cities https://cities.inclusivedesign.ca/ Accessed 16th September 2019.

N. Del Angel, L, Teixido M, Ocelotl E, Cotrina I, Ogborn D Bellacode: localized textual interfaces for live coding music. International Conference on Live Coding, 2019 https://iclc.livecodenetwork.org/2019/papers/paper111.pdf

Noble, S. U. 2018. Algorithms of oppression : how search engines reinforce racism. New York, New York University Press.

Ocelotl, Emilio & N. Del Angel, Luis & Teixido, Marianne. 2018.Saborítmico: A Report From the Dance Floor in Mexico. Dancecult. 10. 10.12801/1947-5403.2018.10.01.11.

Ogborn, D., Beverley, J., N. Del Angel, L., Tsabary, E., Mclean, A., Betancur, E. 2017. Estuary: Browser-based Collaborative Projectional Live Coding of Musical Patterns. International Conference on Live Coding, Morelia, Mexico.

Oliver, Mike (23 July 1990)The Individual and Social Models of Disability. Available at: https://disability-studies.leeds.ac.uk/library/

Pearson, C., And Trevisan, F. (2015) Disability activism in the new media ecology: campaigning strategies in the digital era. Disability and Society, 30(6), pp. 924-940.

Roberts, C., & Kuchera-Morin, J. 2012. Gibber: Live Coding Audio in the Browser. Proceedings of the International Computer Music Conference, 64–69.

Shein, F., Treviranus, J., Brownlow, N. D., Milner, M., & Parnes, P. 1992. Human-Computer Interaction by People with Physical Disabilities. International Journal of Industrial Ergonomics, 9(2), 171-181.

Skuse, Amble & Knotts, Shelly, Diversity = Algorithmic, International Conference on Live Coding, 2018

Woodward, JAames 1972. Implications for sociolinguistic research among the deaf. Sign Language Studies 1, 1-7.

Live coding in Western classical music

Álvaro Cáceres Muñoz
Theatre, Film, TV and Interactive Media, the University of York
alvaro.caceres@york.ac.uk

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

This research project explores how to maximize the usability of live coding tools (which allow improvising music with programming) for classical musicians.

To do so, two goals are set for the project: understanding how live coding can be used in traditional composition, and understanding how do classical musicians feel when live coding. These goals have been achieved using a prototype for a live coding system, which is specifically designed for classical musicians.

Current literature and technologies have been studied to think of potential target users and their skills and needs. Based on this, the prototype has been designed (both from HCI and programming language design perspectives), implemented, and evaluated by classical musicians.

Results show that experienced composers may benefit the most from live coding systems, provided they offer enough expressiveness, responsiveness and feedback, and that their grammar is aligned with musical language.

Keywords: Live coding, Classical music, HCI, Human-computer interaction, Usability, Improvisation, Composition, Algorithmic composition

Research Question

Western-based music creation largely relies on pattern transformation. Therefore, programming algorithmic nature seems to match classical musicians mindset. The hypothesis proposed is that classically-trained musicians use conventional instruments to improvise and compose music, but they could benefit from live coding. However, current live coding programming languages are more oriented towards either experienced programmers or musicians performing electronic music. Therefore, this research tries to address the following problem: How can live coding be used in classical music?

Aims and Objectives

Such problem exposes a series of goals to fulfil.

- Address how live coding can be used in traditional composition: improvisation and programming share things in common (abstraction, structuring, immediateness), but classical musicians may not feel comfortable programming. Live coding systems should reduce these difficulties while helping them to create music more efficiency.

- Understand how classical musicians feel when using live coding: to make sure the usability of these systems is maximized for them.

In order to do this, a prototype for a live coding system has been developed that allows musicians to create music easily. The system notation has been designed to be as similar to solfége notation as possible. Also, it combines code typing with MIDI input, to provide a balance between algorithmic music generation and traditional instrument playing (which classical musicians are more familiar with). This prototype has been evaluated with users using qualitative methods to make sure that it is well suited for their specific needs.

Motivation

Live coding is becoming increasingly popular, with new languages being released every year. Most of these languages are designed for sound synthesis and electronic music. Because of this, this project analyzes how live coding could be used in classical music. The hypothesis for this project has been tested with a high fidelity prototype of a live coding system (BachTracking), based on preliminary user feedback and a study of current live coding programming languages.

Background and related work

Live coding programming languages

Several programming languages have been described in this section, each highlighting different aspects relevant to live coding for classical music. SuperCollider (Wilson et al. 2011) focuses on sound, has an object-oriented and decoupled architecture, and its OSC compatibility makes it the basis of other languages. One of them is SonicPi, designed for school students (Aaron 2016), by providing intuitive UI and documentation. TidalCycles controls SuperCollider via MIDI and OSC and relies on Laurie Spiegel's pattern transformations (Spiegel 1981) to create new music. It is functional like Extempore (Sorensen and Gardner 2017), which grants live low-level audio programming (Sorensen 2014). Other languages like Max and Pure Data (Zimmer 2007) use visual paradigms; similarly, OpenMusic (Bresson et al. 2010) includes graphical score notation. Orca (GitHub 2019) is visual and text-based, and it uses compact base-36 notation. Gibber (Roberts et al. 2015) and Serialist (Github 2016) explore abstractions that feel more familiar to musicians (e.g. a score).

HCI perspective

Magnusson (2019) states that computers are symbolically controlled, but physically actuated; this can pose UI mapping problems. Pane et al. (2002) suggest that "idiomatic" syntax improves programmers learning process, and so special characters should be avoided whenever possible. Wanderley and Orio (2002) propose that usable electronic instruments should be easy to learn, explore and modify, and they should allow precise timing.

Music composition techniques

Like programming, composition exploits the concept of transforming small music fragments. Counterpoint (Encyclopedia Britannica 2019; Jackson 2013) follows this philosophy, using three basic transforma-

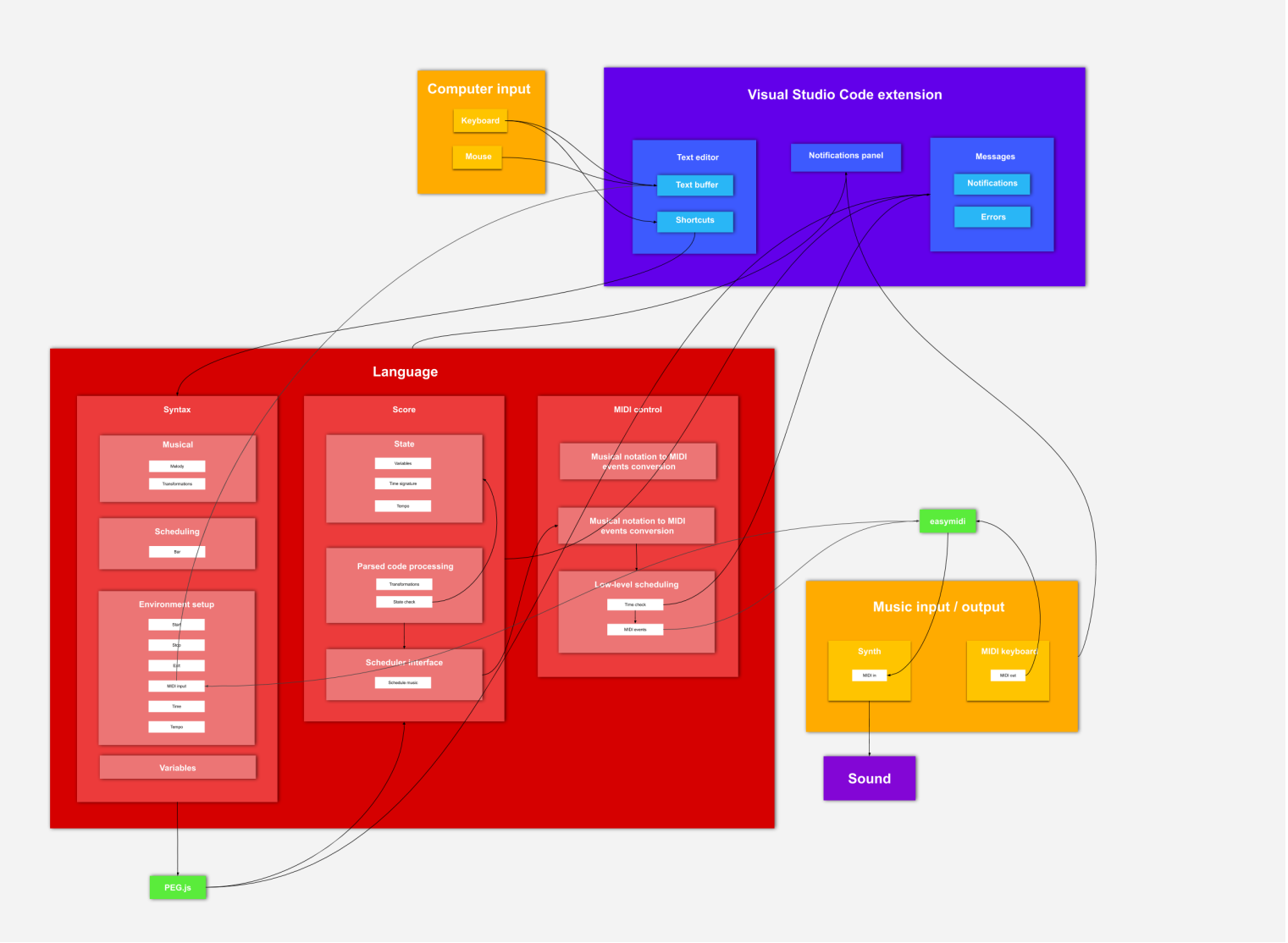


Figure 1: System overview

tions: inversion, augmentation/diminution, and retrograde. Based on counterpoint, serialism (Forte 1973) abstracts notes as numbers, sets and vectors of distances (intervals). Spiegel (1981) proposed similar techniques, specifically focusing on pattern transformation.

Design

Target users

Table 7.1 shows classical musicians can be divided into:

User	Music Expertise	Coding Expertise	Music Abstraction
Composer	High	Low	High
Player	High	Low	Intermediate
Student	Intermediate	Intermediate	Low

Table 7.1

A system that maximizes usability for these users should take their specific skills and needs into account (for instance, it should be possible to write music either note by note or using more abstract structures).

Language

Figure 1 provides an overview of the system. It offers the metaphor of an editor, a score and instruments. The score can be updated and read sequentially. Reading the score sends MIDI to instruments (keyboards, VSTs...)

The grammar (Figure 2) tries to be as similar to natural language as possible. Transformations can process melodies, variables or transformations of any of those. They have been inspired by counterpoint and Spiegel's work: transposition, inversion, retrograde, retrograde inverse, join and mirror. Parenthesis are not required, and shorthand names are available.

User interface

Figure 3 shows the UI proposed for the prototype. The text editor allows users to edit text without modifying the score. Errors are displayed in red, to call the user's attention. Keyboard shortcuts follow consistency whenever possible (Ctrl-S Starts reading, Ctrl-Shift-S stops reading).

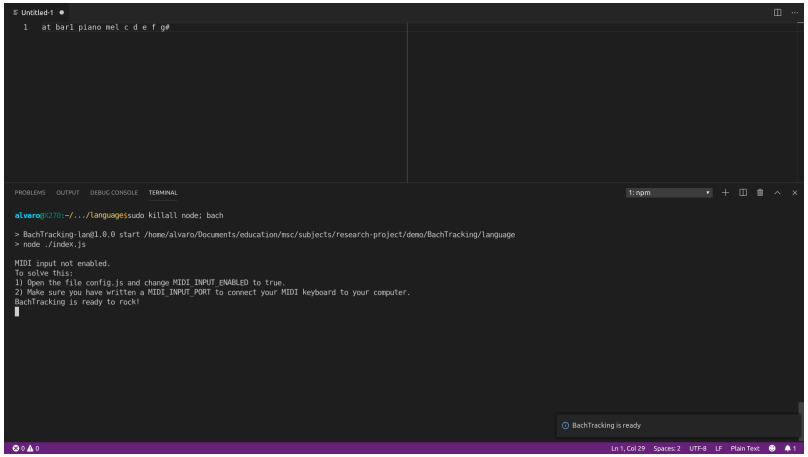
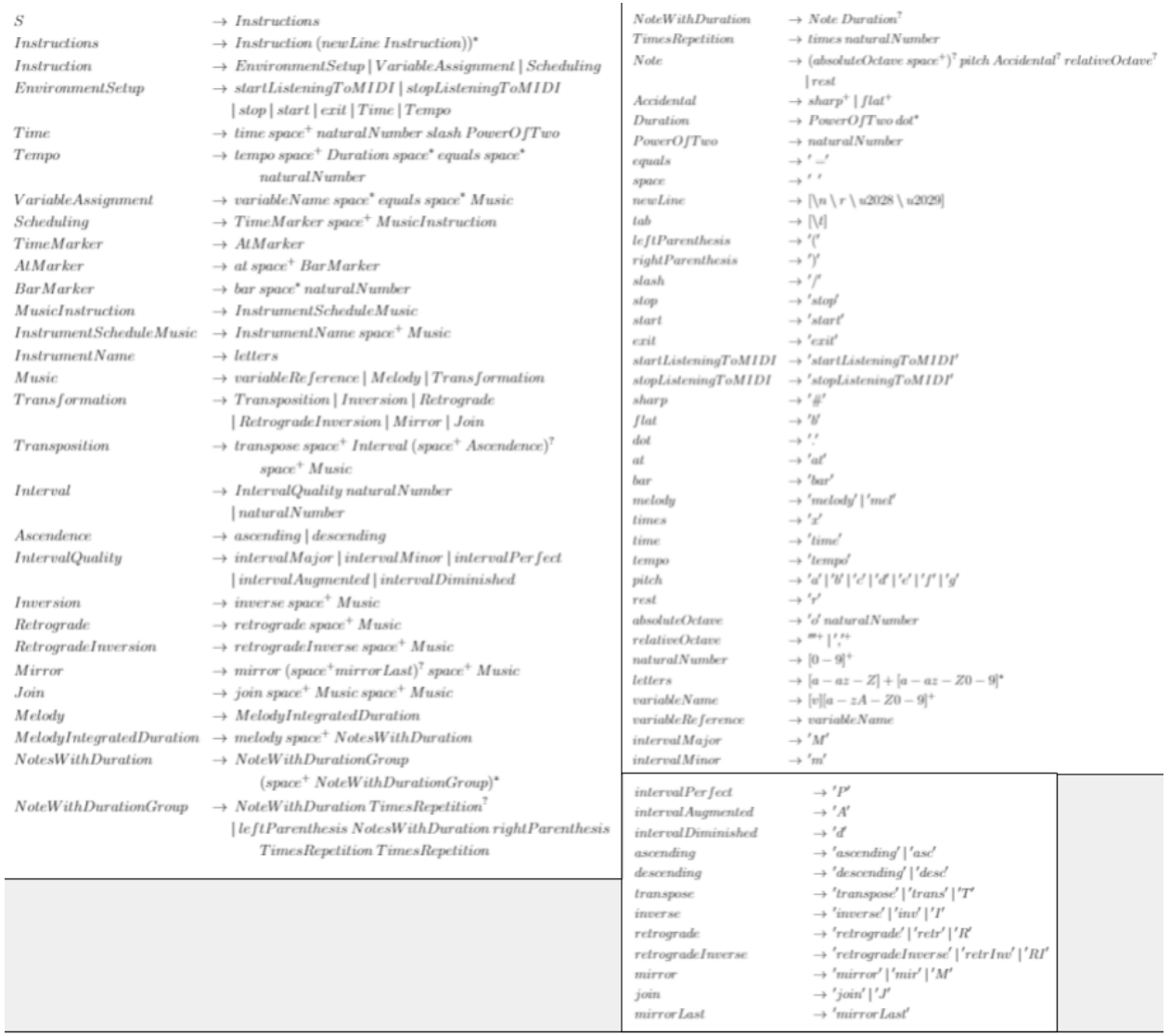


Figure 3: UI Layout

MIDI note input is available as well, inspired by music notation software (Avid 2018). Input is activated with a keyboard shortcut, and it writes notes as plain text (code) in the text editor, so they can be manually edited if needed.



$$
\begin{array}{ll}
S & \rightarrow Instructions \\
Instructions & \rightarrow Instruction\ (newLine\ Instruction))^{*} \\
Instruction & \rightarrow EnvironmentSetup \mid VariableAssignment \mid Scheduling \\
EnvironmentSetup & \rightarrow startListeningToMIDI \mid stopListeningToMIDI \\
 & \quad \mid stop \mid start \mid exit \mid Time \mid Tempo \\
Time & \rightarrow time\ space^{+}\ naturalNumber\ slash\ PowerOfTwo \\
Tempo & \rightarrow tempo\ space^{+}\ Duration\ space^{*}\ equals\ space^{*} \\
 & \quad naturalNumber \\
VariableAssignment & \rightarrow variableName\ space^{*}\ equals\ space^{*}\ Music \\
Scheduling & \rightarrow TimeMarker\ space^{+}\ MusicInstruction \\
TimeMarker & \rightarrow AtMarker \\
AtMarker & \rightarrow at\ space^{+}\ BarMarker \\
BarMarker & \rightarrow bar\ space^{*}\ naturalNumber \\
MusicInstruction & \rightarrow InstrumentScheduleMusic \\
InstrumentScheduleMusic & \rightarrow InstrumentName\ space^{+}\ Music \\
InstrumentName & \rightarrow letters \\
Music & \rightarrow variableReference \mid Melody \mid Transformation \\
Transformation & \rightarrow Transposition \mid Inversion \mid Retrograde \\
 & \quad \mid RetrogradeInversion \mid Mirror \mid Join \\
Transposition & \rightarrow transpose\ space^{+}\ Interval\ (space^{+}\ Ascendence)^{?} \\
 & \quad space^{+}\ Music \\
Interval & \rightarrow IntervalQuality\ naturalNumber \\
 & \quad \mid naturalNumber \\
Ascendence & \rightarrow ascending \mid descending \\
IntervalQuality & \rightarrow intervalMajor \mid intervalMinor \mid intervalPerfect \\
 & \quad \mid intervalAugmented \mid intervalDiminished \\
Inversion & \rightarrow inverse\ space^{+}\ Music \\
Retrograde & \rightarrow retrograde\ space^{+}\ Music \\
RetrogradeInversion & \rightarrow retrogradeInverse\ space^{+}\ Music \\
Mirror & \rightarrow mirror\ (space^{+} mirrorLast)^{?}\ space^{+}\ Music \\
Join & \rightarrow join\ space^{+}\ Music\ space^{+}\ Music \\
Melody & \rightarrow MelodyIntegratedDuration \\
MelodyIntegratedDuration & \rightarrow melody\ space^{+}\ NotesWithDuration \\
NotesWithDuration & \rightarrow NoteWithDurationGroup \\
 & \quad (space^{+}\ NoteWithDurationGroup)^{*} \\
NoteWithDurationGroup & \rightarrow NoteWithDuration\ TimesRepetition^{?} \\
 & \quad \mid leftParenthesis\ NotesWithDuration\ rightParenthesis \\
 & \quad TimesRepetition\ TimesRepetition \\
NoteWithDuration & \rightarrow Note\ Duration^{?} \\
TimesRepetition & \rightarrow times\ naturalNumber \\
Note & \rightarrow (absoluteOctave\ space^{+})^{?}\ pitch\ Accidental^{?}\ relativeOctave^{?} \\
 & \quad \mid rest \\
Accidental & \rightarrow sharp^{+} \mid flat^{+} \\
Duration & \rightarrow PowerOfTwo\ dot^{*} \\
PowerOfTwo & \rightarrow naturalNumber \\
equals & \rightarrow \text{'\_'} \\
space & \rightarrow \text{' '} \\
newLine & \rightarrow [\backslash n \setminus r \setminus u2028 \setminus u2029] \\
tab & \rightarrow [\backslash t] \\
leftParenthesis & \rightarrow \text{'('} \\
rightParenthesis & \rightarrow \text{')'} \\
slash & \rightarrow \text{'/'} \\
stop & \rightarrow \text{'stop'} \\
start & \rightarrow \text{'start'} \\
exit & \rightarrow \text{'exit'} \\
startListeningToMIDI & \rightarrow \text{'startListeningToMIDI'} \\
stopListeningToMIDI & \rightarrow \text{'stopListeningToMIDI'} \\
sharp & \rightarrow \text{'\#'} \\
flat & \rightarrow \text{'b'} \\
dot & \rightarrow \text{'.'} \\
at & \rightarrow \text{'at'} \\
bar & \rightarrow \text{'bar'} \\
melody & \rightarrow \text{'melody'} \mid \text{'mel'} \\
times & \rightarrow \text{'x'} \\
time & \rightarrow \text{'time'} \\
tempo & \rightarrow \text{'tempo'} \\
pitch & \rightarrow \text{'a'} \mid \text{'b'} \mid \text{'c'} \mid \text{'d'} \mid \text{'e'} \mid \text{'f'} \mid \text{'g'} \\
rest & \rightarrow \text{'r'} \\
absoluteOctave & \rightarrow \text{'o'}\ naturalNumber \\
relativeOctave & \rightarrow \text{'}^{+}\text{'} \mid \text{','}^{+} \\
naturalNumber & \rightarrow [0-9]^{+} \\
letters & \rightarrow [a-az-Z] + [a-az-Z0-9]^{*} \\
variableName & \rightarrow [v][a-zA-Z0-9]^{+} \\
variableReference & \rightarrow variableName \\
intervalMajor & \rightarrow \text{'M'} \\
intervalMinor & \rightarrow \text{'m'} \\
intervalPerfect & \rightarrow \text{'P'} \\
intervalAugmented & \rightarrow \text{'A'} \\
intervalDiminished & \rightarrow \text{'d'} \\
ascending & \rightarrow \text{'ascending'} \mid \text{'asc'} \\
descending & \rightarrow \text{'descending'} \mid \text{'desc'} \\
transpose & \rightarrow \text{'transpose'} \mid \text{'trans'} \mid \text{'T'} \\
inverse & \rightarrow \text{'inverse'} \mid \text{'inv'} \mid \text{'I'} \\
retrograde & \rightarrow \text{'retrograde'} \mid \text{'retr'} \mid \text{'R'} \\
retrogradeInverse & \rightarrow \text{'retrogradeInverse'} \mid \text{'retrInv'} \mid \text{'RI'} \\
mirror & \rightarrow \text{'mirror'} \mid \text{'mir'} \mid \text{'M'} \\
join & \rightarrow \text{'join'} \mid \text{'J'} \\
mirrorLast & \rightarrow \text{'mirrorLast'}
\end{array}
$$

Figure 2: BachTracking grammar

Implementation

Language

In this system [1], the language works as a server that listens to text coming from the UI. This text is parsed and sent to a score object. The score controls a MIDI scheduler, which communicates with any connected instruments. It also handles MIDI note input (Figure 4).

User interface

The UI has been developed as a Visual Studio Code extension, given its active support (as of 2019). The UI (Figure 3) has three main software-based UI elements[2]: text editor, notifications panel, and information/error messages.

Evaluation

Qualitative user evaluation was applied, and it was split into two phases:

- Online interviews: users followed semi-structured interviews about their experience making music with technology and live coding. This was done first to improve the prototype before testing it.

- In-lab user evaluation: users were asked to read a tutorial explaining how to use BachTracking during one day, before trying the system. After that, I gave them a small crash course on how to use BachTracking with the MIDI keyboard (this was done in my house, using a laptop and a MIDI keyboard that had been tested beforehand). Then they were asked to try the system by playing some music with BachTracking. After this, they followed a semi-structured interview to get feedback about their experience creating music with this prototype.

Online interview

Demographics were balanced in this case. 3 out of 6 participants were professional composers (in their mid-twenties) with studies at graduate level from Kings College London, Conservatorium van Amsterdam (Netherlands) and Katarina Gurska music academy (Spain). Two participants were intermediate classical piano students (18-22 years old) at the Professional Music Conservatory of Getafe (Spain). The remaining participant was a music hobbyist in their late fifties with intermediate studies in clarinet, who played in brass music bands in Pinto and Getafe (Spain).

No participants knew about live coding previously, but a few had programmed before. Composers were used to music technologies (VST's, DAW's...), but students mostly used recording and notation software as their music tools; they explicitly stated they preferred acoustic instruments. Participants were asked how they create music, and they described it as first relying on muscle memory and intuition, and then structuring the idea in a more cerebral way.

Online interviews were conducted after starting to develop the prototype and before testing it with in-lab user evaluation, thus helping to refine features and functionality.

In-lab user evaluation

Composers used all of the language's available functions, and they even felt the freedom to experiment and force the system by using notes with an incredibly small duration (thus creating blazing fast

[1] The code can be downloaded from `https://mega.nz/$#$F!RbpizSxB!EOIbhFjPXB8QbwqzrsZGSw` (notice it has only been tested in Ubuntu Studio).

[2] There are also physical UI elements such as the MIDI keyboard, keyboard or mouse

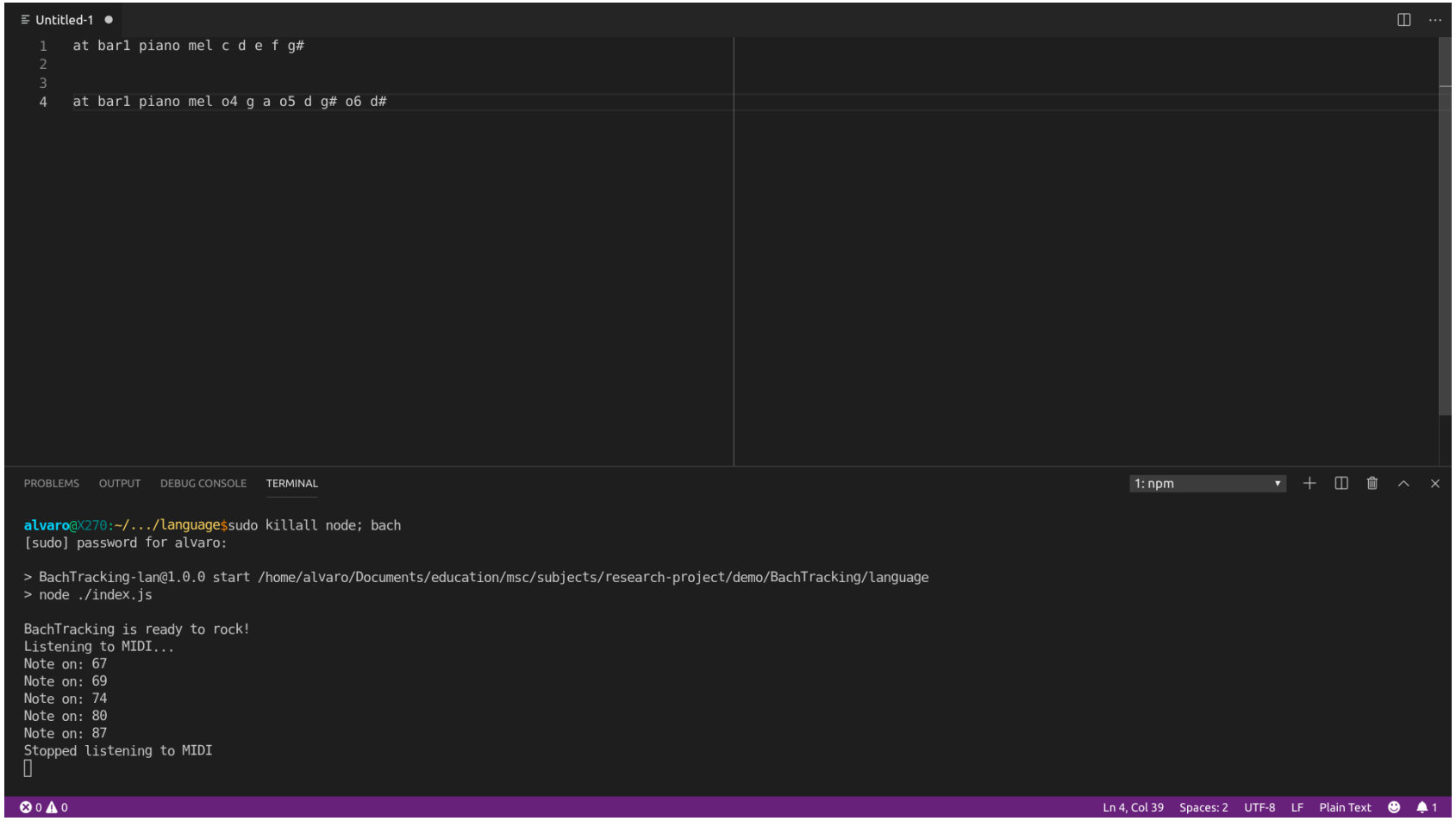


Figure 4: MIDI note input

sequences of notes). On the other hand, music students used MIDI input for the most part of their performances.

Participants found some aspects of the system confusing, like not seeing the bar number printed on the screen, and having to write the octave number before the note duration.

Suggestions included having access to a live graphical score, a graphical preview of the code to be executed, and accidentals omission. The possibility of writing code with MIDI input was proposed to users to see if they would be interested in that feature; composers and players thought it could be powerful, but students found it confusing.

Most participants composed rather than improvising live, perhaps due to their classical background, or to insufficient practice time.

Conclusions

Key results and significance

The following main points can be extracted from this study:

- Expressiveness of the initial musical motif (with traditional instrument input), conciseness and transformation composability can improve live coding usability for classical musicians.
- Classical musicians may prefer using live coding for fast prototyping when writing music.
- Feedback and situation awareness are crucial to provide the responsiveness classical musicians find in the instruments they are used to play.
- User-centered design may help to create live coding programming languages that best adapt to the musicians who are going to use it.

Future work

This project could benefit from more extensive user evaluation, which could point out features to be included in the system, or usability errors that should be corrected. Increasing training time would allow participants to feel more comfortable improvising with BachTracking. Jazz musicians could be considered as participants in the future as well. The language could incorporate more features: scales, tempo transformation, harmony (voice leading), microtonality, larger structural transformations, form... Developing two syntax modes (large/understandable, and concise/ergonomic) could make the system more flexible, and suitable to both compose and improvise.

References

Aaron, S. (2016). Sonic Pi – performance in education, technology and art. International Journal of Performance Arts and Digital Media, 12(2), pp.171-178.

Avid. (2018). Sibelius Reference Guide version 2018.1. [online] Available at: `http://resources.avid.com/SupportFiles/Sibelius/2018.1/Sibelius_2018.1_Reference_Guide.pdf` [Accessed 18 Apr. 2019].

BBC. (2019). Melody - Edexcel - Revision 6 - GCSE Music - BBC Bitesize. [online] Available at: https://www.bbc.com/bitesize/guides/zwj2jty/revision/6 [Accessed 9 Apr. 2019].

Bresson, J., Agon, C., and Assayag, G. (2010). OpenMusic – visual programming environment for music composition, analysis and research. ACM MultiMedia (MM'11).

Encyclopedia Britannica. (2019). Inversion | music. [online] Available at: `https://www.britannica.com/art/inversion-music` [Accessed 13 Apr. 2019].

Forte, A. (1973). The Structure of Atonal Music. Yale University Press.

GitHub. (2016). serialist. [online] Available at: https://github.com/irritant/serialist [Accessed 5 Mar. 2019].

GitHub. (2019). Orca: Esoteric Programming Language. [online] Available at: https://github.com/hundredrabbits/Orca [Accessed 5 Mar. 2019].

Jackson, R. (2013). Counterpoint | music. [online] Encyclopedia Britannica. Available at: https://www.britannica.com/art/counterpoint-music [Accessed 11 Apr. 2019].

Magnusson, T. (2019). Sonic writing. New York, NY: Bloomsbury Publications.

Pane, J., Myers, B., and Miller, L. (2002). Using HCI techniques to design a more usable programming system. Proceedings IEEE 2002 Symposia on HumanCentric Computing Languages and Environments. IEEE, pp. 198–206.

Roberts, C., Wright, M., and Kuchera-Morin, J. (2015). Music program-ming in gibber. ICMC.

Sorensen, A. (2014). GOTO 2014 Programming In Time - Live Coding for Creative Performances

Andrew Sorensen. [online] Youtube. Available at: https://www.youtube.com/watch?v=Sg2BjFQnr9s [Accessed 20 Apr. 2019].

Sorensen, A., and Gardner, H. (2017). Systems level liveness with extempore. Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM, pp. 214–228.

Spiegel, L. (1981). Manipulations of musical patterns. Proceedings of the Sym-posium on Small Computers and the Arts, pp. 19–22.

SuperCollider Docs. (2019). 02. First Steps | SuperCollider 3.10.3 Help. [online] Available at: http://doc.sccode.org/Tutorials/Getting-Started/02-First-Steps.html [Accessed 23 Apr. 2019].

Wanderley, M. and Orio, N. (2002). Evaluation of input devices for musical expression: Borrowing tools from hci". Computer Music Journal 26.3, pp. 62–76.

Wilson, S., Collins, N. and Cottle, D. (2011). The SuperCollider book. Cambridge, Mass.: MIT Press.

Zimmer, F. (2007). Bang. Hofheim: Wolke, p.133.

Live Coding Tools for Choreography: Creating Terpsicode

Dr. Kate Sicchio
Virginia Commonwealth University
ksicchio@vcu.edu

Zeshan Wang
Virginia Commonwealth University
wangz24@mymail.vcu.edu

Marissa Forbes
Virginia Commonwealth University
forbesmc@mymail.vcu.edu

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

Terpsicode is a developing mini programming language for live coding dance performance scores. This paper explores the process of creating a choreographic patterning language using images, and discusses the creation, capturing, and naming of movement. It also reflects on the premiere performance using this system with a live coder and improvising dancer and how score-making with code may be translated by a performer. The final result of this venture seeks to provide a computer language for choreography that utilises dance terminology alongside visual performance scores that may be used within various improvising settings.

Introduction

This paper discusses an ongoing process in the development of a programming language for choreography. It explores the idea of sampling movement through still images as a way of forming discrete units for creating patterns over time. These patterns in turn become composed through algorithmic processes. The main goal of the language is to be utilized within live coding practices, as seen in previous work around choreography, and in real time score making[1]. The language itself is meant for a choreographer to utilise when creating work, building off of choreographic tools by artists such as Myriam Gourfink (Gourfink, 2013), Wayne McGregor or WIlliam Forsythe (de Lahunta, 2014).

Terpsicode will allow for computers and algorithmic processes to become more integrated into choreographic practices. The unexpected outcomes of image concatenation will inspire artists to break out of habitual movement patterns, which creates new movement possibilities for choreographers and dancers to consider unexplored avenues for creative work and improvisation (Collins, 2011).

Project Background

Choreography lends itself well to the realm of live coding due to the nature of dance, and how just within a single performance can alter itself from the so-called score. Unlike music or language, up to the present there is no widely accepted "language" for transcribing dance. It cannot be replicated without being experienced, to see or feel a body moving through time (Birringer, 2013). This inherent "liveness" and mistranslation of the original is magnified with the use of live code simply as a medium to invoke reactions from dancers without thought, and the programmer's own lack of expectation of the final outcome of their code.

Within dance, one way of communicating movement is the use of scores. Scores may be compared to musical scores in that they may notate a work but more so in dance they are used as a starting point for movement instructions that are to be performed by a dancer or dancers. Choreographer Jonathan Burrows (2010) discusses two approaches to dance scores, one being akin to a classical music score where there are clear instructions for a piece, but the other being an inspiration for a performer. "...what is written or thought is a tool for information, image and inspiration, which acts as a source for what you will see, but whose shape may be very different from the final realization." Score are also common in live arts practices and have been part of various art movements such as Fluxus in the 1960s. Scores may be given to performers via speech, text, visuals or other means and may be created before, during or after a performance. Many dance improvisation artists work from scores, or sets of guidelines or instructions meant to be interpreted in order to create movement in a given time and space.

Because the movement and sequences in dance can be algorithmic in nature, communication between a programmer and dancer is not so difficult to accomplish. The body is instructed to move between basic positions, one after another, forwards or backwards or repeated in loops, or maybe splitting their interpretations like an if/else statement. These analogies provide a natural transition to transcribe dance from paper and oral scripts to codes and bits.

Moving Patterns

Developing Terpsicode has taken several steps, during which language, imagery, and choreography have been developed. This process was also inspired by the previous performance work Moving Patterns.

Moving Patterns was a live coded dance performance, which utilized DanceDirt[1], a custom library written by Tom Murphy for TidalCycles[2]. In DanceDirt, the programmer was able to use the patterning capabilities of TidalCycles to create visual patterns of images, which were viewed in VLC player. Within the performance, the images acted as a score for dance improvisation. The piece created a feedback loop between the performer, who improvised movement based on the projected images, and the live coder, who selected the images and composed them in real-time sequences in reaction to the performer's movements. The collaboration of these two elements resulted in a codependent performance in which the coder and the performer sought direction from each others' decisions.

However, Moving Patterns made it apparent that TidalCycles did not have the capabilities to live code choreography in a way that felt complete for a dance composition. Much of its grammar and functions were based upon principles found in sound, or that result in specific sonic outputs such as reverb or reverse. The desire to develop a language more adapted to the vocabulary used when working in a dance studio arose during this process. Thus, the incentive to design a choreographic programming language was realized.

Sampling Movement with Photos

One of the questions that live coding visual choreographic scores raises is what medium is best used to convey instructions or inspiration to the performer. Still images were decided to be an appropriate medium due to the atomic nature of photo in capturing movement and their ability to be sequenced arbitrarily, like shuffling frames in a video.

Such flexibility allows for the spontaneity that defines live coding. Although images have become a starting point for Terpsicode, other media such as video or motion capture data may be more appropriate at a later point in this process of language creation.

The images used to create the visual score are from a large library of photographs taken through a time lapse camera. The camera took two photos per second of a dancer improvising movement. This resulted in a range of imagery that could be used for sampling movement.

*

Similar to how a sound sample is a select moment from a longer audio track, the still image is a single moment from a larger set of movements. While this could be seen as problematic in the documentation of dance, as it is missing key elements of movement such as spatial paths and dynamics, it provides interesting source material for dance improvisation and choreography. Choreographers will have access to frames of movement that they can piece together in unconventional ways, and dancers must instantaneously interpret the connections between those images. These photo libraries reference the initial Moving Patterns performance, where the use of images as samples was first implemented as a framework for improvisation.

Tagging Movement

After the images were captured, the clustering algorithm t-SNE was used to assemble like images together into one document, resulting in a spread of groupings based on visual similarities. The images were then labeled according to the dancer's inferred actions. t-SNE lacked the ability to understand the body as a three dimensional object, leading to discrepancies in how the images were to be labeled. For example, if the dancer was doing the same movement but facing different directions, the algorithm recognized the directional shift as separate from the original. This resulted in similar images existing in different groupings throughout the document. These unnecessary separations made labeling the images more difficult. Additionally, the process of tagging still images of motion unearthed the apparent gap in kinesthetic understanding of movement versus the ability to verbalize the action, or rather, one frame in an assumed action.

Figure 2: Movement terms for development of the grammar and syntax of the programming language.

Manual labeling of frames was done after the initial clusters were formed, adding a human choreographic approach to the tagging of images. This process initially started as a simple way of finding groups

Figure 1: Example of images used within the development of Terpsicode as sorted by the t-SNE algorithm. Dancer: Marissa Forbes.

of images that would have the same name, but it also came to demonstrate how the machine learning algorithm is of service to an artistic process, rather than the art being produced strictly by a computer.

Developing the language

Terpsicode is under development using PEG.js[3] to create a parser and javascript to create a mini language. In order to populate the lexical grammar, appropriate terminology had to be selected from the discipline of dance and choreography.

To begin this process, choreographic vocabulary describing movement, timing, and phrasing was collected and collated into categories. Some of the positional terms were used in tagging the images, while choreographic terms around compositional phrasing and timing structures became the key words for function names in the programming language.

Pseudo code was utilised as a means to determine if the language had the ability to convey choreographic information, from the programmer to the computer, in a way that still worked with the domain specific ways that choreographers communicate with performers. For example, terms such as “retrograde” or “coin flip” expresses language that choreographers understand , but must be broken down in order for computers to process. Writing the pseudo code was useful in determining how pattern structures, timing and rhythms, and movement combinations could be created, as well as what text fragments the computer must be taught.

Writing a parser to pattern the images in a browser window was the next step in the development of the mini language. Taking the pseudo code and applying this to displaying the tagged images was done in javascript. The tagged images can also be patterned in different orders and the length of time they are displayed are also determined in javascript. Figure 3 gives an example of how the code appears in the console log and the image appears in the browser.

Further development of the language includes expanding the image library with more figures, expanding those images outside of the human form, and building Terpsicode into a working plugin for existing compilers.

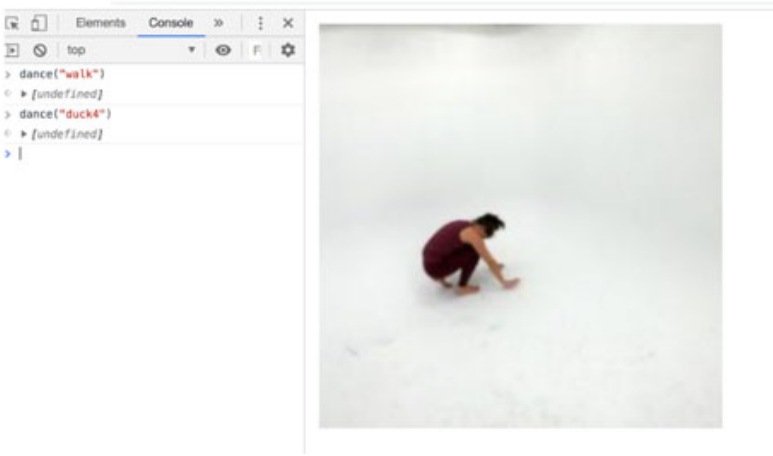


Figure 3: Terpsicode in the browser calling the tagged images.

Terpsicode in Performance

Though the language is still under development, the first performance with Terpsicode was May 17, 2019 at the Festival of Algorithmic and Mechanical Movement in Sheffield, UK, danced by Tara Baker and live coded by Kate Sicchio. Here, a twenty-minute dance was improvised using a limited lexicon from the mini language, which included the terms “fall”, “duck”, “walk”, “stomachspiral” and “flick”. Each word displayed the first tagged image in the directory for that term for a default time of three seconds. As the words were added via the live coding language, images were added into a cycle of pictures. If a number followed the term, that image would appear for the number specified amount of time. The order, pattern and timing of the images were determined by the live coding.

Even with a limited amount of images, language and no real com-

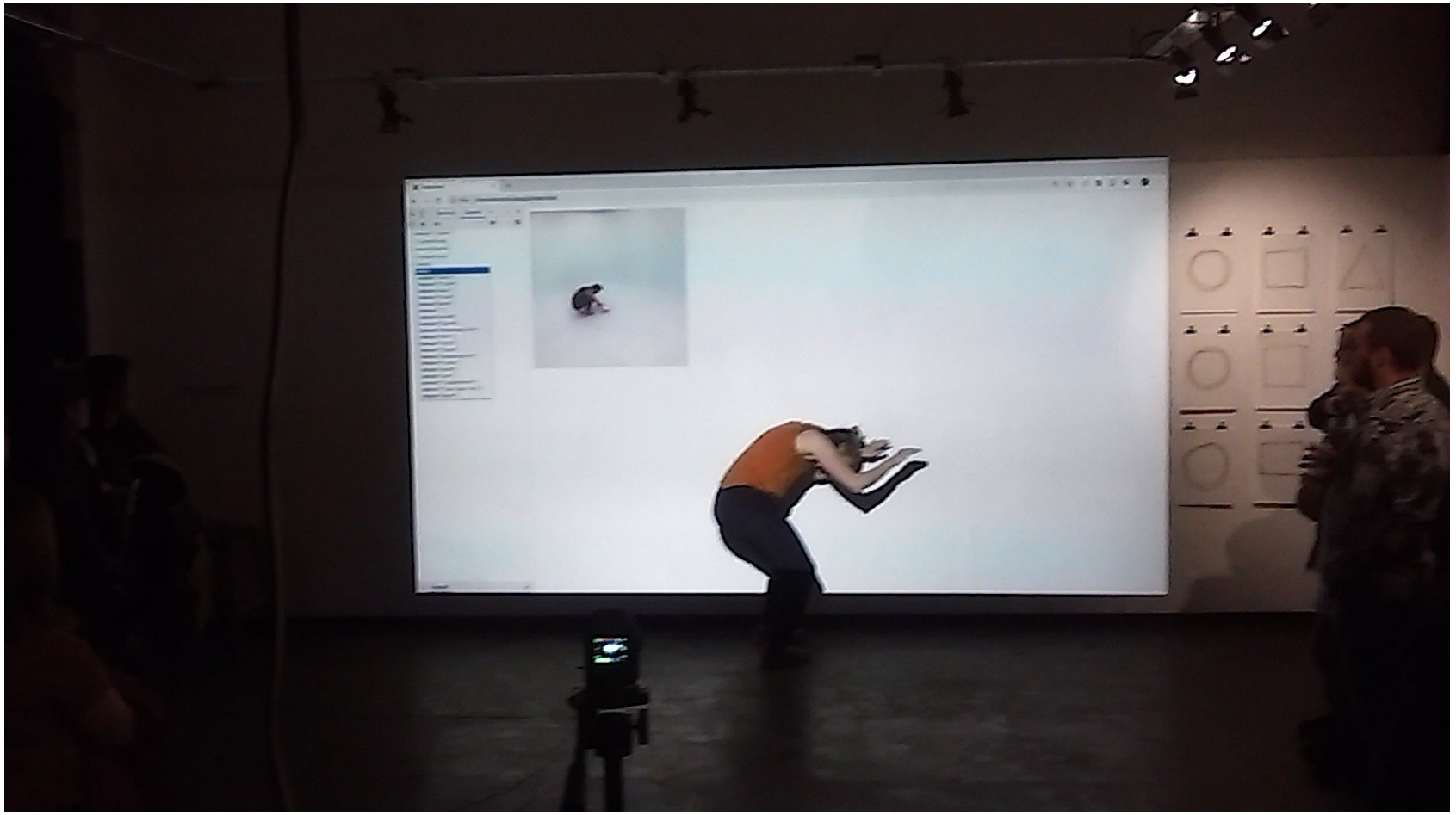


Figure 4: Terpsicode in performance at AlgoMech Festival, May 2019. Dancer: Tara Baker

positional terms available for the performance, a piece was beginning to unfold in this improvisation. The dancer was given the instructions to begin the piece by copying the images and transitioning from shape to shape. The order and pattern of the images should be demonstrated clearly for the first minutes of the work. However as the piece progressed, the dancer was told to respond to the images, giving her more freedom to interpret the shapes or use the images as a starting point for further exploration in her movement.

This approach of strictly following the score versus opening it up for interpretation also affects the live coding of the images. The opening must consider the speed of changes and the complexity of patterns. If the live coder speeds up quickly they must be aware that it may be challenging for the dancer to follow. This may mean certain shapes are not created or an impossible score is created. This in itself is not an issue as this can lead to create problem solving on behalf of the dancer, but it should be made aware as a choice of the live coder. After gaining familiarity with the process, the score becomes more of an inspiration for the dancer, the live coder is less in control of the output and therefore playing more with possibilities for the dancer to explore. These ways of interpreting and working with the visual score resonate with the definitions above from Burrows (2010). Live coding becomes an exploration of the possible but not necessarily what is actually performed because the output is meant to be danced by a human, who has agency within this work to make the final decision on what movement is created.

One issue arising from this work was the presence of the code onscreen next to the images. While this idea was to highlight the nature of the score being produced live and provide an example of the TOPLAP Draft Manifesto, it was questioned as part of this work. The dancer was unclear which to follow at times, the words or the images. Also having the images next to the words may start to imply certain movement must be performed when the shape itself is just an artefact of that movement. This starts to undo some of the reasoning in using still images as described above. The dancer may have less autonomy in the pathways of the shapes when text is also presented as part of the score. More experiments with the text will be explored in future iterations of this piece.

Summary

The development of a mini language for patterning images to create a choreographic score is an ongoing exploration in live coding, sampling movement, and the creation of dance improvisation performance with technology. There is no predominant choreographic dictionary so key words were only derived from professional practice. The process of designing Terpsicode brings to question the naming and vocabulary used for dance in the context of reformatting to programming syntax and how choreographers may further interrogate the utilization of live coding within their creative work. Samples of dance movements were recorded and processed with machine learning algorithms in order to break down some of that vocabulary. Implementation of the resulting language designed for coding was built on javascript. Further development will be made into making Terpsicode more usable. A first iteration was used in performance to create an image based score for dance improvisation, demonstrating how this language can be developed further for this purpose. Future performances will start to explore more compositional elements in the language as well as more ways of presenting the final score within the performance.

References

Birringer, J (2013) What score? Pre-choreography and post-choreography in International Journal of Performance Arts and Digital Media 9(1):7-13.

Burrows, J (2010) A Choreographer's Handbook. Routledge: London.

Collins, N (2011) Live Coding of Consequence in Leonardo 44(3):207–211.

deLahunta, S (2010) The Choreographic Resource: Technologies for Understanding Dance in Contact Quarterly 35(2):18–27.

Gourfink, M (2003) L'Innomee. Available from: www.myriam-gourfink.com/lInnommee.html [accessed Sept 10, 2019].

Sicchio, K (2014) Hacking choreography: Dance and live coding in Computer Music Journal 38 (1), 31-39.

Sicchio, K (2019) Programming paradigms for the human interpreter, International Conference on Live Coding. Available from: http://iclc.livecodenetwork.org/2019/ingles.html [accessed Sept 10, 2019].

The Mégra System - Small Data Music Composition and Live Coding Performance

Niklas Reppel
Eurecat Barcelona
nik@parkellipsen.de

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

This article describes the Mégra music system, a code-based, stochastic music system that can be used in a live performance context, as well as for longer compositions. Mégra relies on Probabilistic Finite Automata (PFA) as its fundamental data structure. A case is made for the use of PFAs as a data model that can not only be trained (in the sense of machine learning), but also be interacted with on the basis of predefined operations and, as a side effect, enables one to creatively use the imperfections that occur when using very small data sets to infer musical sequence generators with the help of machine learning methods.

Introduction

Mégra is a code-based system that can be used to create music in a live performance context, as well as in a composition context. It allows one to interactively infer or create musical sequence generators by using the live coding or exploratory coding method.

Its development started out from the idea of, as Fiebrink and Caramiaux (2018) put it, 'machine learning in which learning algorithms can be understood as a particular type of interface through which people can build model functions from data,' and the subsequent search for a suitable data model. Further criteria were the efficiency of the training methods so that they can be used in the context of real-time music creation, as well as a syntax that reflects the model well while being sufficiently intuitive for live coding.

The 'imperfections' which occur when using very small data sets (small enough to be observed and entered by humans) are a welcome side effect of the chosen approach.

Furthermore, the system attempts to include methods to manipulate the learned structures by means other than just choosing the input data. Instead, it aims to employ a model that is more semantically meaningful in comparison to other models currently common in machine learning and artificial intelligence, such as deep neural

networks. This semantic quality at a high level should allow for the definition of more meaningful operations and interactions inside the model, at a granular level. In that sense, the Mégra language presents a case study on how a machine learning method can be put into action within live coding practice.

Section 2 of this paper will evaluate this idea in the contexts of Algorithmic Composition and the current Big Data trend, or in opposition of the latter. Section 3 will evaluate the chosen data model and eventual alternatives. Section 4 will give a brief overview over the Mégra system, and how the model is used in the context of live coding. Section 5 will present some usage examples.

Small Data as a Creative Tool

The part of the machine learning world commonly associated with the Big Data buzzword seems to be dominated by huge data sets and huge hardware effort. While it is hard to deny that the results are impressive, they are still mostly beyond reach for people that are not part of a company or a research group, as neither the necessary hardware (like supercomputers or AI accelerators) nor the data sets are commonly available[1].

Furthermore, 'black-box' models that are commonly in use, like deep neural networks, barely have any semantic significance for humans (one might say they aren not *cognitively available*). The semantic meaninglessness of the data models makes a 'discourse with models' (Roberts and Wakefield 2018, p. 303), e.g. by applying meaningful operations to them, quite hard, apart from 'shaping model behavior through data' (Fiebrink and Caramiaux 2018). Manually modifying the internal structure of the 'black box' would not only be tedious, but also virtually impossible to do in a controlled manner.

Here is where the idea of Small Data comes into play. The term came up in different fields, most prominently in marketing (Lindström 2016) or in e-Health systems research (Estrin 2014).

There does not seem to be a final consensus on what constitutes Small Data. The idea is usually characterized in direct opposition to Big Data, for example by data sets that can be generated by a single person (Estrin 2014) or that are within the realm of human comprehension, leading to intuitive insight (Lindström 2016). Decentralization of the data sets also plays a role (Pollock 2013).

In this specific context of interactive music creation, the aspect of smallness also can be extended to direct feedback. While Big Data applications usually need, in addition to hardware, a lot of time to give accurate results, small data sets can be immediately evaluated and a structure inferred within an exploratory, real-time-oriented compositional approach, as is typical for live coding methods.

Furthermore, the idea of smallness can be extended to the model itself, where the observation of the model or rules inferred from the data can lead to further insight, as semantically meaningful models are preferred.

Using small data sets and real-time inference surely won't return 'perfect' (or perfectly predictable) results (e.g., in the sense of a stylistic imitation that can't be distinguished from an original). Given the current state of technology, the representational capacity of the methods proposed in the following are behind of what is currently achieved with deep learning methods and the like. In the context of artistic production, though, this isn't the only criterion. In fact, it might not be all that significant as long as the results are inspiring and artistically valid, with the methods being available to anybody with access to regular hardware.

In some sense the Small Data idea is also reminiscent of the early days of computer composition, when neither the amount of data nor the hardware capacity was anywhere near today's level, and more

[1] It should be mentioned that in recent times some effort has been made to make machine learning technology more widely available, with projects like the Julia language (Innes 2018), or, in the context of music (especially synthesis and instrument creation, as opposed to structure generation), the Nsynth dataset and GANSynth (Engel et al. 2019). Their usefulness in the context of live coding has yet to be evaluated.

domain-specific knowledge may have been needed to achieve valid results. Today's technical world allows us to re-create some of this in real time.

Thus, the idea of Small Data in the given context could be seen as less of a black box approach, but a more human, democratic use of machine learning methods (Pollock 2013), not with the goal of giving accurate predictions or categorizations of the world around us, but rather for artistic inspiration, by means of discourse with the model itself.

Probabilistic Finite Automata

Using Markov Chains of different orders for musical sequence generation is a well-known method in algorithmic composition (Nierhaus 2009). Common Music, for example, provides methods to create sequence generators based on those (Taube 2014).

Probabilistic Finite Automata (PFA) (Ron, Singer and Tishby 1996) are a representation of Variable Order Markov Chains, which combine the predictive power of fixed higher-order Markov chains (where needed) with a more compact memory footprint.

Mathematically, PFAs are described by a 5-tuple $Q, \Sigma, \Gamma, \Upsilon, \Pi$ and a memory length N, where:

- Q is a finite set of states, the states being labeled over $\Sigma^n, n \leq N$;
- Σ is a finite alphabet;
- Γ is the transition function, determining the next state given a current state and an emitted symbol;
- Υ is the next symbol probability function, determining the probability of a symbol

The PFA model isn't necessarily interesting for its novelty, but for its versatility, and in this case, the ability to support a semantically meaningful dialog. The semantic quality (i.e. the capacity for transmitting meaning) of the model becomes clear if you think of the elements of the alphabetas musical events. Thus, the states, their labels and the transitions between them can easily be read in natural language ("after four repetitions of a bassdrum follows a snare with 50

Probabilistic Finite Automata can be used as sequence generators, whether they are inferred from user-provided rules or trained from given sequences of observations. The training and inference algorithms are efficient enough to be used in the context of real-time, on-the-fly composition (as the examples in the following sections will show).

Furthermore, it is fairly straightforward to interact with and manipulate the learned or inferred structures to create variation, either by adding rules or by manipulating the inferred structures directly through predefined operations. Thus, PFAs are a good approximation of the Small Data idea and present a data model that is trainable in the machine learning sense while also allowing for discourse with the model through live interaction.

Especially with limited data sets, the model is somewhat intuitive. Smaller sequence generators (that might nonetheless produce interesting results) can be even written by hand or drawn onto a sheet of paper (Fig. 1). This intuitive accessibility corresponds well to the idea of using Small Data for artistic inspiration.

Findings by David Huron might give hints regarding the usefulness and limitations of the PFA model in relationship to human cognition. As Huron writes:

> In describing conditional probabilities, two concerns are the contextual distance and contextual size. Some states are influenced only by their immediate neighbors (i.e., small contextual distance). Other states are influenced only by states that are far away in space or time (i.e., large contextual distance). [...] The size of the context of probabilistic influence is sometimes also called the probabilistic order. [...]

> As we will see in later chapters, music exhibits a complete range of such dependencies. Most of the time, the principal constraints are of low probability order and involve a near context (e.g., one note influences the next note). But music also exhibits distinctive patterns of organization where distant contexts are more influential than near contexts and the probability order is quite large. (Huron 2006, p. 56)

In the context of live coding, the smaller contexts can be easily represented by the PFA model, as we will see later on, while, due to restrictions regarding processing power and time, the larger contexts are still in the hands of the performer.

Alternative Models

Among alternative models that might fit the Small Data idea, in that they are comparatively human-readable, one might be Augmented Transition Networks, as previously applied by David Cope (Nierhaus 2009), even though based on extensive musical corpus analysis rather than intuitive data entry in a live coding situation.

Generative Grammars or Probabilistic Generative Grammars (Nierhaus 2009) might also be considered, even though they might be more suited to offline sequence generation rather than real-time generation due to the way non-terminal symbols are handled.

The use of these models in the context of live coding needs further research to determine their practicality.

The Mégra System

In Mégra, the mathematical details are transparent to the user; only the details essential to interaction made it to the syntax. Also, the model can be visualized fairly easily, as seen in the code examples and their visualizations 1-4.

The system embodies the Small Data idea by providing a compact and semantically meaningful syntax both for creating sequence generators by hand, to gain a better understanding of the underlying model, as well as for inferring structures from tiny data sets, which can then again be turned into code, visualized, manipulated, and, of course, sonified.

Learning, Inferring and Extending Structures, Making Up Rules

The Mégra system allows for the creation of musical sequence generators on the fly in several ways. One way is to explicitly specify a set of transition rules (Listing 1).

```
;; Code Example I: a simple beat generator inferred
   from explicit rules
;; see visualisation in Figure 1
 (infer 'beat
     (events (x (sn)) (o (bd)) (- (hats))) ;; symbol-
        to-sound-event mapping,
                ;; x = snare, o = bassdrum, - = hats
     (rules
                ((x) - 1.0) ;; hats follows snare,
                   always
                ((o) - 1.0) ;; hats follow bassdrum,
                   always
                ((-) x 0.4) ;; after hats, either
                   have another snare,
                ((-) o 0.4) ;; ... another bassdrum,
                   ...
                ((-) - 0.2) ;; ... or, less
                   frequently, another hats.
                ((- - - -) o 1.0))) ;; after four
                   sequential hihat sounds, always
                   emit a bassdrum
```

Listing 1: Mégra Code Example I, a simple beat generator

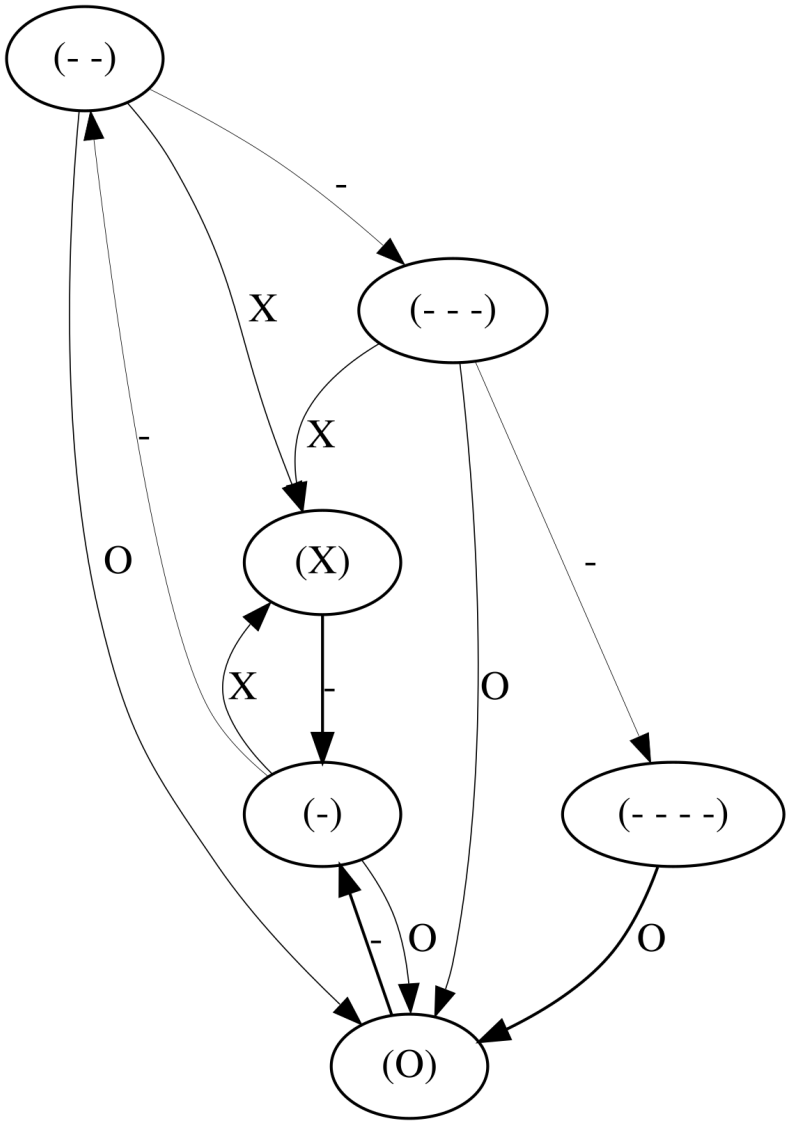


Figure 1: Graphical Representation of Code Example I

Another way is to train a sequence generator from an observed or made-up sequence (Listing 2). Once the sequences' structure has been inferred, it can be modified in different ways. A straightforward way would be to export the learned structure as code, such as in the first example, which is possible, but the amount of code might quickly grow too big to manually edit it.

```
;; Code Example II: a more complex beat generator
    learned from a pattern
;; see visualisation in Figure 2
(slearn 'beat
     (x (sn) o (bd) - (hats)) ;; symbol-to-sound-
         event mapping (shorter syntax)
       "x-o-x-o-x-o-x-o-x-o-x-o-x-o-x-o-x-
        o-x-o-x-o-x-ox-xo-x--ox-ox--xo-x-xo
        -xox-ox-ox-ox-xx-ox") ;; a sample string to
            learn from
```

Listing 2: Mégra Code Example II, Training Generator

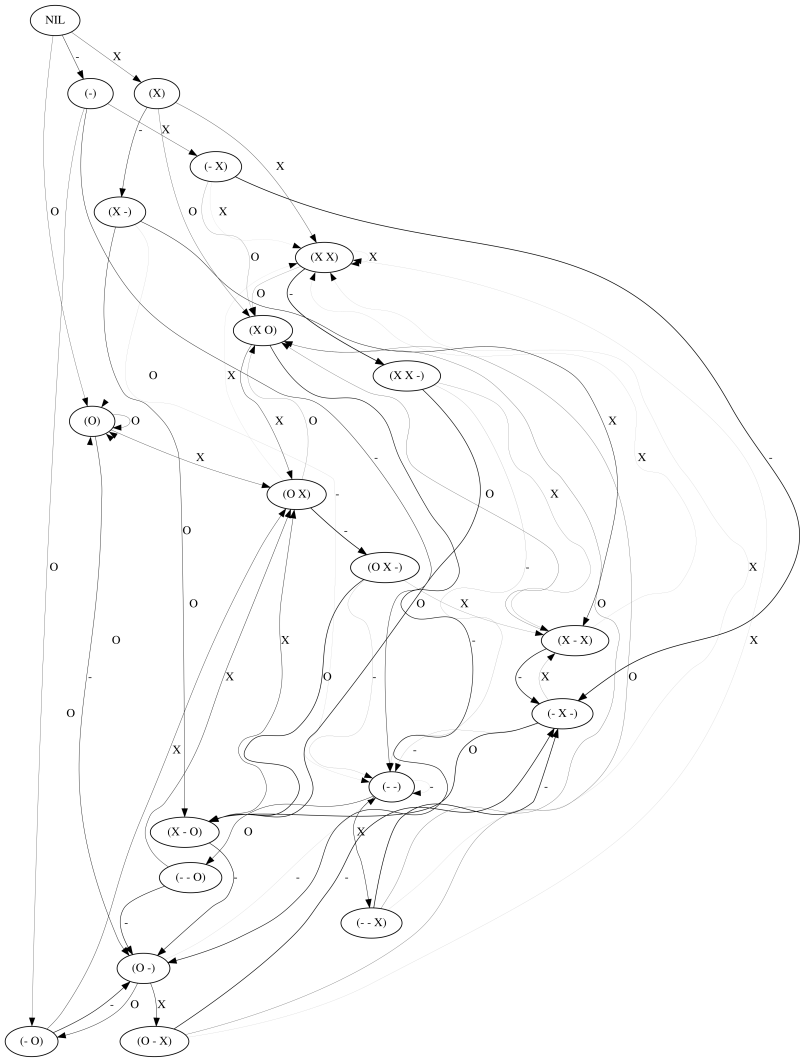


Figure 2: Graphical Representation of Code Example II

Thus, a more manageable interface with the model would be to manipulate the learned PFA by inserting additional states on the basis of predefined operations, and the actual history of emitted events. In that manner, it is also possible to start from a very simple structure (Listing 3 and Figure 3) and 'grow' the sequence generator successively by inserting nodes and edges following certain criteria.

```
;; Code Example III: a very simple starting point, a
     nucleus
;; see visualisation in Figure 6
(s 'the ()
  (nuc 'nucleus (saw 'a2 :dur 102 :atk 2 :rel 100 :
     lp-freq 1000)))
```

Listing 3: Mégra Code Example III, Training Generator

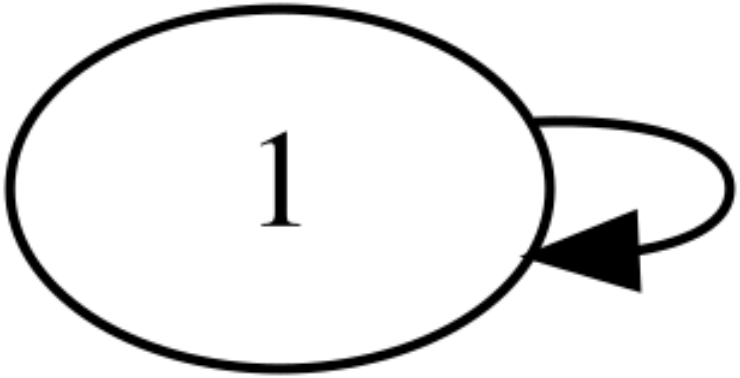


Figure 3: Graphical Representation of Code Example III

The growth operation, as described in the following (Listing 4 and Figure 4) is an example of this kind of interaction with the model. It

will spawn a new node based on the last one that has been evaluated, modify the parameters of the formerly emitted event(s) with an average variance of 0.3, and arrange the new edges in a way that small loops of three musical events will emerge.

```
;; Code Example IV: growth operation
;; successively extend - see visualisation after
    extension in Figure 4

(grow 'nucleus :variance 0.3 :method 'triloop)
```

Listing 4: Mégra Code Example IV - Growth operation

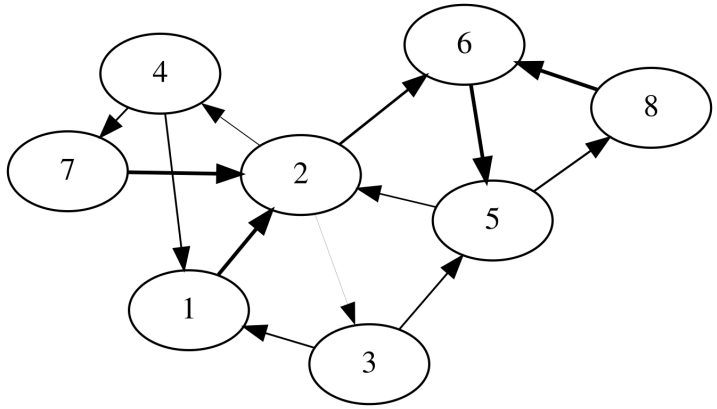


Figure 4: Graphical Representation of Code Example IV after growth iterations

The growth method above can again be automated, e.g. by a simple life-modeling algorithm. This gives each node a certain lifespan after which they perish and spawns new nodes after a specified amount of evaluations in the way described above, tied to the availability of a predefined amount of "resources." This way, the generated generators will stabilize (or perish) after a while, once the resources for further growth run out. If the context requires a more deterministic outcome, the Mégra system also allows for the creation of sequence generators from a more musically oriented description, e.g. from a layered loop (or pattern) syntax (Listing 5), which translates to the same PFA model and thus allows for using the same interactions later on.

3mm

```
 ;; Code Example V: layered loop syntax
(s 'beat ()
   (cyc 'layer2 "hats ~ ~ hats ~ ~ ~ hats hats") ;;
      loop 2
   (cyc 'layer1 "bd ~ ~ bd sn ~ ~ casio:'hi")) ;;
      loop 1
```

Listing 5: Mégra Code Example V

Time Handling

The examples above rely on a fixed time spacing (no explicit time information given), but it is also possible to have explicit time control, with time values specified in milliseconds (Listing 6).

```
 ;; Code Example VI: Same as example 1, with some
    explicit time values
(infer 'beat
     (events (x (sn)) (o (bd)) (- (hats)))
     (rules
                ((x) - 1.0 100) ;; hats follows snare
                    , always, after 100ms
                ((o) - 1.0 100) ;; hats follows
                    bassdrum, always
                ((-) x 0.4) ;; after hats, either
                    have another snare,
```

```
                ((-) o 0.4) ;; \dots another bassdrum
                    , \dots
                ((-) - 0.2 50) ;; \dots or, less
                    frequently, another hats (50ms)
                ((- - - -) o 1.0))) ;; after four
                    sequential hihat sounds, always
                    emit a bassdrum
```

Listing 6: Mégra Code Example VI, a simple beat generator with some explicit time control

Pragmatic Interaction

```
 ;; Code Example VII: Event Streams
(s 'sawtooths () ;; <- This is an event sink.
   ;; ^
   ;; | Events flow in this direction ...
   ;; |
   (prob 30 (rev 0.2))  ;; Modifier! 30% chance to
       add some reverb.
   (nuc 'nucleus (saw 'a2 :dur 102 :atk 2 :rel 100 :
       lp-freq 1000)) ;; Source
```

Listing 7: Code Example VII: Pragmatic interaction by modifying the event stream.

Especially in an exploratory situation it is sometimes helpful to quickly change certain parameters or create some variation by modifying a parameter with a certain probability.

The Mégra system allows for those pragmatic modifications (pragmatic in the sense that they're not necessarily covered by the PFA model) by altering the event stream with certain operators that select and modify the parameters of the passing events (Listing 7), inspired by what is commonly called Reactive Programming, e.g. as described in Maier et al. (2010).

[2]Demo: `https://vimeo.com/321099751`

Technical Foundations

Mégra is built upon Common Lisp as a base language and utilizes SuperCollider for sound synthesis. The two communicate via Open Sound Control.

The Common Lisp language has been chosen for its expressive power and the syntactical freedom it provides. Furthermore, there are several powerful libraries for music creation available, most prominently Incudine (Latini 2019) and an ancient, but functional, version of Common Music (Taube and Finnendahl 2019).

Usage Examples

In the following, two specific usage examples and some more generic remarks about the usage of the Mégra system will be presented.

Creating a Pattern Syntax on the fly

The Mégra system can be used to create small pattern languages on the fly by associating sound events with symbols and entering a sequence as a string[2]. The possibilities can be explored by modifying the string and eventually adding new symbol/sound associations.

The resulting syntax might be somewhat similar to other approaches (think of Gibber (Roberts n.d.) or FoxDot (Kirkbride 2019)), allowing the application of previous pattern knowledge. But, as a non-deterministic sequence generator is inferred, the results might not be totally expected. Code Example II above (Fig. 3) also follows a similar idea.

Simple Language Sonification

Another possible use of this system is to enter a slightly larger, given data sequence and to associate its symbols with sounds, e.g. to sonify text snippets[3] from 'simple' language texts like Toki Pona (Lang 2014).

Performance and Composition

The Mégra system has been continuously used for composition[4] and live coding concerts[5] since its inception, which has been an important aspect in its development process. Especially through the use in a live situation, many inconveniences in syntax and handling have been exposed and successively improved, and it is currently approaching a somewhat stable state. 6 FUTURE OUTLOOK While further simplifying the syntax and increasing expressive possibilities is an ongoing project, a major future goal is inferring the PFA structures not only from code input, but also from audio input. That way, a sequence generator could be created simply by clapping a rhythm, or singing a melody. This has, as a first step, required some audio feature extraction, which has been done by creating Common Lisp bindings for the popular Aubio library. As a next step, a method to transfer the extracted features into the symbolic domain is needed so that they can be used in the same manner as the code-based input.

Conclusions

With Probabilistic Finite Automata, a trainable data model that still allows for versatile real-time interaction and thus, discourse with the model, has been identified and implemented.

From personal experience, engaging in an active discourse with the PFA model, training or inferring sequence generators, keeping or discarding the results and subsequently applying the mentioned operations to transform the results allows for the frequent discovery of non-obvious, yet interesting sequences and sound combinations.

My aim with this project was to engage the audience (and myself as a performer) by creating and exposing a semantically meaningful discourse with a Small Data model through sound and code.

It does need some practice to be uses effectively, and the pragmatic interaction (as described in Section 4.3) is still an important part, especially in the context of performances that require more rapidly-shifting dynamics, such as Algorave. On its own (without much pragmatic interaction), the presented system shines in types of music that unfold more slowly in time.

Nonetheless, having used it successfully in performances so far, the Mégra system continues to be extended and field-tested.

References

Estrin, D. (2014) Small data, where n=me. Communications of the ACM, 57(4), pp.32-34.

Engel, J., Agrawal, K.K., Chen, S., Gulrajani, I., Donahue, C. and Roberts, A. (2019) Gansynth: Adversarial neural audio synthesis. arXiv preprint arXiv:1902.08710.

Fiebrink, R. and Caramiaux, B. (2018) The Machine Learning Algorithm as Creative Musical Tool. In The Oxford Handbook of Algorithmic Music (pp. 181–208). Oxford University Press. `http://doi.org/10.1093/oxfordhb/9780190226992.013.23`

[3]Demo: `https://vimeo.com/321099989`

[4]Demo: `https://ellipsenpark.bandcamp.com/track/hayaoki-ii-raintech`

[5]Demo: `https://www.youtube.com/watch?v=IJPKeKZ6bv0` (w/ Turbulente on visuals)

Huron, D. (2006) Sweet Anticipation. Cambridge, MA, USA: The MIT Press.

Innes, M. (2018) Flux: Elegant machine learning with Julia. J. Open Source Software, 3(25), p.602.

Kirkbride, R. (2019) FoxDot: Live coding with Python. [online] FoxDot: Live coding with Python. Available at: https://foxdot.org/ [Accessed 7 Sep. 2019].

Lang, S. (2014) Toki Pona - The Language of Good. Tawhid.

Latini, T. (2019) Incudine. [online] incudine.sourceforge.net. Available at: http://incudine.sourceforge.net [Accessed 7 Sep. 2019].

Lindström, M. (2016) Small Data. St. Martin's Press.

Maier, I., Rompf, T. and Odersky, M. (2010) Deprecating the observer pattern.

Nierhaus, G. (2009) Algorithmic Composition. Dordrecht: Springer.

Pollock, R. (2013) Forget big data, small data is the real revolution. [online] The Guardian. Available at: https://www.theguardian.com/news/datablog/2013/apr/25/forget-big-data-small-data-revolution [Accessed 8 Sep. 2019].

Reppel, N (2019) https://github.com/the-drunk-coder/megra. [online] Mégra. Available at: https://github.com/the-drunk-coder/megra [Accessed 7 Sep. 2019].

Roberts, C. (n.d.) [online] Gibber.cc. Available at: https://gibber.cc/ [Accessed 7 Sep. 2019].

Ron, D., Singer, Y. and Tishby, N. (1996) The power of amnesia: Learning probabilistic automata with variable memory length. Machine learning, 25(2-3), pp.117-149.

Taube, R. and Finnendahl, O. (2019) Common Music 2.12. [online] GitHub. Available at: https://github.com/ormf/cm [Accessed 7 Sep. 2019].

Taube, R. (2014) Common Music 3. [online] commonmusic.sourceforge.net. Available at: http://commonmusic.sourceforge.net/cm/res/doc/cm.html#markov-analyze [Accessed 11 Sep. 2019].

Roberts, C. and Wakefield, G. (2018) Tensions and Techniques in Live Coding Performance. In Oxford Handbook of Algorithmic Music (pp. 293–317). Oxford University Press. http://doi.org/10.5281/zenodo.1193540

Poly-temporality Towards an ecology of time-oriented live coding

Alejandro Franco Briones
McMaster University
francoba@mcmaster.ca

Diego Villaseñor
Independent Researcher
diego.vid.eco@gmail.com

David Ogborn
McMaster University
ogbornd@mcmaster.ca

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

The current paper traces the development of three platforms for poly-temporal live-coding: Canon-Generator, FluentCan and TimeNot. The platforms rely on concepts and ideas developed by the Mexican-American composer Conlon Nancarrow and are based on an ongoing collaboration around sonic experimentation and time. The paper describes how a process of tensions and resistances have become a productive context for research and knowledge production.

Introduction

The current research has at its core the path opened by the composer Conlon Nancarrow and his life-work on time and music, more specifically poly-temporal composition strategies. This research is an on-going critical reflection that has pushed the authors to think differently about certain problems regarding time, sound, live coding and poly-temporality. Firstly, it was necessary to produce a conception of Nancarrow's work that differs from the main North-American / European narrative, even resist the tendency to assess Nancarrow as an expatriate American composer in order to reframe him as a Mexican-American socialist artist, thus conveying a constellation of implications that go beyond the conventional understanding of Nancarrow's work. Secondly, as access to the artifacts and scores regarding his oeuvre are restricted (particularly for people that are not supported by academic infrastructures), it was necessary to refer mainly to the ideas and notions that Nancarrow produced and that were interpreted and analysed by people like Gann, Murcot, Collins, Thomas, Sandoval, Estrada, among many others. We appropriated these loose sets of ideas and used them as a creative and imaginative starting point. And finally, the conversations kept by the authors of this text rely on the idea of resistance (Franco and Villaseñor, 2018). Resisting each others impulses to dominate the conversation and push forward what emerged from the tension between the arguments.

The output of this research took its first form as the platform

Nanc-In-A-Can/Canon-Generator, a series of SuperCollider classes and functions capable of producing poly-temporality based on the concepts, ideas and strategies of Conlon Nancarrow enmeshed with live coding's different conceptions of rhythm and time. From this point, two diverging parallel paths were enabled. Namely, FluentCan, a SuperCollider extension and notation that offers new possibilities to produce poly-temporal structures in a way that fits the purposes and idiosyncrasies of live coding, and the computational notation TimeNot. The present paper describes the latter in detail, both its notation and its multi-contextual approach which breaks from some core ideas of Nanc-In-A-Can in order to expand upon the expressive capabilities for time-oriented live coding.

The three aforementioned projects critique the infrastructure and time conceptions of live coding communities, seeking to widen the possibilities of their practices. The poly-temporal structures enabled by this notation form a space of resistance that might allow listeners and performers to experience time beyond the scope of accelerated and linear neoliberal subjectivity. The present project is an extensive attempt to produce a mode of performance that emerges from an experience and conception of time as slow, constant, in resistance, multiple, simultaneous, non-linear, digital/analogue, and rhythmic.

Nanc-In-A-Can/Canon-Generator is a SuperCollider library designed to produce temporal canons, like the ones proposed by Conlon Nancarrow (Franco and Villaseñor, 2018). The ideas of Nancarrow are explored in order to create new temporal conceptions within the field of live coding. Notationally Canon-Generator offers an API that has a compositional focus. It requires the performer or composer to define a canon with all the traditional parameters one would expect for it: a sequence for durations and melody, a convergence point, a list of tempos, and also other less conventional options. This means that, on the one hand, the musician has to define every one of these parameters from the very beginning. On the other hand, the musician, upon reading the code, can have a fairly clear idea of what is going to happen.

In contrast, the FluentCan extension for SuperCollider is an API wrapper for Canon-Generator that responds to the necessities of live coding performances. It offers a highly expressive syntax, various strategies for providing default values, and powerful tools for expressing novel musical ideas in the field of poly-temporality..

TimeNot, is a computational notation that is capable of producing, in an expressive way, complex rhythmic ideas embedded in poly-temporal structures. Many relevant aspects of the notation draw its main features, particularly poly-temporal strategies for music creation, from Canon-Generator. The notation encourages performers to project musical ideas further from the present than in the conventional interaction model of live coding.

TimeNot allows the production of tempo canons as in the platforms FluentCan and Canon-Generator but, in addition, it explores new and expressive ways of representing them by complementing the production of canons with strategies to describe other forms of temporal organisation such as global durations and a specific form of rhythm. The notation of TimeNot has been implemented in two different complementary ways with distinct advantages: It is embedded in Estuary (Ogborn et al. 2017) as a mini-language which allows it to engage in ensemble dynamics and be integrated in a rich ecosystem of languages that encourage diverse ways of thinking time and music. It is also an extension of the platform SuperCollider (Wilson, 2011) using its IDE and server taking full advantage of its sound-synthesis power and allowing it to be easily distributed.

Context

The poly-temporality that Nancarrow proposed can be regarded as highly algorithmic; in it, temporal and pitch mapping functions are the basic principle. This means that the melodic material of any given music work (which consists mainly of pitch and duration series) can by transposed into any tempo or any pitch register. Nancarrow developed the concept of a convergence point (CP) as a way of organizing this intricate musical material. The CP is a point in time in which the formal and the chronological temporalities of a musical idea are

identical. According to Thomas (1999), this strategy allows us to listen different timelines moving towards the same point in chronological and formal time. Given that a poly-temporal canon can be generated very easily by algorithms, it is a musical strategy that can be implemented in computational settings with relative ease. Collins (2003, pp 1) describes a compositional system capable of producing various kinds of tempo canons, a precedent for this notation. More recently we designed the software Canon-Generator (Authors, 2018). Canon-Generator provides a programming notation that allows musicians, artists, programmers and other creative users of code to create multiple and simultaneous sound timelines based on the work of Conlon Nancarrow. This software was developed in SuperCollider (McCartney, 2002) because of its powerful computational and synthesis capabilities and its extended use among live coding practitioners in Mexico City. Listing 1 allows the user to create a major scale in three different tempos that converge at the fifth event (G, midinote 67).

```
Can.init;
(
// convergence canon;
~conv = Can.converge(
    melody: Can.melody(
    [8,8,8,8,8,8,8,8].reciprocal, // 1/8 rhythmic
        figures
    [60,62,64,65,67,69,71,72]),
    cp: 5,
    voices: Can.convoices([50,72.5,75],[-12,0,12])
);
~conv.visualize(s)
);
```

Listing 1: Canon-Generator

What we find more interesting are the "human-to-human" communication aspects of this notation, including but not limited to the cognitive dimensions of computational notation (Green, et al. 2001). This piece of code is not only role-expressive (in Green's terms) but also brings to the foreground a series of cultural references that locate the user of the program within a cultural and socio-political framework: the work of Conlon Nancarrow and its most salient temporal strategy to establish poly-temporal musical structures: Tempo Canon. The word 'can' also makes reference to the rigid steel or tinplate container normally used to preserve food which often is related with vulgar or industrialised products and, in this case, cheap nourishment for survival. This notation attempts to re-contextualise and re-appropriate musical strategies often related to a privileged music elite and academic activity, that claimed the ideas of Nancarrow as part of their tradition even though the Mexican-American socialist artist required an untraditional context to flourish (Franco and Villaseñor, 2018). The canon generator, by distributing it in communities away from the people that often control the canon of the music academy, allows new meanings for the work of Nancarrow.

The notation style of Canon-Generator is heavily embedded in SuperCollider's inherent notation and it has a limited scope. There are some aspects that do not facilitate live coding performance, for instance: the nested parentheses very particular to SuperCollider. SuperCollider language is not an expressive notation for rhythm as time, in this context, is mostly represented through inter-offset durations, namely wait patterns as understood in SuperCollider's Pattern Library (Harkins, 2009). This action produces a wait pattern, which is expressed as duration, obfuscating its meaning and compromising role-expressiveness. A more transparent use of the duration could refer to the total length of a musical idea (which cannot be expressed easily in the SuperCollider Pattern Library) or the length of a sound event (which is expressed as legato). Rhythm is more intuitively described in terms of offsets and onsets over an underlying grid, which can hardly be inferred by the duration value described in SuperCollider Pattern library. It also requires a binary distribution to be installed, which is an impediment in some settings. From this starting point a need for a new notational system that might be capable of representing the novel and compelling temporal forms already

made possible by Canon-Generator was identified. At this point the endeavours of the two principal authors of this platform diverged: Diego Villaseñor implemented FluentCan as a wrapper and extension of Canon-Generator while Alejandro Franco developed the notation of TimeNot.

Diverging Paths

FluentCan

FluentCan responds to one of the original premises of Nanc-in-a-Can/Canon-Generator: to design a live coding interface and notation that can be self-contained in SuperCollider. This has allowed its integration to a broad ecosystem of software development and practices; one with strong communities worldwide and which has been particularly active in Mexico City (Nancarrow's creative home).

Conceptually FluentCan is a direct response to the particular necessities of live coding practices, where a flexible, composable and expressive notation is often desired. Flexibility means that the order of parameters in the definition of a canon, or the fact that some of them may be missing, should not be tremendously important: behind the scenes, the parameters should be put in the correct order and default values should be provided. Composability suggests that small ideas should be able to gradually grow and transform through the course of the performance. And expressivity means that it should be easy to generate abstractions, be they of canons or of transformations of canons (for example, allowing the musician to easily use one canon as a prototype for another).

FluentCan takes its name from the so called "fluent interface" technique of object oriented programming. Using this technique it is possible to wrap the data driven and immutable Canon-Generator API into a notation that can make use of method chaining to build musical ideas. This simple wrapping offers not just the ability to write canons in a more efficient way (by providing defaults), but also allows for inheritance (canonB may inherit it configuration from canonA) and offers a whole range of methods to transform inherited and non-inherited data, the most radical of which is .apply (described below).

From a conceptual and processual standpoint, it has allowed Canon-Generator to enter into a resistant dialogue with TimeNot: because TimeNot and FluentCan are, so to speak, livecoding native notations, they have allowed their authors to exchange ideas in a fruitful dialogue of shared interests.

Syntactic simplicity

Because the new interface provided by FluentCan does not require the user to provide all the parameters that Canon-Generator needs, it allows them to take a simpler approach for making music. This means the user can now use FluentCan for making non-canonic structures, that are effective both syntactically and musically.

```
// Isorhythmic single voice sequence, 7 notes within
    2 seconds. So called 'color' comes from notes, '
    talea' from durs

Can.init;
c = FluentCan(\can1).notes([60, 62, 67]).durs([2, 3])
    .period(2).len(7).play;
```

Listing 2: Code Example 2

This idea can then be easily extended into a temporal canon like so:

Listing 3: Code Example 3

```
// Converts into a 2 voice canon with lower voice a
    perfect fifth below.
c.transps([0, -7]).tempos([2, 3]).play;
```

In this sense, now it is idiomatic to express non-canonic ideas while at the same time providing the means for the music and musicians to project these ideas into a poly-temporal dimension.

3.1.2 The .apply case Throughout the TimeNot development process, a notation for expressing rhythm called xo notation was developed; a notation that is similar to ixilang (Magnusson, Thor, 2011) and Gibber (Roberts, Charlie & Kuchera-Morin, JoAnn, 2012) in which graphic notation takes an important role. At its core, it consists of a string of x's and o's (i.e. xooxooxo) which serves to determine whether an event should sound (x) or not (o). The concept of the notation itself is conducive to different interpretations and variations. This has been the case not only between TimeNot and FluentCan, but also within FluentCan itself. Triggered by the effervescent dialogue with the TimeNot philosophy, for FluentCan it was chosen to allow any implementation of xo notation to be possible and easy to use. This was the main driving force behind the development of the .apply method.

The .apply method is a generalization of the necessity to allow for user generated functions to be used at runtime. The function itself does nothing more than applying the built up canonic data structure (inside the FluentCan instance) to a function that returns a new FluentCan instance (in Haskell style type notation it could be easily described as FluentCan -¿ FluentCan). This method can be called multiple times so that the effects of these calls are composed (see example below).

The class IsoFluent is one that provides static methods (as pure functions) which fulfill precisely this interface. Because Canon-Generator can take functions for transposition[1], the method IsoFluent.xo creates a transposition function that iterates over it's melody, for each voice of the canon (a MidiNote array). Using modulo arithmetic it returns a new melody with rests in place of a o's and the input melody notes in place of the x's, or when given a number (that corresponds to the voice index) it returns the input note if the index voice corresponds to the given number or a rest if not.

Through .apply FluentCan effectively extends Canon-Generator in a possibly infinite number of ways. It gives the performer the ability to create and modify their own functions (at compile time or on the fly). Even more so, it opens Canon-Generator and the exploration of time to the communal imagination. Now libraries that follow this interface can be independently developed, each of which may explore different ideas that nevertheless compose with any other ideas coming from the community.

```
Can.init; Can.defaultServerConfig;s.boot;

// Models
m = FluentCan().period(1.5).len(5); // model
n = FluentCan().period(2).len(4); // model 2
o = FluentCan().transps([0, -7]).tempos([1, 1/2]).
    period(2); // use this to create canon, tempos
    1/2 and 2 work well

//custom function for .apply
(
~aksakish = {|...nums| // input any sequence of
    numbers
         var aksakXos = nums.collect({|n| // make a
             list of x's and o's
         n.collect({|i| if(i == 0, {"x"}, {"o"})});
    }).flatten;
         // return a function that will receive the
```

[1]For transposing Canon-Generator takes an array of values on the key transp. Each value corresponds to the transposition of the voice that has the same index as the value. transp takes either an array of numbers ([Number]) or an array of functions ([[Number] -> [Number]]) or a combination of numbers and functions. When provided a Number, it simply maps the melodic sequence array over a function that adds the given number; if the value is a function [[Number] -> [Number]], it maps the melodic sequence over it.

```
            FluentCan instance
        {|fluentCan|
                // do whatever with the instance
                fluentCan
                .len(aksakXos.size)

                                .apply(IsoFluent.xo(
                                    aksakXos));
        }
}
);

// it is possible to switch from the different models
     (m, n, o)
a = o.def(\1).notes([65, 62]).apply(IsoFluent.xo("
    xo01o")).cp(2).len(7).play;
(
b = o.def(\2)
.notes([70, 77])
// 'applyable' functions can be composed:
.apply(~aksakish.(7, 5, 3))
.apply(IsoFluent.xo("10")) // voice 0 and 1 alternate
.cp(3)
.play;
)

(
c = o.def(\3)
.cp(4)
.period(6.5)
.notes([65, 62, 67, 75])
.len(15)
.apply(IsoFluent.xo("o1ox0xo"))
.play;
)

a.pause;
b.pause;
c.pause;
```

Listing 4: Code Example 4

TimeNot

TimeNot was created in a specific context: at McMaster University, and as a major research project (MRP) by Alejandro Franco in the MA program in Communication and New Media. Moreover, its development has been influenced by the activity in the Networked Imagination Laboratory (NIL), a space in which the concept of network is explored broadly and often in relationship with live coding practices. One notion that networked imagination conveys is the understanding of artistic creation and research as an ecology of — among many other things — natural and computational languages, artistic styles, techniques, technologies, strategies, etc. In other words, in the NIL a heterogeneous set of media is articulated, hacked, developed and explored in order to produce aesthetic and intellectual works. The premise that creation depends on an ecological approach to knowledge resonates with Mexico City's live coding scene (where Alejandro Franco developed many of his artistic practices), where collective creation and assemblages of heterogeneous artistic practices are often prominent. The main idea targeted by this notation is the tempo canon. However, what is particular about this exploration in the context of the poly-temporal ecology here explored is the many additional temporal and rhythmic strategies and techniques that can be notated easily.

The author opted to create a new, independent, software project based on the ideas of Canon-Generator so the notation could be adapted to many contexts. TimeNot is now available in two forms: within the Estuary platform, and as a SuperCollider extension. Estuary (Ogborn et al., 2017) is an experimental software that can be defined as a platform for learning and creating, and using live coding as a main strategy that favours a multilingual approach, and which makes live coding languages available on a zero-installation basis, like

Gibber (Roberts & Kuchera-Morin, 2012), LiveCodeLab, and Hydra. SuperCollider (McCartney, 2002) has a powerful audio synthesis engine as well as a well established community of users around the world. Both platforms respond in different ways to the author's cultural and social context; these are the best ways to give access both to a broad, general userbase as well as to key developers and more specialised audiences. There are other positive aspects of this decision; the separation of TimeNot and Canon-Generator give room for Diego Villaseñor to develop the notational style embedded in SuperCollider that would become FluentCan.

```
|. 4s .| xxxxxxxx
ra: 4:5:6 tr: 0|12|24 cp: last
synths: saw sqr tri pitch: 60 62 64 65 67 69 71 72
```

Listing 5: Code Example 5

Listing 5 reproduces the musical scale of the previous Canon-Generator example and presents a program that exemplifies all the main possibilities that TimeNot allows: a sequence of global durations and a rhythm arrangement in line one, a canonic configuration in line two, and a configuration of instruments and pitch in line three. The interplay among these four components produces rich poly-temporal sonic textures. When this example is executed it produces a series of musical events extending from the moment of evaluation into the future (a C major scale repeated in three octaves with the temporal proportions of 4:5:6 over a total duration of 4 seconds).

Strata

The architecture of the TimeNot notation is organised into four main strata or sub-notations. Three of these allow the user to organise sound events in time. The last stratum aids the organisation of sound parameters.

The three time-oriented sub-notations reflect three main temporal strategies that imply a heterogeneous understanding of temporal relationships. The first one allows the user to embed the sound output in a specific overall duration; this allows the user a very intuitive degree of control over the music material. The second sub-notation provides an expressive and comprehensive way to create rhythmic structures; this aspect of the notation was identified as the one that diverges the most from the possibilities endemic to SuperCollider's notational style. The third sub-notation is the stratum in which rhythmic ideas are transformed into a canon. This is the part of the notation in which the tempo canon ideas developed by Nancarrow are put into practice. The last sub-notation provides a simple syntax to invoke instruments and organise its parameters into different kinds of patterns.

Duration Notation

Having control over the global duration of the canonic/rhythmic structure helps to achieve an overall view of the result. This stratum represents time as a succession of events that do not favour detailed categorisation or differentiation among its internal components. Time here corresponds to the concept of durèe (Bergson, 2002) allowing users to produce simple sequences of events producing an ever-going sense of becoming.

The duration of the rhythmic/canonic structure is determined by a number followed by an s that represents seconds. However, the number could represent beats in a given tempo by adding a t or indicate a certain amount of cycles (cps) by adding a c. These multiple ways of expressing duration respond to the multi-contextual nature of this notation.

To determine a duration the number should be embedded in a special list that uses symbols and separators. The lists that uses the |: :| as a delimiter and the | as an internal separator generate an infinitely looping musical idea. An unlooped event is delimited by |. .|. A finite number of repetitions can be expressed with the symbol % followed by whole number that determines the number of repetitions. If the

duration of the structure is omitted, the default is a 2 second event.

Rhythmic Notation

The rhythmic aspects of the notation can be used independently from the canonic ones. Nevertheless a minimal rhythmic idea has to be written for the notation to produce sound. An example of a comprehensive use of this rhythmic notation might be the idea in the following code example, in which an opening idea is presented in 5/8 manually introducing rhythmic onsets and offsets, then it is concatenated to an Euclidean pattern representing a tresillo (ie. the Euclidean rhythm that distributes 3 attacks over 8 slots), over a two attack onset pattern that is repeated two times. Finally, another 5/8 idea is manually expressed to close the musical idea.

```
|. 8s .|
xoxxo !3:e:8 p: xx #2 xooox  samples: hibongo
```

Listing 6: Code Example 6

Onset Patterns

Notating minimal rhythmic ideas as onset patterns has two main advantages; it provides a declarative materiality to the notation, and it frees the user from being limited to algorithmic structures (such as Euclidean rhythms), allowing arbitrary organisations that do not respond to explicit logical or computational patterns.

```
xxooooooo xxooooooo samples: hibongo
xooxooxo ooxoxooo samples: cabasa
```

Listing 7: Code Example 7

Patterns are notated with the characters x and o, where x denotes an attack and o denotes a rest. In the following example, the first pattern produces two attacks followed by 6 rests and the second pattern produces the Cuban clave rhythm. The following expressions should be evaluated one at the time, to evaluate both add a ; at the end of the first expression.

Repeat Patterns

Repetition allows the user to produce meta-metric cycles in which the same idea is presented until a variation marks the end of a period. This can be achieved because the onset patterns and the repeat patterns can be composed together to express a single musical idea. With the symbols ! and # we can indicate how to repeat a pattern. It is possible to create nested ideas within this language, such that a repeat pattern can contain an onset, repeat or Euclidean pattern. After the ! the onset, repeat or euclidean pattern to be repeated should be written and after the # a value that represents the number of repetitions. The first line in Listing 8 can be simplified as the example shown in the last section.

```
!xx!o#6#2  samples: hibongo
```

Listing 8: Code Example 8

3.2.1.2.3 Euclidean Patterns TimeNot's Euclidean Pattern subnotation is a very expressive and complete tool for rhythm which can integrate embedded onset, repetition and other euclidean patterns. The non-optional values to be given are n and k values as specified by Godfried Toussaint (2003). The Euclidean algorithm produces patterns of distribution of integer numbers as evenly as possible. Given n time intervals and k impulses, this algorithm provides a simple way to distribute the k impulses over the n time intervals. These patterns are found in many forms of music, particularly "in sub-Saharan African

music, and world music in general" (Toussaint, 2003, pp. 1). The syntax proposed in this notation is k:e:n, a number representing impulses and another representing intervals. The first line of the following code example generates a Cuban tresillo and the second generates a clave that is the combination of a tresillo concatenated to a 4/4 measure with onsets only in the second and third beats.

```
3:e:8 samples: hibongo
3:e:8 oo xo xo oo samples: hibongo
5:e:8:r:4 p: xx  samples: bd
```

Listing 9: Code Example 9

The k and n values are chained by the operator :e:;There are two other ways to manipulate the structure produced by this sub-notation: :r: which creates a rotation value, wherein the pattern produced will maintain its same number of impulses and same intervals but will appear starting from a different point of the structure. The operator p: produces a structure that can be a repeat, an onset or another euclidean. This pattern becomes the k value and the non-k values of the time intervals, a series of offsets occupying the same length as the pattern are produced, if k is xx the non-k will be oo, as in the example above

Canonic Notation

A fundamental difference between the tempo canons produced by the earlier Canon-Generator project and those produced by TimeNot concerns the minimal components of a tempo canon. Conlon Nancarrow was restrained by the media at hand: the player piano. The player piano can not play two or more notes that are the same, in the same register, at the same time. To perceive distinct tempos it was necessary to differentiate the voices of the canon somehow. This was achieved by pitch transposition. In TimeNot, where an alternative mechanism to differentiate voices is provided, pitch transposition is not as fundamental. In TimeNot pitch transposition is not a mandatory parameter for canonic transformation. Differentiation of voices is acquired by providing a straightforward mechanism to select instruments. Different voices can be identified by different timbral qualities instead of, or in addition to, pitch and register. Canons can be thought of as based on samples. In TimeNot, pitch transposition is optional in the case of pitched instruments and, in the case of unpitched (sample-based) instruments, it is incompatible. In this way, less typing is needed to produce a "canonic transformation" and the idea of a tempo canon becomes even more focused on temporal aspects of the musical ideas.

The model that is implemented in TimeNot so far is what Nancarrow would have called a "convergence canon", that is to say a canon with only one CP and without tempo change per voice.

In TimeNot, the number of voices per canon is decided by the number of values given to the argument ratio. Ratio is a list of proportions which can have a corresponding list of transpositions that 'canonise' the rhythmic structure.

```
|. 35s .| xoox!5:e:8#5xxoxoooxx
ra: 13:17:20 tr: 0|5|8 cp: last
synths: saw sqr tri
eu pitch: 60 67 65 68 72 75.5 67 55 56 59 60
```

Listing 10: Code Example 10

Sound Notation

Sound notation allows the performer to parametrise the sound aspects of the program. At the moment, there are 5 non-temporal aspects that can be arranged: instruments, pitch, amplitude, panning and out bus. If a list of instruments is provided, it will be distributed as one instrument per voice of the canon. The rest of the sound parameters are organised as sequences that will be replicated exactly in each

voice of the canon. This can be arranged in two different ways: as an isorhythmic configuration, in which each value is assigned from left to right until all rhythmic events have been exhausted, or as a Euclidean distribution that distributes evenly k sound values over n rhythmic events.

Conclusions

With regards to FluentCan's .apply method, it is important to remark that it's development nevertheless is the direct result of the contrapuntal dialogue between the authors of both libraries. The original necessity for this method was to use it to implement a type of xo notation (as proposed by TimeNot). However its generality was able to exceed the requirements of this notation (as explained in 3.1.2). The result was a powerful opening up of both FluentCan and Canon-Generator to any number of possible extensions and new ideas coming from anyone interested from the SuperCollider and Live Coding communities. These results (conceptual, social, and technical) even if tangential in their surface relationship with TimeNot are strongly linked to it by the network of ideas underlying the conceptual space that has become Nanc-in-can. Conversely, TimeNot derived its duration sub-notation from the notion of period that is a product of the FluentCan development. In an interesting parallel with the properties of temporal canons, namely, their tendency to continuously diverge and converge (from which unforeseeable textures emerge); the process so far followed by the authors has spawned a critical and creative flux of independent-related ideas that can further find convergence points in the softwares so far developed or in new ones to come.

TimeNot pushes to its limits the notion of permeable autonomy. This means that it is part of multiple processes that consume it but not in its totality. Neither the possibilities enabled by Estuary, the algorithms inherited from Canon-Generator nor the dialogue that it maintains with FluentCan are sufficient to explain what TimeNot might actually be. Although TimeNot stands as an autonomous software, it is only possible for it to exist as the outcome of the multiple explicit and implicit conversations that happen around it. In this sense, its boundaries are unclear at the same time that its identity remains unambiguous. As this context can be identified as relevant to the development of this software a similar pattern can also be identified in the way its sub-notations are organised. Moreover, it also stands as a valid reflection on the way the multiple concepts of time interplay: Poly-temporality, rhythm and duration might be three ways of explaining time that presents three different perspectives of the same phenomenon. The ecology that surrounds the development of TimeNot is also the form that the notation takes and it is, as well, the way in which it allows artists to think about time: as an ecology.

The authors of the libraries here described, FluentCan and TimeNot, have paid particular attention to the way that researchers relate with knowledge. The concept of co-creation used by Donna Haraway (2015) seems to be relevant here. Co-creation here implies that knowledge is the product of conversations and exchange of ideas, but in a way that resistance and tension is not inhibited but foregrounded.

Forthcoming (Divergent) Convergences

Our exploration of the multiple temporal dimensions in music has begun to bring forth fundamental questions:

1. How is time and poly-temporality represented in other domains?
   a. What particular dimensions of time are exposed by each of these domains?

2. How can the calculations involving time become as flexible, free and liberating, as their results?

   - In purely technical means, how can poly-temporal expressions evolve beyond the current strategy of precalculating each event in advance, into a completely live interaction that would allow us to interact with time and its effects in full real time?

The first set of questions beg for platforms other than SuperCollider and the TimeNot language. These platforms need to be powerful enough to deal with disparate mediums such as audio, graphics, web, poetry, etc; and should also be capable enough to deal with the larger scale architecture that dealing with multiple media requires. At their core, such questions requires a focus on pure time and an architecture that can route time events into different media effectuations.

The second question demands a disassembling of our current notions for constructing time oriented structures. The diverse elements that conform these structures must become independent of each other. Each one must be able to change without needing to stop or restart the temporal flow of events.

This new perspective and aspiration takes its cue from the experience in divergent development. What has been experienced so far is a fertile inter-fluence of independent elaborations over a shared conceptual environment, a truly evolving ecosystem made up of experiences and ideas about time. The first degree of divergence exposed here, two programs diverging and exchanging ideas, is to be interiorized into multiple divergences within a single convergent ecosystem-platform: now the mediums and the parameters of pure time must be allowed to diverge as well. The future platform should be simultaneously divergent, which means: no single point of focus, no single dominant idea or way of doing things, etc; and convergent, that is to say, composable ideas feeding into each other, multiple mediums expressing events organized by singular and multiple polytemporal instances and free routes for interpretations that expose the nature of time.

References

A.F. Blackwell, C. Britton, A. Cox, T.R.G. Green, C. Gurr,G. Kadoda, M.S. Kutar, M. Loomes, C.L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, R.M. Young (2001). Cognitive Dimensions of Notations: Design Tools for Cognitive Technology . Proceedings of the 4th International Conference on Cognitive Technology. Coventry, UK.

Bergson, Henri (2002). Concerning the nature of Time in Henri Bergson: Key Writings. Edited by
Ansell Pearson and John Mullarkey, London: Continuum.

Collins, Nick (2003). Microtonal Tempo Canons After Nancarrow/Jaffe. Proceedings of the International Computer Music Conference, Singapore.

Franco, B and Villaseñor, D. (2018). Nanc-in-a-Can Canon Generator. SuperCollider library for generating and visualising temporal canons critically and algorithmically. Proceedings of the International Conference of Live Coding, Madrid.

Haraway, Donna; Kenny, Martha (2015). Anthropocene, Capitalocene, Chthulhu. Art in the Anthropocene. Davis, Heather and Turpin Etienne. Encounters Among Aesthetics, Politics, Environments and Epistemologies. London, UK: Open Humanities Press

Harkins, James (2009). A Practical Guide to Patterns.
http://distractionandnonsense.com/sc/A_Practical_Guide_to_Patterns.pdf Last accessed: 12th August, 2019.

Magnusson, Thor (2011). The IXI Lang: A SuperCollider Parasite for Live Coding.

McCartney, James (2002). "Rethinking the Computer Music Language: SuperCollider." Computer Music Journal 26 (4). MIT Press: 61–68.

Ogborn, David; Beverley, Jamie; Navarro Del Ángel. Luis; Tsabary, Eldad; McLean, Alex. Betancur, Esteban (2017). Estuary: Browser-based Collaborative Projectional Live Coding of Musical Patterns. International Conference on Live Coding (ICLC) 2017.

Roberts, Charlie & Kuchera-Morin, JoAnn (2012). Gibber: Live coding audio in the browser. 64-69.

Thomas, Margaret (1999). Nancarrow´s Temporal Dissonance. Intégral 13.

Toussaint, Godfried (2003). The Euclidean Algorithm Generates Traditional Musical Rhythms. School of Computer Science, McGill University Montréal, Quebec, Canada.

Wilson Scott; Cottle, David and Collins, Nick (2011). The Supercollider Book. The MIT Press.

Liveness, Code, and DeadCode in Code Jockeying Practice

Jamie Beverley
University of Toronto, McMaster University
jbeverley@cs.toronto.edu

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

This paper explores notions of 'liveness' and 'code' to situate emerging Code Jockeying (CJing) practices in reference to Live Coding and blank-slate improvisation. Chun's (2008) notion of 'source code as fetish' and theory on slow design are employed to critique compulsions in Live Coding towards code, liveness, and ephemerality. It is suggested that CJing evokes different manifestations of liveness and code, perhaps deviating from celebrated blank-slate approaches. The DeadCode (Dead) CJing software is presented; a browser-based, tablet-friendly, and language-agnostic CJing interface that supports improvised mixing of pre-written code ('deadcode'), different code projections for audience members and performers, and gestural controls. Dead's divergence from both liveness and normative representations of code are examined, inviting criticism as to whether the use of Dead still constitutes 'Live Coding'.

Introduction

Both 'liveness' and 'code' are celebrated components of Live Coding which have provided useful for situating Live Coding within a broader New Media landscape. Liveness in performance has been closely associated with improvisation, as Magnusson (2014, p. 22) articulates in stating that the "default mode of live coding performance is improvisation". Blank-slate live coding, in which the performer begins from a blank interface and develops their performance from scratch (Collins and McLean 2014), is one mode of performance that especially exhibits livness and improvisation. Such practices have been framed in distinction from other forms of computer-mediated audio-visual performance that have been critiqued in Live Coding literature for being overly contrived. For instance, Parkinson and Bell (2015) distinguish the 'we all hit play' performance practices of electronic dance music producer deadmau5, from those of free improvisation jazz guitarist Derek Bailey and argue that Live Coding more resembles the latter.

Earlier literature has also differentiated Live Coding from other

forms of computer art that rely on conventional Digital Audio Workstations (DAWs) and other graphical user interfaces (GUIs). Collins et. al (2003) suggest that part of the reason Live Coders gravitate towards code rather than DAWs such as Ableton Live or Reason is to escape the rigidity of such "fixed interfaces" in favor of more improvisational, live, and perhaps risky tools. Blackwell and Collins (2005) similarly contrast Ableton Live with the ChucK (Wang et. al 2015) Live Coding language. More recent research has explored the continuum between highly graphical interfaces and text-based programming languages, augmenting GUIs with scripting capabilities, and vice-versa (e.g. Gibber (Roberts and Kuchera-Morin 2012), Gibberwocky (Roberts and Wakefield 2016), and Visor (Purvis et. Al 2019))

The Live Coding practice of sharing one's screen aims to achieve transparency and performance theatrics through both liveness and code. Code provides a declarative expression of the performer's creative process, while its projection serves to convince an audience of its live composition. The TOPLAP Manifesto Draft (TOPLAP 2010) suggests that this composition of liveness and code distinguishes Live Coding from more obscurant performance practices, in which the audience cannot be confident that the performer has not just hit play on a pre-recorded set and is checking their emails. Moreover, as Rohruber et. al (2007) articulate, the practice of sharing one's screen in live performance can facilitate "public reasoning" between performers and audience members over processes that are normally deliberated behind the closed doors of a studio. While compulsions towards both 'liveness' and 'code' have motivated exciting new forms of expression that challenge and build upon previous forms of computer-mediated art, both performance qualities can be critiqued in light of the established and continually evolving practices of Live Coding. This paper explores some of these critiques and situates Code Jockeying (CJing) as a hybrid practice that combines elements of Live Coding with other audio-visual performance practices that were once framed in opposition to (or to some extent separate from) Live Coding. Such opposing practices include dependency on pre-scripted content, heavy reliance on graphical software abstractions, and other deviations from 'liveness' and 'code'.

The proceeding section employs Chun's (2008) conception of code as fetish to challenge practices that position source code at the center of Live Coding. The temporal politics of liveness, speed, and ephemerality in Live Coding are then likened to the precarity of the entrepreneurial 'gig economy'. The DeadCode (Dead) CJing software is introduced and examined, indicating how it responds to some of the critiques presented while also perhaps diverging from the principles that initially distinguished Live Coding from other forms of computer-mediated art.

Live Coding and Code Fetishism

In 'On "Sourcery," or Code as Fetish', Chun (2008, p. 301) critiques how "software has recently been posited as the essence of new media, and knowing software as a form of enlightenment". For instance, speaking in reference to the free and open-source software movement, Chun (2008, p. 303) highlights how "insisting that freedom stems from free software ... amplifies the power of source code, erasing the vicissitudes of execution and the structures that ensure the coincidence of code and its execution". Moreover, Chun (2008, p. 311) cautions how "we "primitive folk" worship source code as a magical entity-as a source of causality- when in truth the power lies elsewhere, most importantly in social and machinic relations".

Live Coding practices that (at least superficially) organize around code can be critically examined under Chun's insights. Chun might caution against allowing the software employed in Live Coding performance to stand-in for the motivations that have evolved around Live Coding. This sentiment is also reflected in the TOPLAP Manifesto Draft (TOPLAP 2010) which states that "Live Coding is not about tools".

The practice of sharing one's screen in Live Coding performance can be dissected to illustrate Chun's cautions. In Live Coding performances, the projected code is often a central focal point. A literal

interpretation of 'show us your screens' could allow a performer to claim transparency and liveness on the basis that they presented a flood of text too small for an audience to interpret. Code fetishism risks equating adherence to the medium of code (and it's projection) as Live Coding; as transparent, live, theatrical, cutting-edge, and experimental.

Conversely, code can also be employed obfuscate and mask hidden operations and intentions. High-level abstractions and 'impure' functions that have many side-effects may present to an audience as a single operation (what would Live Coding's equivalent of lip-syncing look like?). To this point, Chun (2008, p. 315) adds that "we know very well that source code is not executable"; source code's transition to action contains steps of translation, both social and technical, that are erased when source code is perceived as immediately executable.

In the TOPLAP manifesto, "show us your screens" is preceded by "obscurantism is dangerous" (TOPLAP 2010), suggesting that at least part of the reason code is projected is to provide transparency. Technocratic obscurantism behind source code that is deceitfully presented in the name of transparency has the potential to be more obscurant than admitting to "just hitting play" (Parkinson and Bell 2015) on a pre-mixed performance. In sum then, interfaces that glorify or aestheticize the complexity of code may reify the performer's 'sourcery' and the relations of knowledge-power that Live Coding seems vested in disrupting.

Liveness, Emphemerality and Speed

The venerated qualities of liveness, improvisation, and speed in some Live Coding performance settings can also be re-examined in light of critiques of the accelerated temporal qualities of technology. Rosa (2013) identified a paradox in which accelerations in technological progress have been accompanied by parallel accelerations in the perceived 'pace of life'. Information communications technologies that promised to provide conveniences that would free up one's time instead impose new temporal demands, manifested in ubiquitous connectivity. Time that is relinquished by modern technologies is often expected to be redirected towards "displaying competent – which is to say profitable – subjectivity" (Gregg 2018, p. 188). Rosa (2013) argues that tendencies towards speed are driven by cultural, economic, and structural motors.

In response to social acceleration, critical theorists and technologists have compelled further appreciation for, and exploration of 'slowness' and its qualities. Walter Benjamin (1983) offered the icon of the flâneur; an urban roamer who juxtaposes their inherent appreciation of their activity against their busy industrial backdrop. Compulsions for slow interaction have also manifested in the philosophy and design of slow technologies. Strauss and Faud-Luke (2009) advocate for a form of 'slow design' that facilitates reflection, sustainability, and other qualities beyond utility. Hallnäs (2015) similarly articulates slow technologies as artefacts of expression that reveal their profundity through use, rather than the functional role they fulfill. Odom et. al (2012), Odom (2015), and Grosse-Hering et. al (2013) have employed principles of slow design in the creation of new technologies that counterbalance the perceived accelerations brought by ubiquitous media. Taken together, these works identify a potentially harmful social (or market) preference trending towards efficiency and speed, and propose designed slowness as one response.

In alignment with these slow philosophies, "Slow Coding" (Hall 2007) has been suggested as a practice of Live Coding that celebrates a "non-competitive, meditative, conversational ethos" rather than virtuosity and danger. However, compulsions towards 'liveness' in Live Coding often favour speed and ephemerality (for instance, in the context of an Algorave performance). Improvisation without the aid of graphical abstractions requires the performer to be mentally and physically dexterous so not to subject an audience to stagnant aesthetic results. Code is generated and subsequently vanishes as quickly as the duration of a half-hour Algorave set, providing little opportunity to experiment with longer compositional forms. Indeed, Slow Coding seems to juxtapose other articulations of Live Coding that emphasize efficiency and speed in such performance situations. For instance,

Mclean and Wiggins (2010a, p.8) suggest one goal of bricolage and interactive programming to "provide a more efficient creative feedback loop". Similarly, the TidalCycles live coding language has aligned its development towards "terse syntax [that] allows for faster expression of ideas, and therefore a tighter programmer feedback loop more suitable for creative tasks" (McLean and Wiggins 2010b, p.2, emphasis added).

Chun's problematization of the conception of immediate program execution suggests that the prospect of 'live' program execution may erase other steps that are necessary for translating code into executable binary (or perceived action more broadly) and thereby serve to reify source code as the essence of new media. A Live Coding performance may be reduced to the performer's code as it is created 'live', negligent of hours of rehearsal, prior software development, preprogramming, community building, and other factors that produce such spectacles.

'Fast' blank-slate Live Coding practices that celebrate the ephemerality of code may also subject the performer to a form of precarity familiar to the contemporary 'gig economy'. In blank-slate performances, the performer often discards all written code at the end. Like the gigger's employment, code is impermanent, undependable, and prone to termination. Parkinson and Bell (2015) illustrate a connection between the 'liveness' of a performance and the audience's perception of the performer's labor. They note that performances are seen as more 'live' when it appears that the performer exerts significant labor. Thus to classify as 'live' and improvisational, Live Coder's may be asked to exert greater labor while also accepting that the results of their efforts (or at least their code) will soon vanish. Parallels can be observed with how 'gigs' demand significant labor while failing to provide persistent economic security. In both situations, compulsions towards speed (in code or employment turnover) enable precarious labor conditions. An appreciation for 'slow' qualities, graphical abstractions, and pre-written code has informed the design of the Dead CJing software presented here, reflecting an attempt to balance liveness and emphemerality with persistence and security.

Code and Liveness in Code Jockeying

Code Jockeying (CJing) (Collins and McLean 2014; Purvis et. al 2019) is an approach to Live Coding in which the performer is primarily concerned with transitioning between pre-written code snippets, rather than generating new ones live. Liveness and improvisation in CJing practice focus on discovering new combinations of code snippets and orchestrating their timely execution. CJing can be seen as a different flavor of Live Coding than the blank-slate approach discussed, with its own different expressions of 'code' and 'liveness'.

Purvis et. al (2019) identified usability limitations of text-based code interfaces for CJing, proposing a suite of gesture-based abstractions (sliders, knobs, MIDI controller inputs) to supplement text-based code in their Visor Live Coding environment. Salazar's (2017) tablet-based Auraglyph software has similarly been motivated by a desire to find gesture in Live Coding, beyond text-based interfaces.

Such motivations to replace or augment text-based interface with continuous controls and gesture-based interactions suggest a desire to transcend a particular manifestation of code fetishism that only recognizes text-based code. When source code is understood not to be 'executable' (as Chun advocates), abstractions such as sliders and MIDI controllers no longer appear to be impurities of otherwise pure and transparent source code; source code is impure from the start and gestural widgets (impurely) continue without claiming higher-order transparency. The emerging practices and software surrounding CJing thus continue to question what counts as 'code', at what point CJing software becomes the same DJ/VJ software that earlier Live Coding literature began to deviate from in the first place, and whether or not it is even desirable to erect such distinctions between code and not-code. The Dead CJing software continues this discussion.

Purvis et. al (2019, p.2) also note the "the difficulty of reusing and sharing content in audiovisual tools" (emphasis added) as a motivation informing the development of the Visor CJing environ-

ment. By deliberately encouraging the preparation and reuse of code, CJing seems to deviate from the form of improvisation, liveness, and ephemerality discussed earlier. Code written for CJing must have persistence beyond a single performance (unlike blank-slate live coding) to enable the performer to explore new forms of improvisation. However, reliance on previously-written code could enable performance practices similar to the 'we all hit play' philosophy of deadmau5, in which the performer merely hits play on a pre-recorded set. Parkinson and Bell (2015, p.1) articulate how deadmau5's "notion of performance as spectacle" deviates from Live Coding's orientation towards free improvisation.

Provided these perspectives, CJing appears to either challenge or extend earlier mandates of Live Coding in their conceptions of both liveness and code. The following section introduces the Dead CJing software and its motivations to either redefine or deviate from liveness and code.

DeadCode

Dead is a browser-based, language-agnostic, and tablet-friendly environment for audio-visual live coding and Code Jockeying (see Figure 1). Dead provides an interface for authoring code snippets that can be toggled on and off on a grid of buttons (called 'stems'). The interface contains sliders and other control inputs to facilitate smooth transitioning between stems and 'tracks' (columns of stems). Currently Dead supports the TidalCycles (McLean and Wiggins 2010b) and Hydra (Jack, n.d.) live coding languages, with the capability to include renderers for other languages in future versions.

Motivations

Dead was conceived as the improbable combination of the TidalCycles (McLean and Wiggins 2010b) live coding language, and the Ableton Live DAW. Dead draws inspiration from Ableton's 'scene launcher' interface which enables the user to create a performance workflow while composing in the DAW, while underlying sound and visuals are generated by TidalCycles and Hydra respectively. Earlier it was discussed how Live Coding has been articulated as a practice separate from conventional Ableton Live performances and similar forms of computer-mediated audio-visual expression. However, perhaps one of Ableton Live's most attractive features is its balance of static composition and live performance. Artists using Live can pre-craft well-produced sound layers (a less 'live' practice) and then use Live's scene launcher interface to perform with them live. To support CJing, Dead aims to strike a similar balance by supporting both static composition and improvised exploration and editing of pre-composed stems.

In response to the earlier discussion on liveness and the ephemeral precariousness of blank-slate live coding, a core design goal of Dead was to facilitate the persistence and reuse of pre-written code. For the author, it was frustrating not to have an organized method for reusing or remixing code snippets produced from successful improvisation sessions. Dead seeks to address this issue by providing an interface that can support both persistence and improvisational exploration. For instance, a common workflow that has been established by using Dead includes starting with an improvisation using one of Stem code editors, producing a desired aesthetic result, and then separating the code into multiple stems (e.g. a different stem for bass, drums, and visuals). After several iterations, the Dead Launch Space will contain a library of Stems that facilitate improvised exploration of new permutations and modifications of layers. The interface places no restrictions on how many stems can be added, permitting Dead compositions to grow arbitrarily large. To support this persistence, it was important early in development to implement basic operations for saving and loading state, as well as moving and managing stems (e.g. with copy and paste commands). The state of a Dead composition can be saved to the browser storage with the keyboard shortcut 'CTRL+s' and loaded from browser storage with 'CTRL+l', or saved to a JSON file with 'CTRL+d' and opened with 'CTRL+o'.

A second goal of the interface was to provide gestural abstractions from the code to allow easier continuous control of parameters and in-

Figure 1: DeadCode interface

stantaneous toggling of code execution. In this regard, Dead has been motivated by several previous Live Coding environments that have supported gestural abstractions, including Visor (Purvis et. al 2019), Auraglyph (Salazar 2017), and Estuary (Ogborn et. al 2017). Continuous controls can be challenging to type in text-based live coding environments, usually requiring the performer to repeatedly delete, modify, and re-run small changes to parameters. Such discretized motions can often only be performed one-at-a-time by a single performer. To transition between records, Disc Jockeys typically need to modulate several parameters at once (for instance filtering out bass frequencies from the out-going track while increasing the bass of an incoming track). To enable the performer to execute multiple actions at once, Dead has been designed for tablets. The performer can simultaneously apply a filter, turn off a code stem, and introduce a new stem with a multi-touch gesture.

Dead's primary design goals towards persistence and graphical interfaces could be seen as deviations from both 'liveness' and 'code'. A discussion of whether Dead violates any necessary conditions of Live Coding follows after a complete overview of the software.

Design and Features

The majority of the Dead interface is composed of the 'Launch Space'; columns of 'stems' represented by buttons that can be started and stopped by clicking or tapping a stem button. When a stem button is clicked, the code underlying that stem is executed, and when a stem is turned off, its code is stopped. Vertical sliders are fixed to the bottom of each stem column for controlling the volume of all stems in that column.

Right-clicking (or long-pressing on tablets) on a stem button opens it in the 'Stem Editor' panel on the right of the interface (see Figure 2). In the Stem Editor, the user can write code that will be executed when that stem is turned on. If the stem is already toggled on, code that is evaluated in the Stem editor (by clicking 'Eval', or toggling the code editor to be 'Live') evaluates the code immediately, enabling the user to improvise during a CJing performance. The Stem Editor also contains horizontal sliders for other effects suited to a continuous range, and code toggles for quickly applying and removing effects to the stem. A 'Master' menu also persists in the Stem Editor, for setting the tempo, applying global effects, and defining code macros that are executed before rendering stems.

Figure 2: DeadCode Stem Editor panel (bordered in red)

The final component in the bottom right corner of the Dead interface includes the 'Render View' (see Figure 3). The Render View contains a projection of the active code (ie. the combination of all stems that are turned on) and any rendering visuals. A copy of the

Figure 3: DeadCode Render View visible in a popped-out window (right) and below the Stem Editor

Render View can be popped out into another full-screen window to display to an audience.

Rendering and Language Neutrality

Dead has been designed to be language-agnostic, employing a similar architecture to the Extramuros (Ogborn et. al 2015) collaborative live coding environment. While the current version supports only TidalCycles and Hydra, the render engine can be easily updated to support more languages. For rendering TidalCycles, each stem is appended to a 'stack' expression, with any applied effects prepended. For instance, the following pseudo-code may be generated by a Dead composition that has two active stems, and a global 'gain' effect applied:

```
d1 $ (|* gain 0.9) $ stack [
  (|* lpf ‘‘900’’) $ s ‘‘bd cp’’, --stem 1 with a low
       pass filter effect
  s ‘‘~ hh ~ hh’’                 -- stem 2
]
```

The TidalCycles code generated by Dead is passed via WebSockets to a client NodeJS application which pipes it to a Haskell interpreter, employing a method introduced by Extramuros (Ogborn et. al 2015). This approach is extensible to Live Coding languages that cannot be directly evaluated in the browser.

Hydra stems are similarly combined into a single Hydra expression, by 'adding' the output of different stems. For instance, two stems containing different Hydra oscillator generators (bolded) would be combined as follows:

```
osc(1,0.5,0.95).add(osc(4,0.5,0.3).rotate(0.6)).out()
```

Future iterations of the software will add different options for how Hydra stems are combined (eg. Hydra's 'blend', 'diff', 'modulate', or 'mult' operators could be used in place of 'add').

Modes of Liveness

Tanimoto (1990) has described the various levels of liveness of programming languages, ranging from relatively static languages that require compilation before execution, to languages that execute as soon as code is written. Live coding languages are often interpreted to support more immediate execution than is possible with compiled languages. Ogborn et. al (2017) provide an overview of some different approaches to liveness employed by Live Coding environments. They note that structure editors such as those offered by Estuary and patcher languages (such as Max/MSP and PureData) can offer immediate execution since they prohibit syntactically invalid expressions that may otherwise occur (eg. in text-based editors) while transitioning from one state of the code to the desired expression. The LiveCodeLab Live Coding environment (Della Cassa and John 2014) proposes an alternative method to achieve immediate execution in a text-based editor, in which code is interpreted immediately, but syntactically invalid expressions are not executed.

Dead offers four distinct levels of liveness. All changes made to sliders in Dead are immediately executed (they are L4 live in Tanimoto's (1990) terminology). Code editors in Dead offer two levels of liveness (see Figure 4). When the 'live' toggle is switched off, the user must either click 'Eval' or hit 'SHIFT+enter' on their keyboard for code to be evaluated. When the 'live' toggle is on, code is evaluated after a configurable amount of time (one second by default) has elapsed since the last keypress. This 'debounced liveness' (termed after the common JavaScript 'debounce' utility for delaying events) provides an alternative method for achieving near-live execution of text-based live coding languages while avoiding show-stopping crashes that could occur if the code was executed on each keypress. The performer is permitted to 'finish their thought' before execution occurs, and can choose to delay execution by 'stammering' (continuously typing and deleting a character).

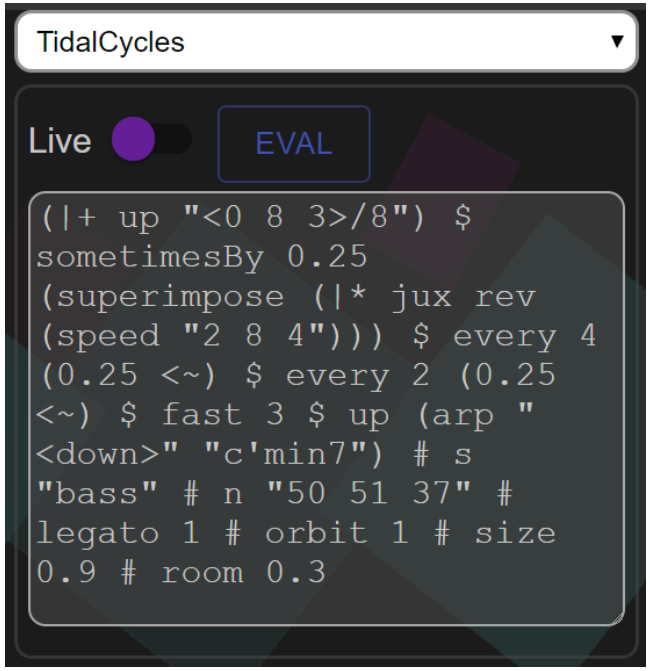


Figure 4: DeadCode code editor with language selection and liveness toggle

From one perspective, Dead's support for the persistence and reuse of code can be interpreted as a fourth level of liveness. Code that is pre-written rather than being authored live (termed here as 'deadcode') has endured a longer process of translation from source code to executable before it is ultimately rendered in front of an audience. The performer's preparatory labor in composing the code, arranging it into navigable labeled 'stems', and other production tweaks can be considered a process of compilation, while execution occurs when that code is invoked in performance. This interpretation of the liveness of deadcode acknowledges the labor involved in such tasks of compilation as a requisite component of CJing performances while maintaining a distinction between such practices and more live blank-slate approaches. Furthermore, framing deadcoding (the act of creating pre-written code for later performance) as compilation, and performance as execution appropriately recognizes the 'live' aspect of CJing as the timely rendering of code in arrangement with other interacting elements.

Multiple Projections for 'Showing Us Your Screen'

In Live Coding performances, the aesthetic of the code is often enough to provide the audience with both 'access to the performer's mind' and something interesting to view, while the interface provides a more functional role. In earlier versions of Dead, it was unclear how the performer would 'show their screen' when their screen was primarily occupied by a functional interface that conceals most of the underlying code. To address this issue, the 'Render View' was added to the bottom right corner of the Dead interface. The Render View contains the code of all active stems overlain on top of the currently rendered Hydra visuals. For performances, new instances of the Render View can be cloned to new windows to present multiple full-screen views to the audience, while the performer can still view the visual output of their performance in the bottom right corner of the performance interface. This approach was motivated by Estuary's (Ogborn et. al 2017) similar capabilities to provide multiple different projections of the same code to suit different viewers and users. The audience is provided a projection that aligns with the aesthetics of a live coding performance, while the performer can control audio-visual results with a more functional projection. Dead's Render View was also informed by VJing software which conventionally allocates a portion of the interface for viewing the video material displayed to the audience.

Architecture and Implementation

Dead was written in JavaScript using the ReactJS/Redux frontend libraries. Live Coding languages used in Dead that cannot be rendered directly in the browser (currently only TidalCycles) are sent over a WebSocket connection to a NodeJS server run locally on the performer's machine and then are piped to the appropriate interpreter. The Dead performance interface communicates with all instances of the Render View via browser post messages. Dead uses the browser's local storage to save state, or users can download a JSON file of their Dead composition and load it back into the browser. Figure 5 provides an overview of the Dead architecture.

Is it Live? Is it Code?

The primary design goals of Dead were to deviate from both liveness and code in preference of a more gesture-based and persistent interface suited to CJing. Therefore, it seems reasonable to question whether the resulting design still qualifies as Live Coding.

The pre-programming encouraged by Dead seems to contrast with Live Coding's celebration of improvisation and uncertainty. In this way, Dead may deviate from McLean and Wiggins' (2010a) formulation of Live Coding as a 'bricoloage' feedback loop between the performer and their machine. Deadcode is generally predictable to the performer. However, the CJing practice supported by Dead facilitates another feedback loop between the performer and the audience that requires another form of improvisation. Similar to DJs and VJs, CJs are tasked with reading their audience to appropriately curate new material to play. Furthermore, unlike most DJ software, the Dead performance interface affords the live alteration of the underlying sound-generation algorithms, and live composition of new stems. Thus while Dead perhaps sacrifices liveness by encouraging pre-programming, it also offers new expressions of liveness consistent with DJ improvisation.

The TOPLAP manifesto's call to 'show us your screens' is also challenged or stretched by Dead. Dead's Render View offers a projection of the code behind currently executed stems to achieve an aesthetic and practice aligned with the common Live Coding tradition of projecting code for the audience. However, as has been discussed, part of the reason Live Coding artists project their code is to provide transparency with regards to what the performer is doing on their computer (ie. not 'just hitting play' and checking emails). A nefarious Render View could easily provide the illusion of liveness, improvisation, and significant labor, while the performer merely hits play on a pre-recorded set. In many respects, such a misleading projection is not so different from the form of 'we all hit play' liveness practiced by deadmau5. As Parkinson and Bell (2015) note, in such performance situations, liveness resides in the spectacle of the performance (the 'Render View'), rather than improvisation. While Dead does not support such views by default, it is noted that projections could enable the form of obscurantism that Live Coding has sought to avoid.

Dead can be used to perform entirely with buttons and sliders, requiring no manipulation of the code underlying each stem. It can be questioned then, whether Dead qualifies as 'coding' to contribute to a continuing discussion in the Live Coding community as to what qualifies as code, and whether or not drawing such distinctions is desired. Beyond just text-based programming languages, the Live Coding community has recognized visual (Max/MSP, PureData), tactile (Cocker 2015), embodied (Baalman 2015), and esoteric representations of code (to name only a few). Defining an exclusive set of expressions of algorithms to qualify as code, therefore, seems capricious, and any definition of 'Live Coding' based on such criteria would be fragile. The earlier discussion of misleading projections of code (for instance in a nefarious Render View) demonstrates how any algorithmic process can trivially be translated into text instructions that satisfy a normative representation of 'code'. Therefore, this paper affirms in concert with Chun (2008), that the essence of Live Coding should not be located in a transfixed representation of code. Stated differently, this paper argues that the 'code' in Live Coding should be

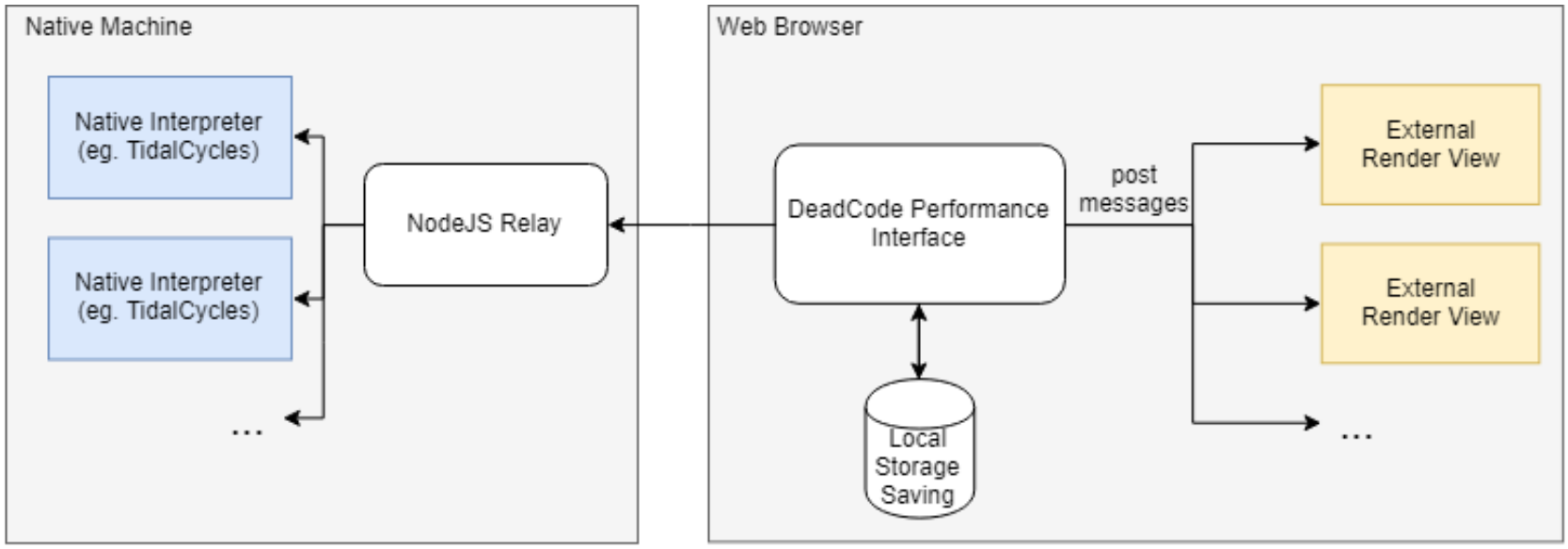


Figure 5: System overview

permitted to breathe into new representations. Whether or not the specific representation presented by Dead qualifies under this broader interpretation is open to scrutiny.

Future Work and TODOs

While this paper has focused primarily on the 'liveness' and 'code' aspects in investigating how CJing and the Dead environment fit within Live Coding, clearly there are other aspects of Live Coding that could have been examined. This paper could have also explored how open source software, networked performance, language neutrality, collaboration, community organization, experimentation, Algorave, and other characteristics of Live Coding are (or are not) represented in Dead's design and use. Such factors could be the subject of future research and suggest several agenda items for the continuing development of the Dead software.

Future development efforts will focus on providing renderers for more Live Coding languages, including but not limited to SuperCollider (McCartney 2002), Gibber (Roberts and Kuchera-Morin 2012), The Force (Lawson n.d.), and Punctual (Ogborn n.d.). Another priority development item will be to support networked collaboration using Dead. One approach could employ a NodeJS WebSocket server application to share state to connected clients (similar to the Extramuros' (Ogborn et. al 2015) server architecture). A secondary objective in supporting networked collaboration may include the creation of a more substantial back end system for sharing Dead stems

and compositions (for instance as is supported by Gibber (Roberts and Kuchera-Morin 2012)). Since Dead will sometimes execute native code on the user's machine (eg. as is done with TidalCycles code that is piped to a GHC interpreter) a stem publishing system would require significant security oversight. Lastly, future work will employ the emerging Web MIDI API to offer support for MIDI inputs for controlling Dead. Since Dead's layout resembles Ableton Live, MIDI mappings will be created for Ableton hardware controllers.

Links

- Demonstration video: `https://youtu.be/nTBwdGbfgmU`
- Source Code Repository: `https://github.com/JamieBeverley/DeadCode`

Acknowledgments

I would like to thank the anonymous reviewers who provided very valuable suggestions for strengthening this work.

References

Baalman, M. (2015) 'Embodiment of code', In Proceedings of the First International Conference on Live Coding (pp. 35–40). Leeds, UK: ICSRiM, University of Leeds. `http://doi.org/10.5281/zenodo.18748`.

Benjamin, W., (1983) Charles Baudelaire: A Lyric Poet in the Era of High Capitalism. 1969. Trans. Harry Zohn. London: Verso, 199.

Blackwell, A.F. and Collins, N. (2005) 'The Programming Language as a Musical Instrument', In Proceedings of the Psychology of Programming Interest Group 17th Annual Workshop (p. 11).

Cocker, E. (2015) 'Live Coding / Weaving -- Penelopean Mêtis and the Weaver-Coder's Kairos', In Proceedings of the First International Conference on Live Coding (pp. 110–116). Leeds, UK: ICSRiM, University of Leeds. `http://doi.org/10.5281/zenodo.19342`.

Collins, N. and McLean, A. (2014) 'Algorave: Live performance of algorithmic electronic dance music', In Proceedings of the International Conference on New Interfaces for Musical Expression (pp. 355-358).

Collins, N., McLean, A., Rohrhuber, J. and Ward, A. (2003) 'Live coding in laptop performance', Organised Sound, 8(3), pp.321-330.

Chun, W.H.K. (2008) 'On sourcery, or code as fetish'. Configurations, 16(3), pp.299-324.

Della Casa, D. and John, G. (2014) 'LiveCodeLab 2.0 and Its Language Livecodelang', In Proceedings of the 2nd Acm Sigplan International Workshop on Functional Art, Music, Modeling & Design (Farm '14'), 1–8. ACM, doi:10.1145/2633638.2633650.

Gregg, M., (2018) Counterproductive: Time management in the knowledge economy. Duke University Press.

Grosse-Hering, B., Mason, J., Aliakseyeu, D., Bakker, C. and Desmet, P. (2013) "Slow design for meaningful interactions", In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 3431-3440). ACM.

Hall, T. (2007) 'Towards a Slow Code Manifesto', Published online; `http://www.ludions.com/slowcode/`.

Hallnäs, L. (2015), 'On the Philosophy of Slow Technology', Acta Universitatis Sapientiae-Social Analysis, 5(1).

Jack, O. (n.d.), 'Hydra', computer software, available online; `https://github.com/ojack/hydra`.

Lawson, S. (n.d.), The Force, computer software, available online; https://github.com/shawnlawson/The_Force.

Magnusson, T. (2014), 'Improvising with the threnoscope: integrating code, hardware, GUI, network, and graphic scores'. In Proceedings of the Internation Conference on New Interfaces for Musical Expression. Goldsmiths University of London.

McCartney, J. (2002), 'Rethinking the computer music language: SuperCollider', Computer Music Journal, 26(4), pp.61-68.

McLean, A. and Wiggins, G.A. (2010), 'Bricolage Programming in the Creative Arts', In 22nd Psychology of Programming Interest Group (p. 18).

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

Odom, W, (2015), 'Understanding long-term interactions with a slow technology: an investigation of experiences with FutureMe', In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (pp. 575-584). ACM.

Odom, W., Selby, M., Sellen, A., Kirk, D., Banks, R. and Regan, T. (2012), 'Photobox: on the design of a slow technology', In Proceedings of the Designing Interactive Systems Conference (pp. 665-668). ACM.

Ogborn, D., (n.d.), Punctual, computer software, available online; https://github.com/dktr0/Punctual.

Ogborn, D., Beverley, J., del Angel, L.N., Tsabary, E. and McLean, A. (2017), 'Estuary: Browser-based Collaborative Projectional Live Coding of Musical Patterns', In International Conference on Live Coding 2017.

Ogborn, D., Tsabary, E., Jarvis, I., Cárdenas, A. and McLean, A. (2015), 'Extramuros: making music in a browser-based, language-neutral collaborative live coding environment', In Proceedings of the First International Conference on Live Coding, University of Leeds, ICSRiM (p. 300).

Parkinson, A., and Bell, R. (2015), 'Deadmau5, Derek Bailey, and the Laptop Instrument -- Improvisation, Composition, and Liveness in Live Coding', In Proceedings of the First International

Conference on Live Coding (pp. 170–178), Leeds, UK: ICSRiM, University of Leeds. http://doi.org/10.5281/zenodo.19350.

Purvis, J., Anslow, C. and Noble, J. (2019), 'CJing Practice: Combining Live Coding and Vjing', In Proceedings of the International Conference on Live Coding 2019.

Roberts, C. and Kuchera-Morin, J. (2012), 'Gibber: Live coding audio in the browser', In Proceedings of the International Computer Music Conference (ICMC 2012).

Roberts, C. and Wakefield, G., 2016. Live Coding the Digital Audio Workstation. In Proceedings of the 2nd International Conference on Live Coding.

Rohrhuber, J., de Campo, A., Wieser, R., Van Kampen, J.K., Ho, E. and Hölzl, H. (2007), 'Purloined letters and distributed persons', In Music in the Global Village Conference.

Rosa, H. (2013), Social acceleration: A new theory of modernity. Columbia University Press.

Salazar, S. (2017), 'Searching for gesture and embodiment in live coding', In Proceedings of the International Conference on Live Coding.

Strauss, C. F., and Fuad-Luke, A. (2009), 'The slow design principles', available online; www.slowlab.net/CtC_SlowDesignPrinciples.pdf.

Tanimoto, S.L. (1990), 'VIVA: A visual language for image processing', Journal of Visual Languages & Computing, 1(2), pp.127-139.

TOPLAP. (2010), 'ManifestoDraft', available online; `https://toplap.org/wiki/ManifestoDraft/`.

Wang, G., Cook, P.R. and Salazar, S. (2015), 'Chuck: A strongly timed computer music language', Computer Music Journal, 39(4), pp.10-29.

Live Coding Procedural Textures of Implicit Surfaces

Charles Roberts
Department of Computer Science
Worcester Polytechnic Institute
charlie@charlie-roberts.com

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

We describe a system for live coding procedural textures on implicit surfaces, and how its implementation led to foundational changes in the end-user API for the live coding environment marching.js. The texturing additions to marching.js enable users to use predefined texture presets, to live code their own procedural textures, or to use other systems for fragment shader authoring, such as Hydra, to generate textures. It also affords using the browser's 2D canvas API to define textures, providing an entry point for performers who might be familiar with web APIs but unfamiliar with lower-level GPU languages like GLSL. We describe how demoscene culture led us to initially adopt design decisions that were inappropriate for our particular system, and the changes to both our underlying engine and end-user interface that resulted from reconsidering these decisions in the context of procedural texturing.

Introduction

We previously developed a library, marching.js, that exposes a ray marching engine for live coding performance. This system enables programmers to describe 3D scenes in JavaScript, which are then compiled into fullscreen GLSL fragment shaders. In our initial writings on the library (Roberts 2019) we described how the scenes generated by the system often felt "...'technical', 'clinical', or perhaps even 'cold'." While post-processing filters were mentioned as one possible solution for this problem, the research presented here instead investigates a variety of techniques to enable procedural texturing of the implicit surfaces (Hart 1993) rendered by marching.js.

In somewhat of a surprise, the implementation of these features led to fundamental API changes in our system and rethinking culturally derived assumptions about how the rendering engine for our system should function. We will describe some of the background that led to these assumptions, and how the implementation of procedural texturing for implicit surfaces led to both a terser end-user programming

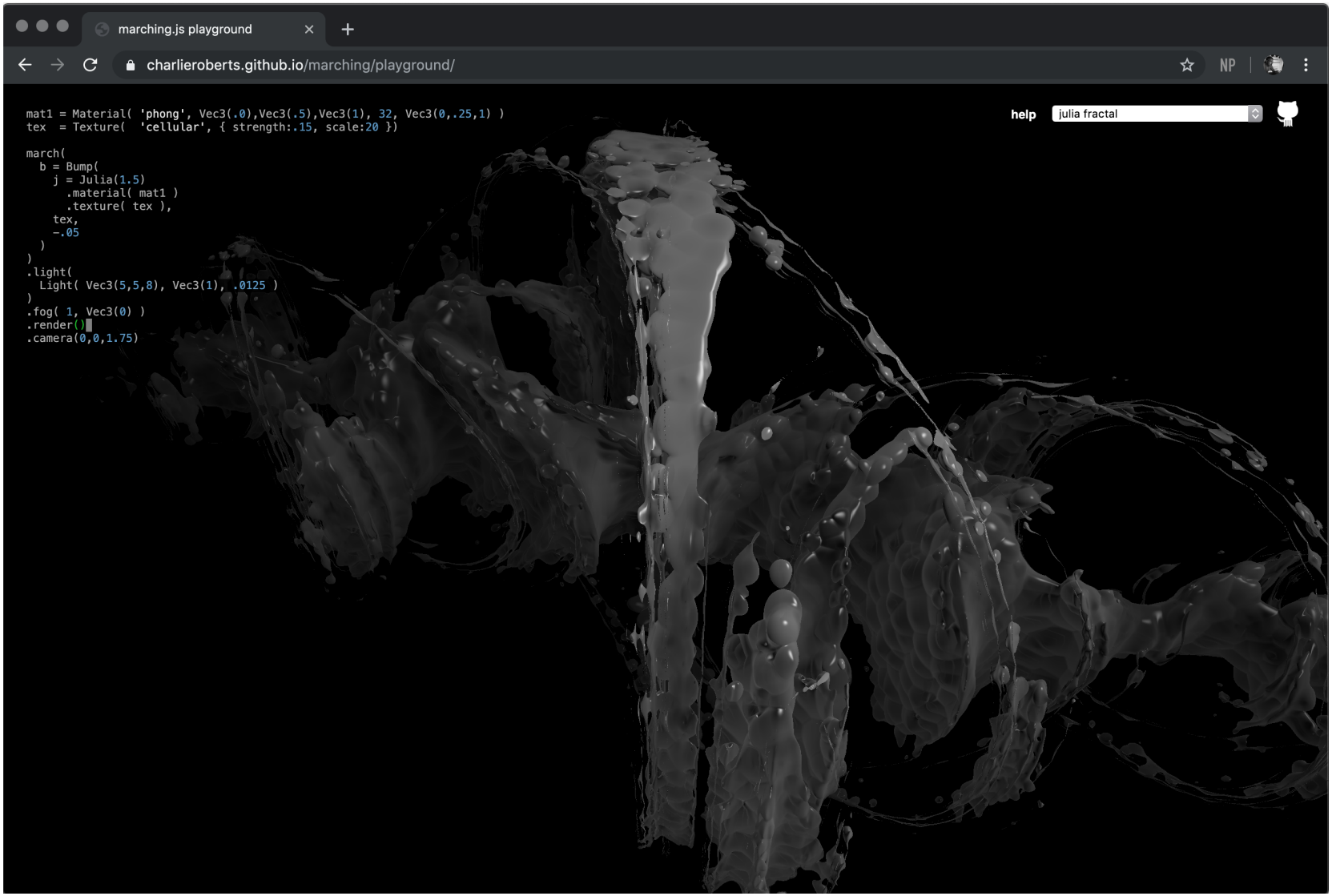


Figure 1: The quaternion Julia set, textured and bump-mapped with cellular noise, as rendered by marching.js

interface and low-level engine optimizations. We then outline various levels of interface for our texturing system, and our attempts to ensure the idealized "low threshold, high ceiling, wide walls" (Resnick et al. 2005) design space that help characterize successful creativity support tools. We conclude with technical and aesthetic directions for future research.

Background

In this section we begin by briefly describing ray marching as a rendering technique. We then discuss how the demoscene (Carlson 2009) affected many of our design decisions when initially authoring marching.js, and contextualize the use of procedural texturing in marching.js within the broader community of live coders.

Ray Marching

Ray marching is a method of rendering (primarily) three-dimensional scenes. It is perhaps best understood in contrast to a much more common 3D graphics pipeline, which incorporates tessellation and rasterization. In this process, geometries are subdivided into triangles (tessellation). The vertices for these triangles are then sent to the GPU, where the triangles are reassembled and projected from their 3D location to the 2D viewing plane (rasterization).

In contrast, ray marching is a physically-informed rendering technique that enables programmers to use mathematical formulae to define and combine geometries, without having to worry about tessellation or rasterization. In ray marching, a ray is projected from a virtual camera through each pixel in the output and into a three-dimensional scene; if this ray strikes an object in the 3D scene the pixel that the ray travels through is assigned the color of that particular object. This rendering technique makes a variety of operations that are complex to perform with tessellated triangles much simpler, such as fluidly morphing from one shape to another, or infinitely repeating a geometry throughout a space. However, tessellation and rasterization are extremely efficient, while ray marching typically requires a fairly powerful graphics card to perform realtime rendering on high resolution displays. We describe ray marching in greater detail in our prior writings about marching.js (Roberts 2019).

Cultural Assumptions in Marching.js

Many popular introductory tutorials on ray marching are presented in the context of the demoscene, a culture that emphasizes the production of audiovisual sketches (termed *demos*) that explore the boundaries of what is possible within technical constraints. These constraints can include the adoption of a particular low-resource technology platform, artificial constraints on the number of bytes a program can occupy in memory, or, in the case of many live demoscene competitions, the challenges of creating complex three-dimensional worlds in realtime in a competitive head-to-head "battle" setting. The demoscene features a variety of online venues for promoting discussion and dissemination of the idiomatic techniques used within it. These include the labyrinthine pouet.net—a popular forum for discussing techniques ranging from shader programming to analog techniques involving overhead projectors and paper cutouts—and shadertoy.com, a site for viewing, editing, and sharing demos realized in the browser using GLSL (Graphics Language Shader Language), one of the most widely used languages for authoring programs that are parallelized to run on the graphics programming unit (GPU) of a computer. The first version of marching.js was heavily influenced by the cultural focus of the demoscene on GLSL demos and the ready availability of related references and tutorials. This led to questionable design decisions that we re-examined in the context of procedural texturing.

Live Coding of Texture

Our research is particularly interested in the application of texture to three-dimensional geometries; however, we note there is also a broader discussion of texture within the live coding community as it relates

to both musical pattern and computational craft (McLean 2013).

Additionally, there is an established practice of live coding fullscreen fragment shaders in the live coding community; these shaders can be thought of as textures for simple rectangles that fill the entire projection. Popular environments for GLSL live coding of this type include The Force (Lawson & Smith 2017), KodeLife (Fischer n.d.), and Veda (Amagi n.d.), while the demoscene community typically uses a standardized system named Bonzomatic (Szelei n.d.) for live competitions. Other visual live coding systems, such as La Habra (Hennigh-Palermo n.d.) and Visor (Purvis, Anslow & Noble 2019) primarily use 2D programming APIs, such as Processing (Reas & Fry 2006) in the case of Visor or a ClojureScript environment for programming Scalable Vector Graphics in the case of La Habra.

The live coding system Hydra (Jack n.d.) adopts a different approach, providing an end-user JavaScript API that wraps a code generation engine for writing GLSL shaders. Hydra is "...a modular and distributed video synthesizer" (ibid.), and similar to most analog video synthesis systems the output can typically be considered two-dimensional.

Our research on using textures within marching.js builds off of many of the ideas found in these other systems, enabling live coders to define textures that are created using the built-in HTML ¡canvas¿ element, and to use Hydra as a texture for the implicit surfaces marching.js provides.

Rethinking the Live Coding Interface for Marching.js

As we implemented procedural texturing for the first time in marching.js, we faced a difficult problem.

The code in Listing 1, creates a box that is rotated on its x-axis and scaled before rendering. When we first tried to map textures to such geometries, the effect became that of a textured blanket layered over the top of the geometry: the box would rotate underneath and scale appropriately, but the "blanket" would just hang in place while barely moving, instead of being wrapped tightly around it so that as the geometry rotated, the texture did as well. More succinctly: our geometry rotated but our texture did not.

```
march(
  Rotate(
    Scale( Box(), .5 )
    Vec3( 1,0,0 ),
    Math.PI / 3
  )
).render()
```

Listing 1: A scaled and rotated box in marching.js

This problem is more complex when we look at aggregate objects that contain transformations applied to individual members of the aggregate as well as the aggregate itself. For example, consider code in Listing 2.

```
march(
  Rotate(
    Union(
      Rotate( Box(), Vec3(0,1,0), Math.PI/3 ),
      Sphere( 1.25 )
    ),
    Vec3(1), Math.PI/5
  )
).render()
```

Listing 2: A rotated box and a sphere combined via a Union combinator which is then also rotated

In Listing 2 we apply a rotation to our box, and then apply a rotation to the union of the rotated box and the sphere. If we textured the resulting aggregate geometry, we would need to take both of these rotations into account at different parts of the texturing process.

The code generation engine in marching.js is effectively divided into two stages. In the first, we determine whether or not rays traveling through a pixel on our screen hit an object in the scene; if so, we need to color that pixel based on the material / texture of the object, and on the lighting of the scene. The second lighting stage calculates this color, however, in marching.js version 1 the lighting stage cannot access the transforms of the geometry that is being lit. We had to significantly refactor the code generation engine in order provide access to these transformations, which in turn led to questioning some of our underlying assumptions about how our engine should function. These changes enable users to freely assign transformations at any level of hierarchy, as shown in Figure 2.

Transform Everything

The first change we made was to assign matrix-encoded transforms for rotation, translation, and scale to every operation in marching.js. Explicitly wrapping individual functions in such transforms was no longer necessary, and a transformation matrix is automatically applied to all geometries.

Listing 3: Comparing old and new syntaxes for rotating a box in marching.js captionpos

```
// old syntax
march( Rotate( Box(), Vec3(1,0,0), Math.PI/2 ) )
  .render()

// new syntax
march( Box().rotate( 45, 1,0,0 ) ).render()
```

One effect of our changes is that the API for applying transformations immediately became much terser; in our opinion its clarity is also improved. Listing 4 shows a more complex example for comparison:

Listing 4: Comparing syntaxes for applying a variety of transformations to a box in marching.js captionpos

```
//old syntax to rotate,translate,and apply material
march(
  Rotate(
    Box( null, Vec3(1,0,0), Material('glue') ),
    Vec3( 1,0,0 ),
    .5
  )
).render()

//new syntax to rotate,translate,and apply material
march(
  Box()
    .rotate( Math.PI / 3, 1,0,0 )
    .translate( 1,0,0 )
    .material( 'glue' )
).render()
```

The new syntax is more explicit about what is occurring, making it easier to read, while only being two characters greater in length in Listing 4. It also helps to avoid deep nesting which can difficult to parse and awkward to augment with additional code. Given that scaling, movement, and rotation are typically important parts of performing with 3D geometries, we feel that improving the application of such transformations is significant.

A tradeoff to these benefits is consistency. In first release of marching.js, nested functions were used to create geometries, transform them, and apply domain operations that could radically alter a scene; in the newest version, transforms, texturing, and application of material are instead achieved by method calls. We experimented with also applying domain operations using method calls, however, we found the resulting syntax to be ambiguous and difficult to apply consistently. In its new form, the programming interface is currently non-homogenous in how various operations are applied, but we still

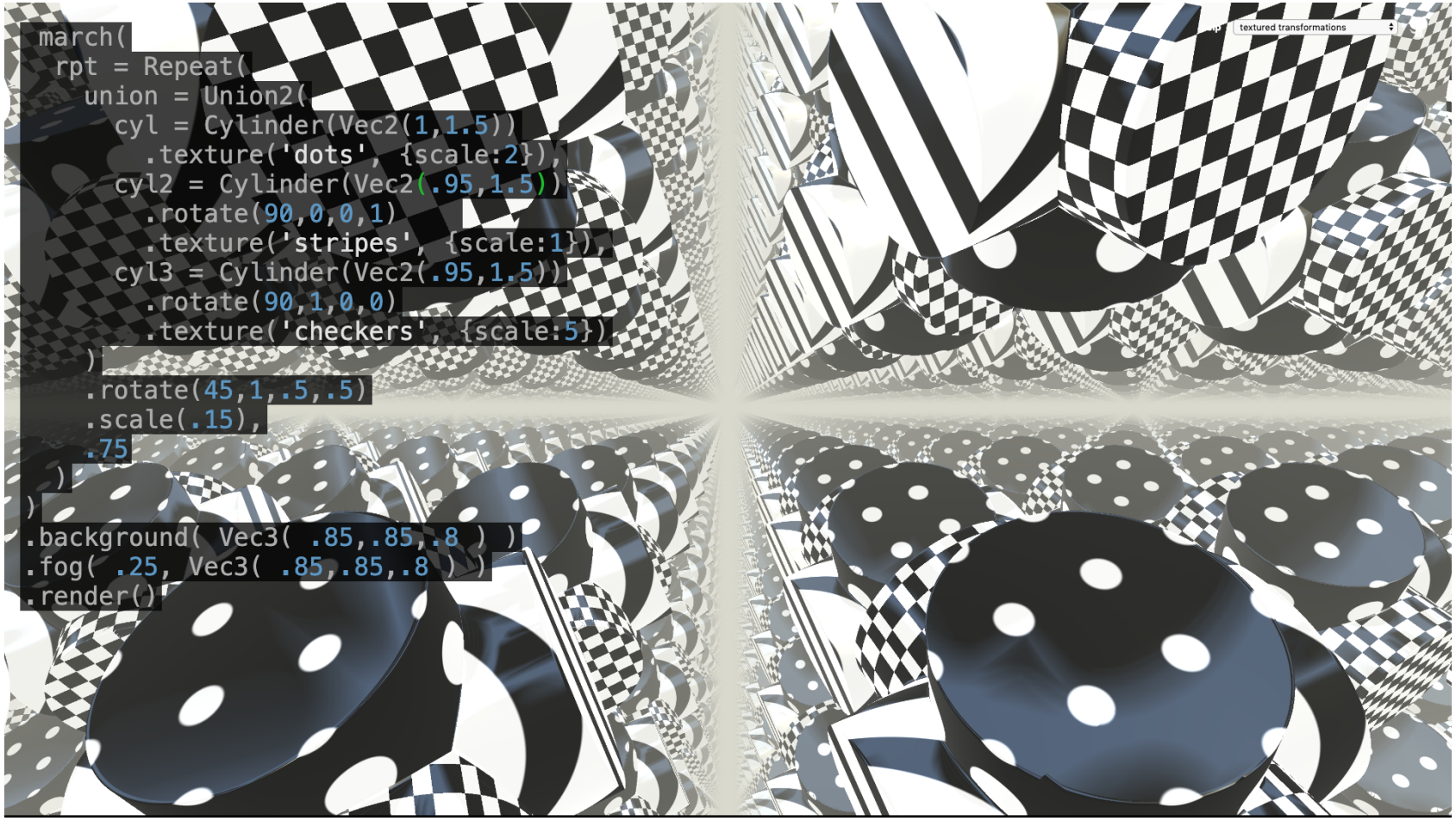


Figure 2: Three cylinders with different rotations, scaled and repeated, with coherent procedural textures.

believe it is clearer than our prior solution.

Improving Efficiency and Code Generation

As discussed previously, the implementation for marching.js was created using online references and code examples. The majority of these references were authored by demoscene participants, who commonly perform all graphics processing on the GPU. This is an aesthetic choice that places all graphics code in a (typically) single file using a single language (GLSL), making it easier for viewers and programmers to understand. However, some operations, such as the transformations described in this section, are in fact more efficient to perform on the CPU.

The reason for this is the parallel nature of GPUs, which makes it difficult to share information across various invocations of the main fragment shader function. Since this main function is invoked once per pixel being rendered, GPU based transformations are thus calculated thousands of times per frame, and then must be repeated on every additional frame. For some transformations, like translation, this is not a significant cost, however, for others such as rotation it is an expense best avoided.

Now that every operation in marching.js has a transformation matrix associated with it, we can calculate this matrix a single time on the CPU, transfer the matrix to the GPU, and then use the same data for rendering every pixel in the operation. The transform doesn't need to be recalculated unless the it is changed in some fashion (for example, increasing rotation), meaning in some cases we only need to calculate the transformation a single time. This is clearly a win over having to recalculate it for every pixel on every frame, regardless of whether any changes to the transformation have occurred. Such optimizations are perhaps obvious in hindsight, but were only achieved by reconsidering the context of the demoscene tutorials, references, and libraries that influenced marching.js.

Changing Code Generation

In the generated shaders, each operation references a matrix that represents the operation's cumulative transformation. This includes transformations applied directly to the operation, transformations applied to any domain operations that wrap the operation, and transformations that might be applied to any higher-level geometry that the operation is a part of. As these various transformations are applied (usually via matrix multiplication, with the code generation engine ensuring correct application order by explicitly writing it into the generated shader), the code generation engine stores each step of the transformation as needed so that it can be referenced during texturing.

Textures

marching.js enables users to approach texturing in a variety of different ways, providing texturing options for beginning programmers as well as more advanced programmers who are fluent in GLSL. In order of increasing complexity, these techniques include:

1. Predefined 2D GLSL textures that can be wrapped around objects
2. Predefined 3D GLSL textures
3. Using a standard image file (.png, .gif, .jpg etc.)
4. Using the 2D <canvas> API provided by the browser
5. Using Hydra and other systems that output to <canvas> elements
6. Writing custom GLSL textures

Predefined Textures

The predefined textures included with marching.js (shown in Fig.2) are accessible via presets that can referenced by name, as shown in Listing 5.

```
march(
  Box().texture( 'dots' )
).render()
```

Listing 5: Using a texture preset in marching.js

Texture objects can also be defined and used in multiple geometries. Additionally when a call to .texture() is used on a geometric combinator (Union, Intersection, Difference etc.) the texture is applied to all surfaces belonging to the combinator; this also applies to domain operations like Repetition. Listing 6 provides code examples of both methods.

```
// define a texture used by multiple objects
tex = Texture( 'truchet' )
march(
  Box().texture( tex ),
  Sphere( 1.35 ).texture( tex )
).render()

// or use a combinator to apply texture
march(
  Difference(
    Box(),
    Sphere( 1.25 )
  ).texture( 'truchet' )
).render()
```

Listing 6: Applying one texture across multiple geometries via reusing a texture and applying a texture to a combinator

Using the HTML <canvas> Element as a Texture

Many beginning web and graphics programmers experiment with the HTML 2D <canvas> API. By offering <canvas> as one of the options for texturing in marching.js, we enable these programmers to easily experiment with texturing without having to learn GLSL. These textures can be animated and updated in the onframe method that marching.js uses for animation.

Integrating with Hydra

Hydra is a popular live coding system that operates on a similar principle to marching.js: users provide a high-level description in JavaScript of a representation which is then compiled to a GLSL shader for display. In Hydra, the operations are typically derived from analog video synthesis techniques, while in marching.js the operations relate to volumetric rendering and constructive solid geometry.

Performers use Hydra to create 2D patterns that change over time, making it a perfect candidate to use as texture generator for marching.js. Fig 4 shows Hydra being used to texture a Mandelbox fractal. The Hydra graph can be edited and redefined at any time to update the applied shader texture. We imagine future collaborative performances where one user could program textures in Hydra while another programmed 3D scenes that used the generated textures.

GLSL Textures

Fragment shaders for texturing can also be authored directly inside of marching.js, via the same API that is used internally to define the various texture presets included in marching.js. This API enables end-users to define points for interacting with their texture as well as the raw GLSL code that is needed to calculate an output color value.

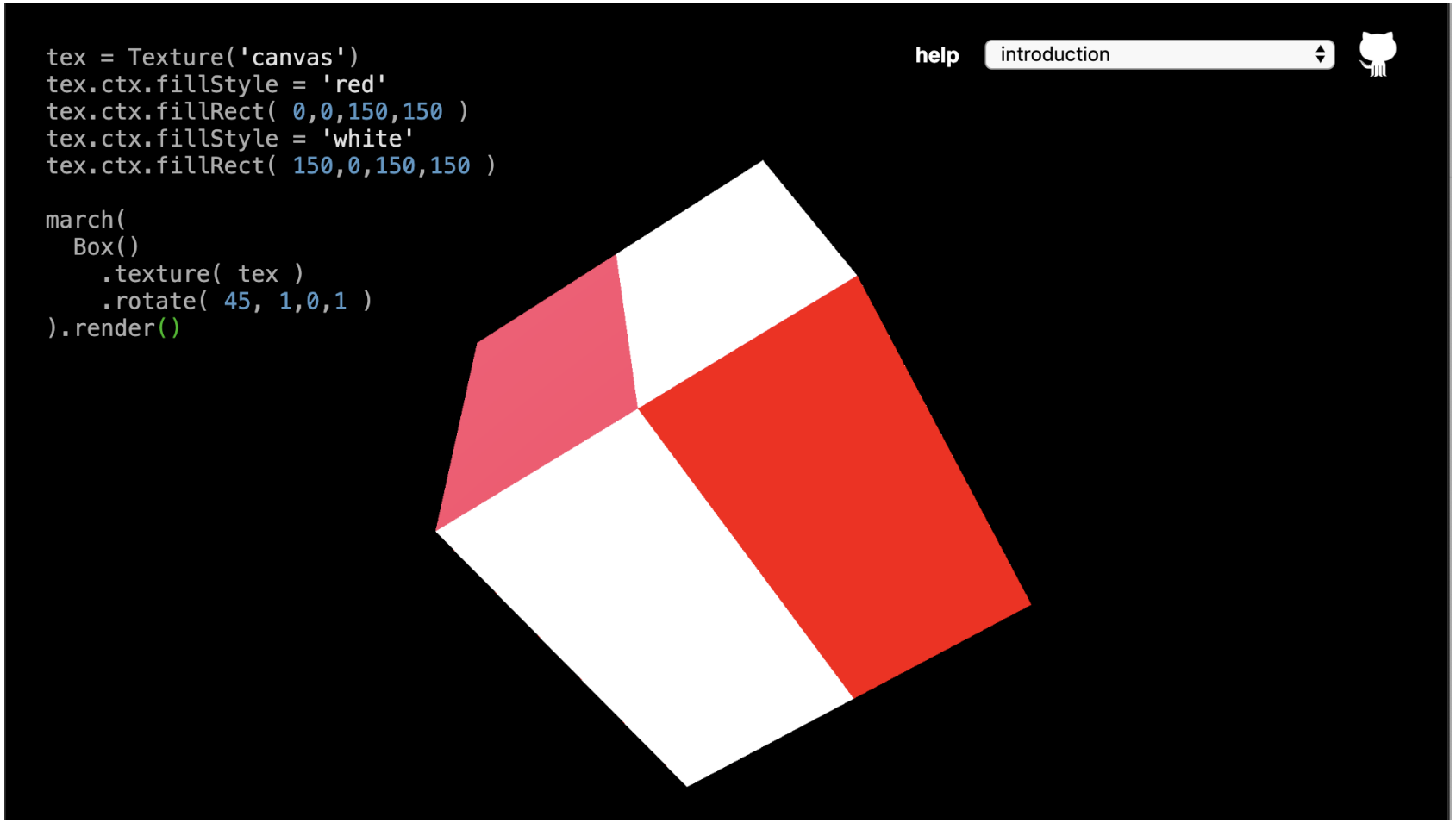


Figure 3: Using a HTML <canvas> element to texture a surface.

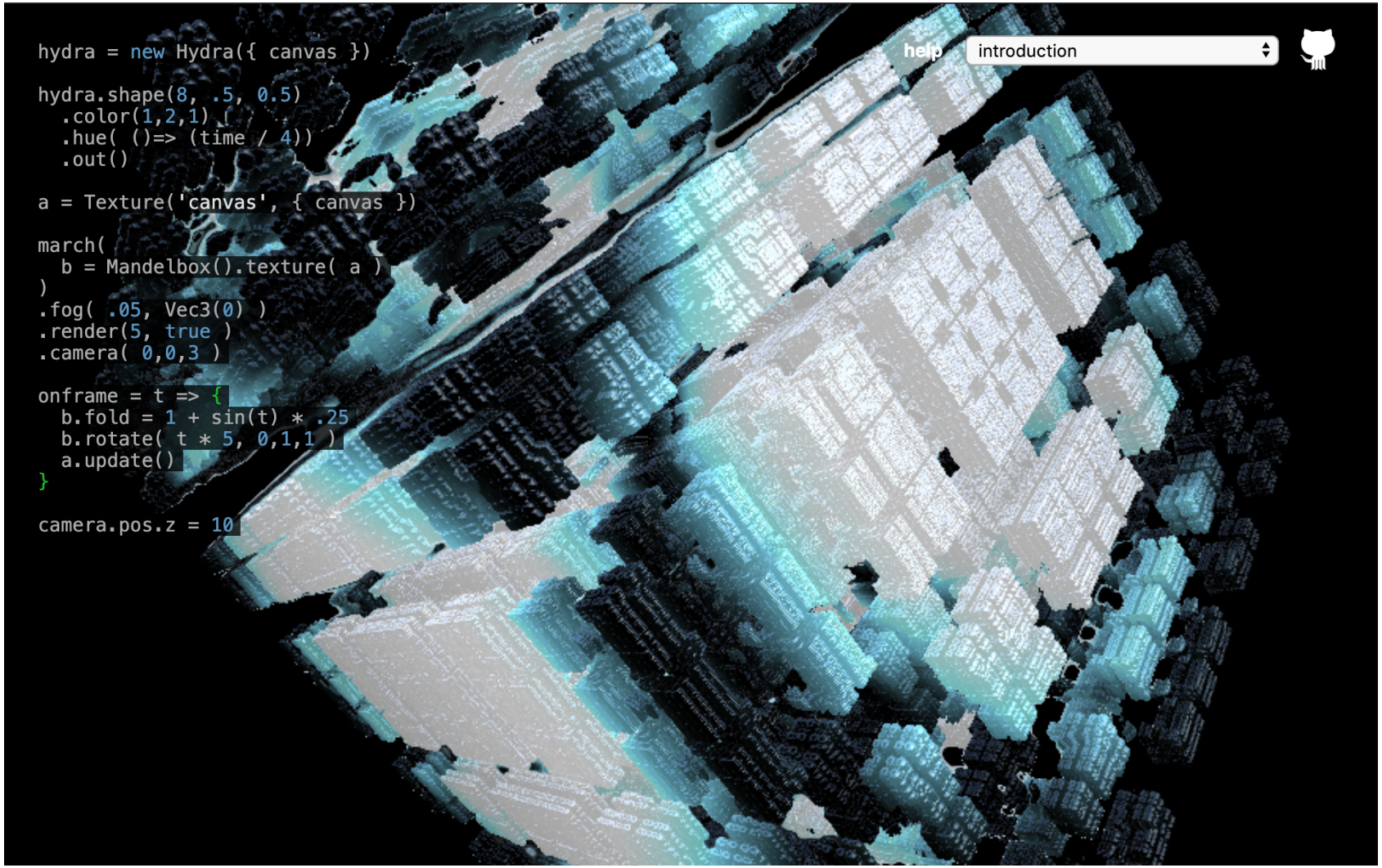


Figure 4: Hydra in use to texture a Mandelbox fractal.

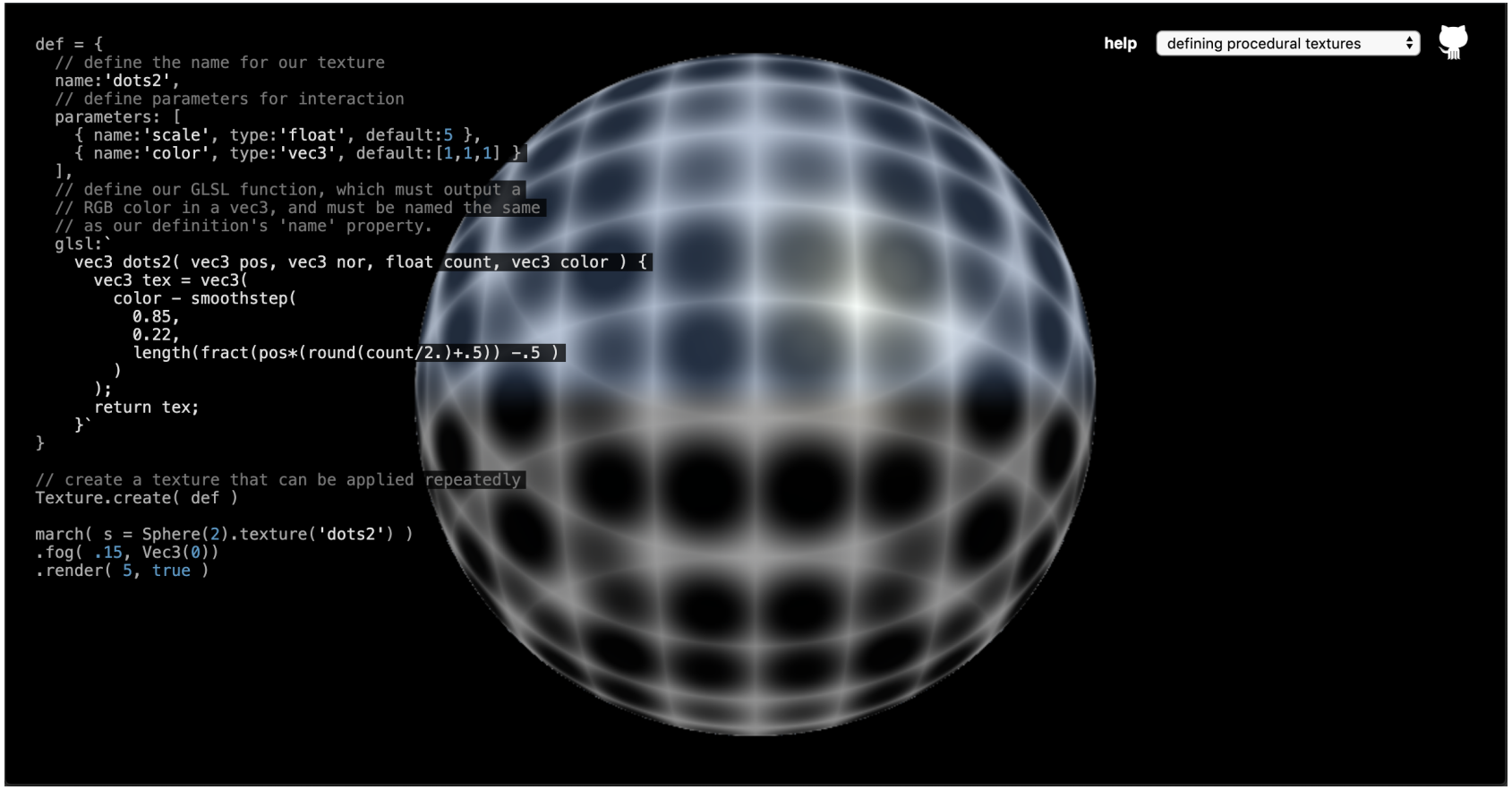


Figure 5: Defining and using a texture written in GLSL.

Manipulating Textures

After applying a texture the properties of the generated texture are added as members of the texture function itself, exposing them for realtime control. In Listing 7, the time property of a 4D simplex noise texture is changed in each frame of generated video.

```
march(
  plane = Plane().texture('noise')
).render()

onframe = function( time ) {
  plane.texture.time = time
}
```

Listing 7: Changing texture properties over time

While most texture properties vary according to the preset used, every texture has a scale property that is used as a scalar to modify the texturing coordinates internally to the shader. Most also have a 'strength' property that determines the effect of the texture in determining the final color of each pixel.

Conclusions and Future Work

We extended a system/library, marching.js, to include a variety of methods for texturing implicit surfaces. This required rethinking fundamental aspects of how its code generation and ray marching engines functioned, but resulted in a terser, more readable end-user API that should lead to more fluid live coding performances. The texturing methods we implemented help to ensure that programmers of varying experience will be able to experiment with texturing, while providing integration with the live coding system Hydra ensures that its users can transfer prior knowledge while experimenting or performing with volumetric rendering techniques. The updates to this system are open source and available online at https://charlieroberts.github.io/marching/playground/.

There are improvements to be made in the sampling algorithms for 2D textures and anti-aliasing more generally in marching.js. Additionally, integration with p5.js, a JavaScript port of Processing (McCarthy, Reas & Fry 2015), could open texturing in marching.js to the many artists and students who actively use that platform. Conversely, we are also considering porting the library to run in p5.js, so that the Processing community will have a relatively easy platform to explore volumetric rendering.

References

Amgai, T. (n.d.) VEDA-VJ app for Atom. [Online] Available at: `https://veda.gl/`. Accessed on Wed, September 25, 2019.

Carlsson, A. (2009), The forgotten pioneers of creative hacking and social networking–introducing the demoscene, Re: Live: Media Art Histories 2009 Conference Proceedings. pp. 16–20.

Hennigh-Palermo, S. (n.d.) La Habra: The Shape of Things to DOM. [Online] Available at: `https://github.com/sarahgp/la-habra` . Accessed on Mon, December 16, 2019.

Hart, J. C. (1993), Ray tracing implicit surfaces. In: Siggraph 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces pp. 1–16.

Fischer, R. (n.d.) KodeLife. [Online] Available at: `https://hexler.net/products/kodelife` . Accessed on Wed, September 25, 2019

Jack, O. (n.d.) hydra. [Online] Available at: `https://hydra-editor.glitch.me`. Accessed on Mon, December 16, 2019.

Lawson, S. & Smith, R. R. (2017), The Dark Side. In: Centro Mexicano para la Musica y las Arts Sonoras (Mexico): Proceedings of the Third International Conference on Live Coding.

McCarthy, L., Reas, C. & Fry, B. (2015) Getting Started with P5.js: Making Interactive Graphics in JavaScript and Processing. Maker Media, Inc.

McLean, A. (2013) The textural x. In: Proceedings of xCoAx2013: Computation Communication Aesthetics and X. pp.81–88.

Purvis, J., Anslow, C. & Noble, J. (2019) CJing Practice: Combining Live Coding and Vjing. In: Proceedings of the 2019 International Conference on Live Coding.

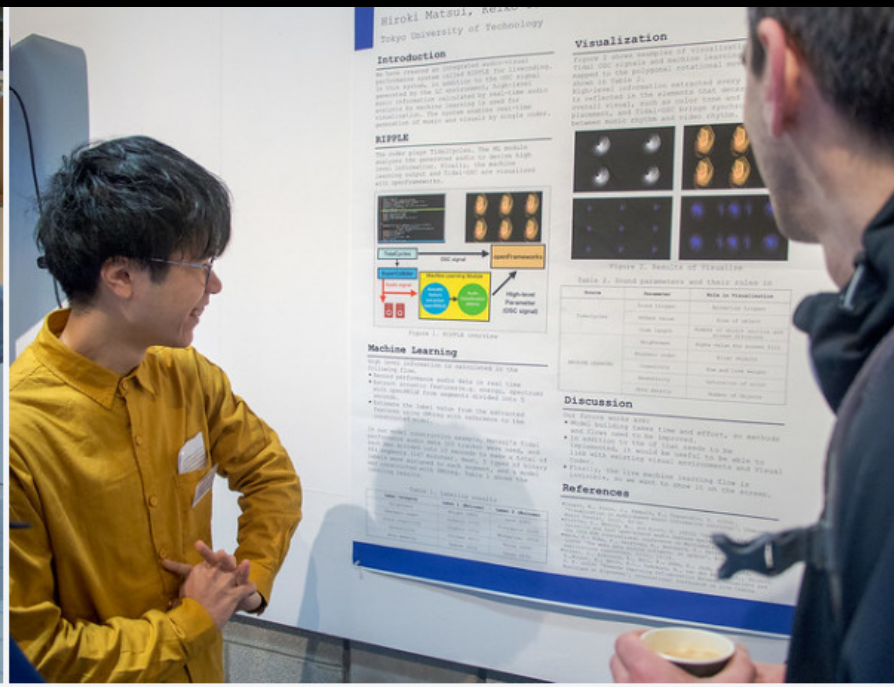
Reas, C. & Fry, B. (2006) Processing: programming for the media arts. AI & SOCIETY, 20(4), 526–538.

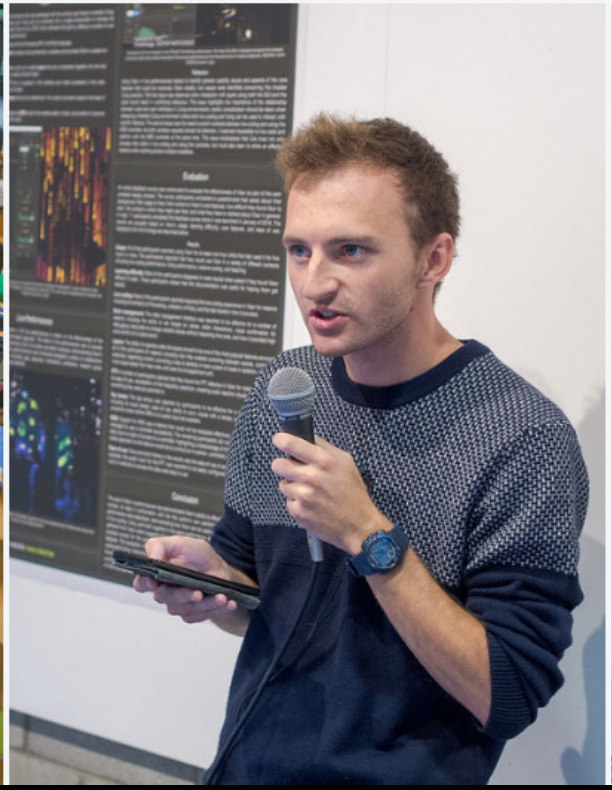
Resnick, M., Myers, B., K., N., Shneiderman, B., Pausch, R., Selker, T. & Eisenberg, M. (2005) Design Principles for Tools to Support Creative Thinking. Technical report.

Roberts, C. (2019) Live Coding Ray Marchers with Marching.js. In: Proceedings of the 2019 International Conference on Live Coding.

Szelei, G. (n.d.) Bonzomatic: Tool for the Live Coding Compo debuted at Revision 2014. [online] Available at: `https://github.com/Gargaj/Bonzomatic`. Accessed on Monday, December 16, 2019.

POSTERS





RIPPLE: integrated audio visualization for livecoding based on code analysis and machine learning

Hiroki Matsui *
g3119019aa@edu.teu.ac.jp

Keiko Ochi*
ochikk@stf.teu.ac.jp

Yasunari Obuchi*
obuchiysnr@stf.teu.ac.jp

*Tokyo University of Technology

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Introduction

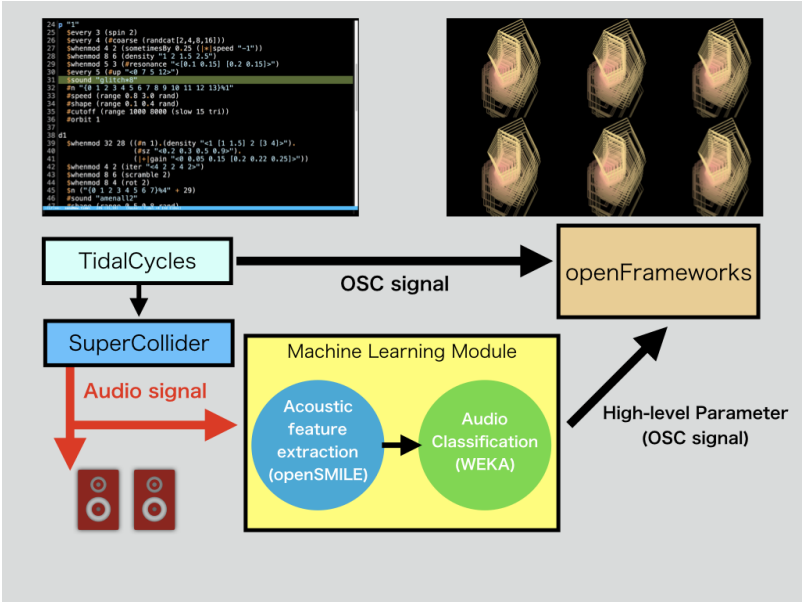


Figure 1: RIPPLE overview

We have created an integrated audio-visual performance system called RIPPLE for livecoding. In this system, in addition to the OSC signal generated by the LC environment, high-level music information calculated by real-time audio analysis by machine learning is used for visualization. The system enables real-time generation of music and visuals by single coder.

RIPPLE

The coder plays TidalCycles. The ML module analyzes the generated audio to derive high level information. Finally, the machine learning

output and Tidal-OSC are visualized with openFrameworks.

Machine Learning

Label Category	Label 1 (#slices)	Label 2 (# slices)
Brightness	Bright (643)	Dark (240)
Rhythmic Order	Orderly (771)	Disorderly (112)
Sound Organicity	Organic (372)	Mechanical (511)
Harmonicity	Pitched (617)	Noisy (266)
Note Density	Sparse (413)	Dense (470)

Table 13.1: Labeling results

High level information is calculated in the following flow.

- Record performance audio data in real time
- Extract acoustic features(e.g. energy, spectrum) with openSMILE from segments divided into 5 seconds.
- Estimate the label value from the extracted features using SMOreg with reference to the constructed model.

In our model construction example, Matsui's Tidal performance audio data (20 tracks) were used, and each was divided into 10 seconds to make a total of 884 segments (147 minutes). Next, 5 types of binary labels were assigned to each segment, and a model was constructed with SMOreg. Table 13.1 shows the labeling results.

Visualization

Figure 2 shows examples of visualization. Tidal OSC signals and machine learning output are mapped to the polygonal rotational movement as shown in Table 13.2. High-level information extracted every few seconds is reflected in the elements that determine the overall visual, such as color tone and object placement, and Tidal-OSC brings synchronization between music rhythm and video rhythm.

Discussion

Our future works are:

- Model building takes time and effort, so methods and flows need to be improved.
- In addition to the oF that needs to be implemented, it would be useful to be able to link with existing visual environments and Visual Coder.
- The live machine learning flow is invisible, so we want to show it on the screen.

Source	Parameter	Role in Visualization
TidalCycles	Sound Trigger	Animation Trigger
	Effect Value	Size of Object
	Code length	Number of object vertics and screen divisions
	Brightness	Alpha value for screen fill
Machine Learning	Rhythmic order Organicity	Align objects Hue and line weight
	Harmonicity	Saturation of color
	Note density	Number of Objects

Table 13.2: Sound parameters and their roles in visualization

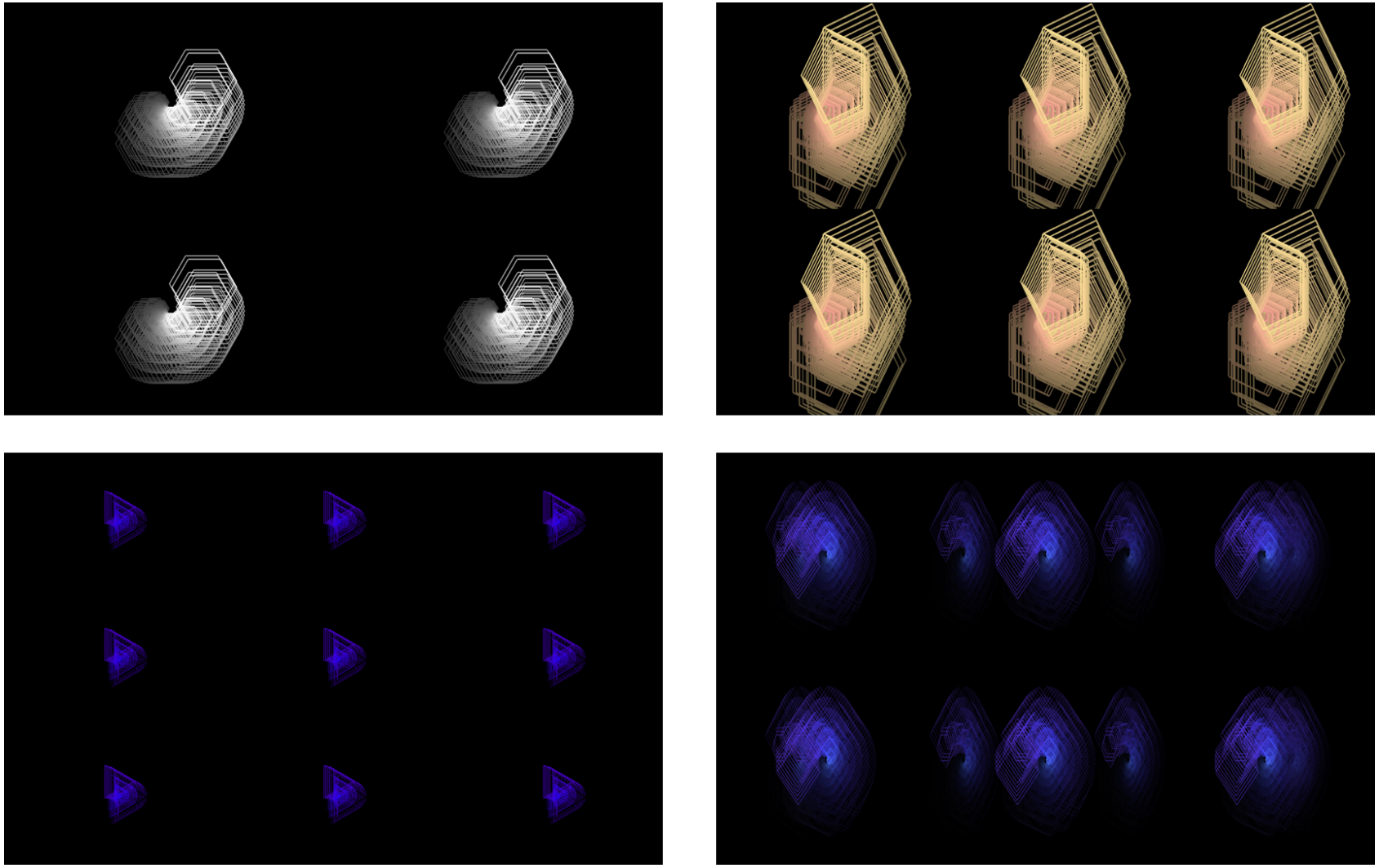


Figure 2: Results of Visualize

References

Cooper, M., Foote, J., Pampalk, E., Tzanetakis, G. (2006). 'Visualization in audio-based music information retrieval', Computer Music Journal, 30(2), 42-62.

Florian, E., Martin, W., and Bjorn, S. (2010) 'Opensmile: the munich versatile and fast open-source audio feature extractor', Proceedings of the 18th ACM international conference on Multimedia, 1459–1462.]

Mark, H., Eibe, F., Geoffrey, H., Bernhard, P., Peter, R., and Ian, H.W. (2009) 'The weka data mining software: an update', ACM SIGKDD explorations newsletter, 11(1), 10–18.

Street, Z., Albornoz, A., Bell, R., John, G., Jack, O., Knotts, S.,McLean, A., Smith, N.C., Tadokoro, A., van der Walt, J.S., Velasco, G. R. (2019) 'Towards Improving Collaboration Between Visualists and Musicians at Algoraves', International Conference on Live Coding.

Filling In: Livecoding musical, physical 3D printing tool paths using space filling curves

Evan Raskob
Goldsmiths, University of London
e.raskob@gold.ac.uk

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

This paper explores how space-filling curves such as Hilbert Curves can be used for live, improvisational making in livecoding performances because of the predictable ways in which they fill up space without crossing over themselves. It explores current research into 2D space filling algorithms known as infills and how these relate to live, improvisational making. A livecoding system for 3D printers called LivePrinter is introduced and used to demonstrate different aspects of live performance with printers. These examples suggest some practical methods for working with Hilbert curves in a performance setting based, along with avenues for further research into livecoding performances using space-filling curves.

3D Printing as Performance

Livecoding can be a means for directing the physical movements of machines, such as robots and 3D printers. This change in media from sound and screen to machine presents some unique challenges. For 3D printing, one of the biggest challenges is how to improvise when making new forms out of extruded lines of plastic, without accidentally destroying those forms in the process. This means moving the printing tool head safely and without hitting previously constructed structures.

3D printing: a type of type of Computer Numerical Controlled (CNC) (i.e. digital) manufacturing technology called Additive Manufacturing (AM). More specifically, it is called Fused Deposition Modeling (FDM) or sometimes Fused Filament Deposition (FFD). FDM (as we will refer to it):

1. Is extrusion-based: physical objects built up by depositing layers of molten plastic material

2. Uses a motor to force a thin plastic filament through a hot extruder, melting it in the process (Turner 2014). Molten plastic sticks and bonds to previous layers, forming a solid

3. Is simple and safe enough process to be utilised and further adapted by maker spaces, universities and small businesses worldwide (Gao et. Al 2015)

4. Utilises a digitally-controlled print head or hot end encapsulating the extruder, moving in 3D space inside the printer cavity

5. Often requires 4 separate motors: Two motors position the print head relative to the print bed in x and y directions where x is side-to-side, y is front to back; one motor moves the print surface or print bed in the z direction, up and down. Another motor (e) feeds the plastic filament through the print head, also pulling it back at times to prevent unwanted material leakage in a process referred to as retraction.

As these motors spin, they vibrate and make sound. People have used 3D printers to make music, notably the Imperial March from Star Wars and, less notably, Nickelback (3D Print 2014). A library for the Python language is even available to convert MIDI note numbers to motor frequencies (Westcott, 2015). The sounds of the motors are relatively quiet but can be captured and then amplified using contact microphones and audio amplifiers. In a 3D printing livecoding performance, the performer choreographs the movements of the printer (speed, direction, duration) and the properties of the printer itself (temperature, fan speed, filament flow rate) by manipulating and writing code. Both the act of making, with its resulting physical forms, and the sounds of making are intrinsic to the performance. This leads to a dual mode of composition when making music for printers, where one can prioritise the aesthetics of the form by composing movements in millimetres or the aesthetics of the sound by composing in milliseconds of movement at specific speeds (e.g. musical notes).

X axis	Y axis	Z axis
47.069852	47.069852	160.0

Table 14.1: Typical speed scale for axis values for the motors used in the Ultimaker 2 printers, from Westcott's MIDI-TO-CNC library (Westcott 2015). Note that no values were given for the filament feeding (e-axis) motor.

The 4 digitally-controlled motors: they are the same model of motor and have identical properties. When the motors spin, they emit sound that can be mapped to notes in the equal temperament scale used by MIDI synthesizers using the following ES6-like pseudocode:

```
// calculate the frequency of the note from
// MIDI note number:

frequency = Math.pow(2.0, (note - 69) / 12.0)
    frequency = frequency * 440.0
// convert to motor speed in millimetres
// per second for GCode (see Table 1)

speed = frequency / speed_scale_for_axis;       (1)
```

Knowing the travel speed of each motor that will produces a desired musical note, along with the desired duration of that note, one can calculate the distance of travel across each axis by using a simple movement equation:

```
d = st              (2)
```

where d is the distance in mm to be calculated, s is a scalar representing the speed of the print head in mm/s in the current direction of travel, and t is the desired movement time in seconds.

In LivePrinter, this is simplified into two functions, one called m2s(NOTE) to translate a MIDI note NOTE to motor speed for an axis, the other called t2d(TIME) to use that speed and a desired duration of movement TIME to calculate distance. The following ES6

code uses these two functions to move the print head making a pitch of MIDI note C5 with a duration of 1 second (1000ms):

```
await lp.m2s(72).t2d(1000).go();                    (3)
```

Normally, the speed at which the printer prints is determined by the desired quality of the print balanced by the total time of manufacture. Slower printing times generally lead to higher quality prints because the print head can follow a more precise path and the layers have time to cool properly before the next layer is applied on top (3D Matter 2015). When making music with printers, the relationship between form and making time is subverted. Performance time and musical quality can have as much weight as the quality of the finished object, whatever that final output might be in such a live setting. Slower speeds that might produce higher quality prints may be either inaudible or in the wrong key or frequency for the piece being performed.

This leads a maker/performer to reframe manufacturing as a mainly durational activity. Instead of describing objects in the usual way using technical drawings or digital models specifying physical dimensions in millimeters, one can consider objects using their durational dimensions and specify them in terms of the speed, angle and duration of movement used to manufacture them.

This tightly integrates the making of the object with the description of the object itself. It stands in opposition to the process planning approach that separates out a design concept from its fabrication processes.

Space Filling Curves Using Liveprinter

One solution to the problem of creating continuous, rhythmic movements in live 3D printing is a Hilbert curve:

- Originates with G. Peano (Peano 1890) &amp David Hilbert (Hilbert 1891; Sagan 1994).
- Curves map a one-dimensional space to a multi-dimensional space by passing through each point in that space once and only once.
- Generates tool paths which do not cross back over themselves
- Has a compelling visual aesthetic and mechanically rhythmic properties, as shown here from our experience performing with them
- Tool movements following a Hilbert curve that fill up spaces in predictable ways

Creating the curve

Curves can be represented as Lindenmeyer Systems (L-systems). There are ways to directly calculate a mapped point in n-dimensional space given an initial Hilbert index and a desired resolution.

These are less useful for describing tool paths because the information about how to move from point to point, i.e. the rotations and directions of movements encoded into an L-System string, are lost and must be re-calculated. The fact that an L-System encodes a literal set of movement instructions for a 3D printer, without need for matrix multiplications or any other form of interpolation, makes it attractive as a means for generating forms.

The LivePrinter example (Raskob 2019) uses a starting axiom of L and replacement rules of:

```
L: +RF-LFL-FR+                (4)
```

```
R: -LF+RFR+FL-                (5)
```

Finally, symbols are iterated in order and mapped to drawing functions written in ECMAScript 6 (ES6):

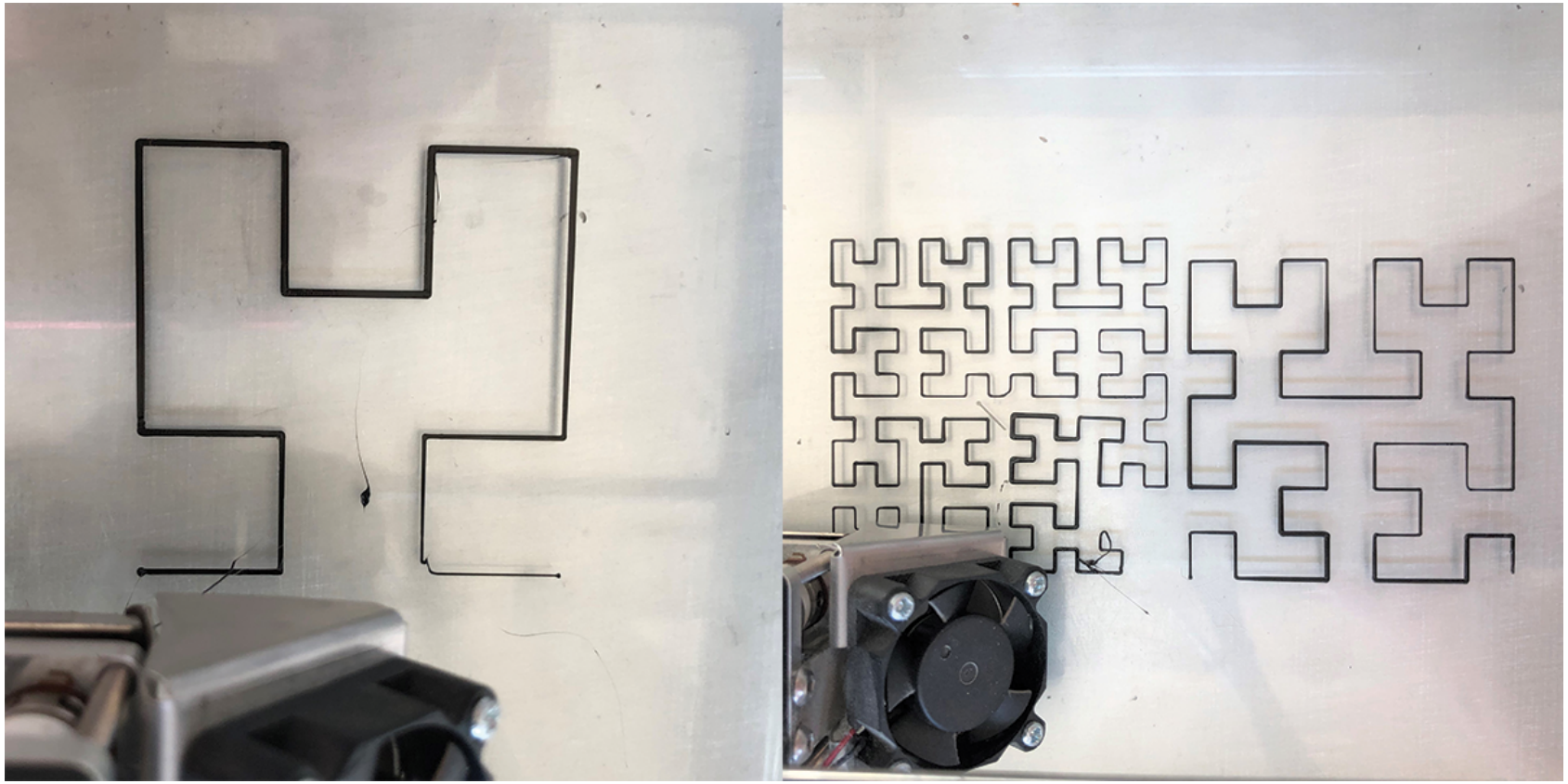


Figure 1: Three iterations of the Hilbert curve printed on a 3D printer. On the left is one iteration, on the far right is two iterations and on the middle right is 3 iterations. The code for generating these can be found in the LivePrinter repository `https://github.com/pixelpusher/liveprinter/blob/master/liveprinter/static/examples/hilbert.js`

```
'F': async () => { await lp.m2s(MIDI_NOTE).t2d(
    DURATION).go(1, false);
return 1; }
                                        (6)
'+': async () => { lp.turn(-90); return 0; }     (7)
'-': async () => { lp.turn(90); return 0; }      (8)
```

Note that:

- L and R symbols are ignored
- In (4,5,6), the lp object is an instance of the LivePrinter API object that converts drawing instructions to GCode
- In (6) m2s(MIDI_NOTE) is a function converting a MIDI note number to a motor speed as described above
- In (6) t2d(DURATION) is a function converting a movement duration in milliseconds to a movement distance in millimeters based on the motor speed as described above go(1, false) compiles the current drawing instructions to GCode and sends it to the physical printer, with the argument 1 meaning that the printer should be extruding during movements and the second argument false disabling retraction so the printer will not pause, retract, and then unretract after each drawing operation. lp.turn() is a virtual operation that rotates the current drawing direction by an amount in degrees, clockwise.

These drawing commands are similar to the transactional, embodied model of Logo's Turtle Graphics (MIT 2015). A drawing cursor (in this case, a 3D printer head) is directed to move and "draw" using short, intuitive statements that describe movement and drawing operations. A full API can be found in the documentation on the project website[1].

[1] https://github.com/pixelpusher/liveprinter

With LivePrinter, drawing a triangle with sides of 100mm could be achieved with the 3 lines of ES6 code:

```
await lp.turnto(0).dist(100).go(1);       (9)
await lp.turn(-120).dist(100).go(1);      (10)
await lp.turn(-120).dist(100).go(1);      (11)
```

This can be specified more concisely in the LivePrinter minilanguage as:

```
# turnto 0 | dist 100 | go 1              (12)
# turn -120 | dist 100 | go 1    (13)
# turn -120 | dist 100 | go 1    (14)
```

Of course, this assumes that the printer is prepared to draw with the print head having been heated up to the melting temperature of the plastic filament, the printing speed set, and the head positioned properly.

Filling Up Space With Plastic

Ordinarily the 3D printing process starts with a digital model created in a Computer Aided Design (CAD) program that defines the geometry of the object that is to be printed. This geometry mostly defines the outer and inner surfaces of the object, leaving the process of describing how the internals are to be manufactured to another piece of software called the "slicer".

The slicer determines the layer-by-layer construction of the object based on the properties of the model of printer to be used in that construction process. The process of starting with a model, optimizing the geometry, and preparing the machine instructions for manufacturing is often referred to as "Process Planning" (Thompson 2007; Livesu et. al. 2017).

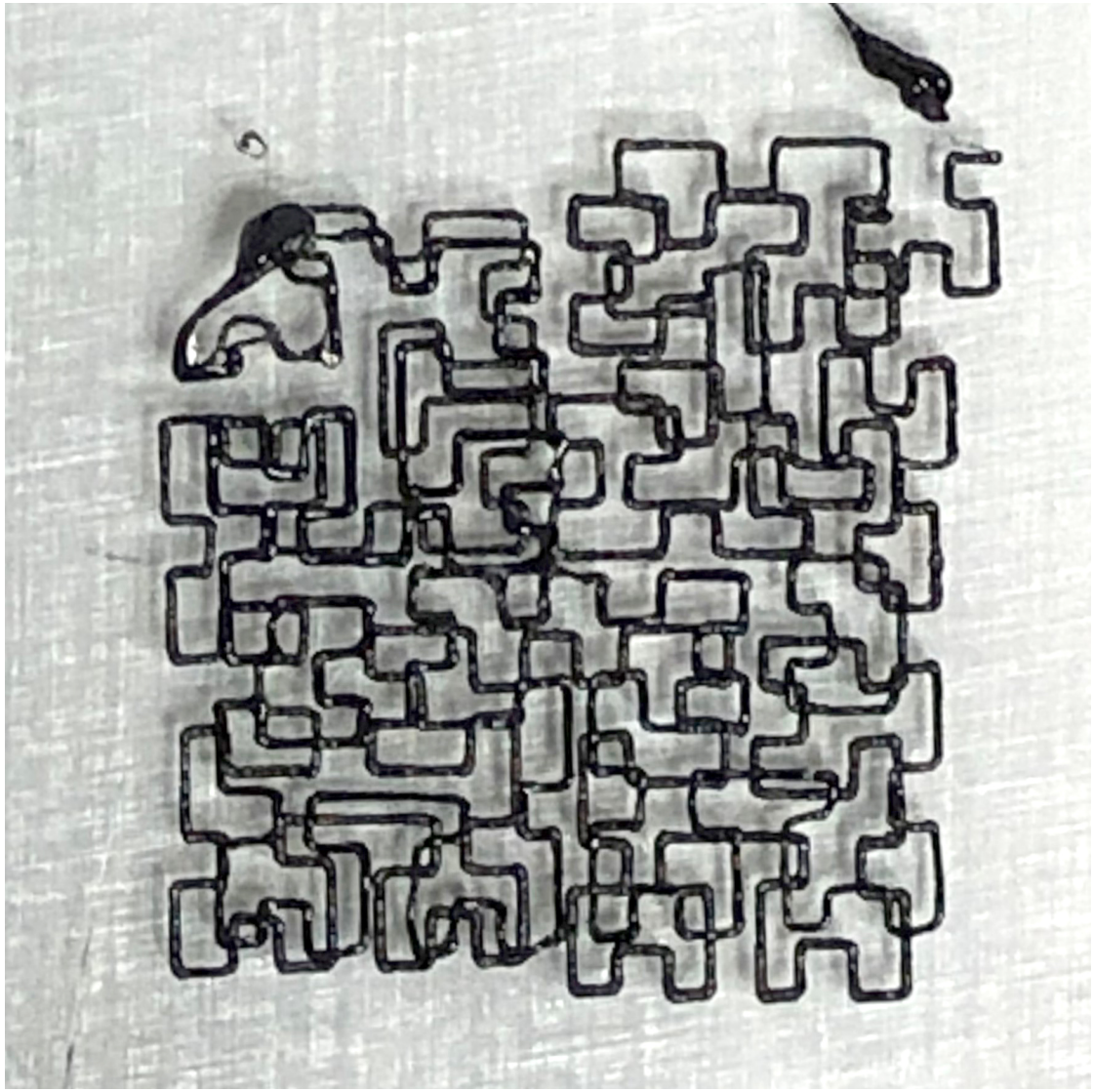


Figure 2: Two layers of Hilbert curve techno with random notes printed on a 3D printer. The code for generating these can be found in the LivePrinter repository `https://github.com/pixelpusher/liveprinter/blob/master/testing/hilbert_livecoding/hilbert-experiment-01.js`

The problem of how to efficiently generate 2D toolpaths that fill up 3D space in a structurally sound way whilst minimising time, material and movement remains an open area of research (Jin et. al. 2004; Jin et. al. 2017; Ding et. al. 2016). Current software packages offer a variety of filling patterns, from linear stripes to diamonds to vector patterns. Recent research has also looked at the properties of the Hilbert curve for filling spaces both rectangular and irregular (Ding et. al. 2016; Papacharalampopoulos et. al. 2018).

Motorised Hilbert Curves as Techno Perfromance

For 3D printing and music-making, the durations of movements and silences are of paramount importance to the musical aesthetics of the piece. As with any movement that creates music, a performer must control it precisely. Any gaps in movement or extraneous movements to position the printing head become part of the performance, for better or worse. A performer needs predictable tool paths at their disposal to improvise with, much as a jazz musician riffs on different musical scales and motifs. Continuous curves such as the Hilbert can be useful in that respect. As Papacharalampopoulos et. al. 2018 observed, Hilbert curves keep the print head moving throughout their length and thus minimize or altogether remove any extra waiting between operations and travel times needed to reposition the head after movements.

With dense Hilbert curves of higher orders, the number of movements can be quite large which is helpful to a livecoding performer trying to compose a dense melody live. They are especially useful when the printer movements are quick, giving them more time to think as they type new lines of code.

For example, a performer might wish to play a sequence of MIDI C5 notes (MIDI number 72) every beat, at a tempo of 120 beats-minute (or 0.5 seconds-per-beat) for a 12-beat segment. That means each beat the print head would be moving at a speed of 11.1165 millimetres-per-second for a distance of 5.5582 millimetres. A second-order Hilbert curve could completely contain that movement because the curve is made of 15 segments packed into a space that is 3 segments by 3 segments square, or 16.6747mm by 16.6747mm which will take a total of 6 seconds to play (12 notes at a duration of 0.5s per note). This whole curve fills up a little over 13% of the total printer bed space at that level, leaving room for about 8 repetitions of that motif before starting a new print layer.

Some calculations:

1. Printable area on the print bed is 200mm
2. 200mm/32 segments of 4th order Hilbert = 6.25mm per division
3. Gives us 1024 beats at 0.5 sec/beat –¿ 512 sec –¿ 8.533 minutes of music using the whole printer bed

Also, when the curve is oriented to the x/y axes of the print bed the tool head will move at right angles when drawing the curve. This uses only one motor at a time, playing distinct notes rather than chords. A performer could take advantage of this fact by alternating motor speeds with every segment of the curve, thus playing arpeggios. It is also possible to make more complex chords using rotated Hilbert curves. Rotating 45 degrees with respect to the x/y axes of the bed engages two motors simultaneously when moving, playing a two note chord consisting of the same two notes. A movement at any other angle produces a more complex tone that in practice can be difficult to control.

Future Challenges

In practice, it can be hard to manage Hilbert curves because the L-Systems representation of them grows exponentially with each iteration. This makes storing them as strings a non-trivial task, especially in a live, real-time performance setting.

Stepping through an entire curve in a performance is a dangerous task because it encompasses hundreds or even thousands of steps.

Some language-specific techniques are needed, such as using ES6 generators to write iterative Hilbert functions whose execution is not continuous (Mozilla 2019).

Additionally, more research is needed into the properties of other space-filling curves that have a number of segments that divide up evenly into common musical time signatures, like 4 and 8 beat segments. There are many other types of space filling curves that could have beneficial musical properties, including diagonal segments that represent chords.

Conclusions

Space filling curves can be a useful tool for live, improvisational physical making performances because of the ways in which they can completely fill up physical space with dense, aesthetically pleasing forms that do not cross over themselves during performance. Further study is needed to come up with methods for integrating different fill patterns over time, across unpredictable and improvised forms that arise during performance. This potentially has applications outside of sculptural performance, in the realm of industrial 3D printing, where 2D and 3D space filling patterns and techniques are current areas of research.

Acknowledgments

A special thank you to my PhD advisors, Prof. Mick Grierson and Dr. Rebecca Fiebrink, for all their support and feedback on this project.

References

Ding, Donghong, Zengxi Pan, Dominic Cuiuri, Huijun Li, and Stephen van Duin. 2016. 'Advanced Design for Additive Manufacturing: 3D Slicing and 2D Path Planning.' In New Trends in 3D Printing, edited by Igor V Shishkovsky. Rijeka: IntechOpen. https://doi.org/10.5772/63042

Gao, Wei, Yunbo Zhang, Devarajan Ramanujan, Karthik Ramani, Yong Chen, Christopher B. Williams, Charlie C. L. Wang, Yung C. Shin, Song Zhang, and Pablo D. Zavattieri. 2015. "The Status, Challenges, and Future of Additive Manufacturing in Engineering." Computer-Aided Design 69: 65–89. https://doi.org/https://doi.org/10.1016/j.cad.2015.04.001

Hilbert, D. 1891. Uber die stetige Abbidung einer linie auf ein Flaechenstueck. Math.Ann. 38, 459–460

Jin, Yuan, Yong He, Jian-Zhong Fu, Wen-Feng Gan, and Zhi-Wei Lin. 2014. 'Optimization of Tool-Path Generation for Material Extrusion-Based Additive Manufacturing Technology.' Additive Manufacturing 1-4: 32–47.

Jin, Yuan, Yong He, Guoqiang Fu, Aibing Zhang, and Jianke Du. 2017. 'A Non-Retraction Path Planning Approach for Extrusion-Based Additive Manufacturing.' Robotics and Computer-Integrated Manufacturing 48:132–44. https://doi.org/https://doi.org/10.1016/j.rcim.2017.03.008

Livesu, Marco, Stefano Ellero, Jonàs Martínez, Sylvain Lefebvre, and Marco Attene. 2017. "From 3D Models to 3D Prints: An Overview of the Processing Pipeline." Computer Graphics Forum 36 (2): 537–64. https://doi.org/10.1111/cgf.13147

Matter, 3D. 2015. https://my3dmatter.com/what-is-the-influence-of-color-printing-speed-extrusion-temperature-and-ageing-on-my-3d-prints/

Matt Westcott. 2015. MIDI-TO-CNC. Retrieved Sept. 3, 2019 from https://github.com/gasman/MIDI-to-CNC/blob/master/mid2cnc.py

Milkert, Heidi. 2014. (Dec. 2014). 3D Printers Play Music from Mario Bros., Star Wars' Imperial March & More. Retrieved Sept. 3, 2019 from https://3dprint.com/29244/3d-printer-music-songs/

MIT. 2015. 'Logo History.' https://el.media.mit.edu/logo-foundation/what_is_logo/history.html%0A

Mozilla. 2019. 'Iterators and Generators.' https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators

Papacharalampopoulos, Alexios, Harry Bikas, and Panagiotis Stavropoulos. 2018. "Path Planning for the Infill of 3D Printed Parts Utilizing Hilbert Curves." Procedia Manufacturing 21: 757–64.https://doi.org/https:/doi.org/10.1016/j.promfg.2018.02.181

Peano, G.: Sur une courbe qui remplit toute une aire plane. Math. Ann. 36, 157–160 (1890)

Raskob, Evan. 2019. LivePrinter; Livecoding for 3D printers. (Sept. 2019). Retrieved Sept. 3, 2019 from https://github.com/pixelpusher/liveprinter

Sagan H. 1994. Hilbert's Space-Filling Curve. In: Space-Filling Curves. Universitext. Springer, New York, NY

Sorensen, Andrew, Ben Swift, and Alistair Riddell. 2014. "The Many Meanings of Live Coding." Comput. Music J. 38 (1): 65–76. https://doi.org/10.1162/COMJ_a_00230

Thompson, Rob. 2007. Manufacturing Processes for Design Professionals. London: Thames & Hudson.

Turner, Brian N., Robert Strong, and Scott A. Gold. 2014. 'A Review of Melt Extrusion Additive Manufacturing Processes: I. Process Design and Modeling.' Rapid Prototyping Journal 20 (3): 192–204. https://doi.org/10.1108/RPJ-01-2013-0012

Zhao, Haisen, Fanglin Gu, Qi-Xing Huang, Jorge Garcia, Yong Chen, Changhe Tu, Bedrich Benes, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. 2016. 'Connected Fermat Spirals for Layered Fabrication.' ACM Trans. Graph. 35 (4): 100:1–100:10. https://doi.org/10.1145/2897824.2925958

Functional Live Coding vs. DAWs and VSTs

Dr. Nick Rothwell
Ravensbourne University London
n.rothwell@rave.ac.uk

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Creative Context: Sitar, Live Coding, VSTs

- Khyal Geometries: Shama Rahman (sitar), Nick Rothwell (software and processing)
- Non-linear environment, combining live coding with commercial VSTs and a matrix mixer
- VSTs address audio-processing tasks without re-inventing the wheel
- VST control via code seems like a fertile area of creative exploration

Tools: Max, VSTs, Node.js, ClojureScript

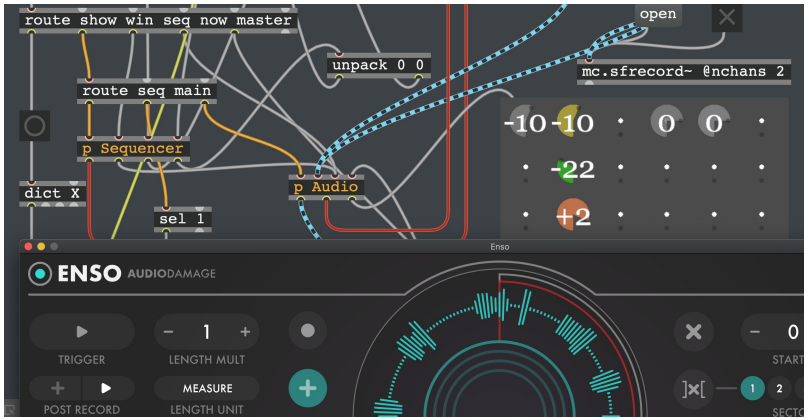


Figure 2

- Max as VST host and audio mixer
- Embedded Node.js acting as a parameter controller for VSTs and mixer
- ClojureScript live-coded via NREPL from Emacs

Figure 1

- No direct interaction with Max patchers or instruments–just on-screen for visual monitoring

Inspiration: Overtone, Gibberwocky

```
// and eighth notes vs. 1/16th notes. If 1/16th
// notes are played, always play two back to back.
tracks[0].midinote.seq( [64,65].rnd(), [1/8,1/16].rnd(1/16,2), 1 )

// play a scintillating bass line
tracks[1].note.seq( [-14,-12,-9,-8], 1/8 )

// play chords with piano sound
tracks[2].chord.seq( Rndi(0,8,3), 2 )

// control bass filter cutoff
tracks[1].devices[0]['Filter Freq']( mul( beats(2), .75 ) )
```

Figure 3

- Step sequencing from generative pattern expressions
- Table/map lookup for device parameters
- Gibberwocky's visual feedback (spark-lines etc.) replaced by Max's graphical interface
- Choice of Clojure for immutable data structures: ability to reverse changes or jump between cues, useful for improvisation and rehearsal

Issues: Resetting and Recalling Devices

```
;; Enso
(go
  ;; RESET
  (do (<! (ctrl/reset-device :Enso))
      (<! (px/request-params PARAMS :Enso)))
  ;; CONFIGURE
  (do
    (<! (px/xmit-some-params-now :Enso
                                 [:Link_Speeds :Off]
                                 [:Play_Speed :+1.0]
                                 [:Rec_Speed :-0.5]
                                 [:Dry_Level 0]
                                 [:Feedback 0.5])))))
```

Figure 4

- Conceptual clash between immutable data of Clojure and editable state of virtual instruments and effects
- Edits can be tracked, but are not always reversible–cannot emulate immutability
- Complete resets to known states (initial, preset)–discontinuous (sudden audio changes), so need to track levels of signal paths (when it is safe to recall a state?)
- Future directions: more sophisticated state tracking needed

References

Grosse, Darwin (2019). Node For Max Intro – Let's Get Started! January 2019. Available at `https://cycling74.com/articles/node-for-max-intro-%E2%80%93-let%E2%80%99s-get-started` (accessed [2019-09-22 Sun]).

FARM (2019). ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design, Berlin, 18-23 August. Available at `https://icfp19.sigplan.org/home/farm-2019` (accessed [2019-09-22 Sun]).

Roberts and Wakefield (2016). Live Coding the Digital Audio Workstation. International Conference on Live Coding, Hamilton, Canada.

McGranaghan (2012) Mark McGranaghan. ClojureScript: Functional Programming for JavaScript Platforms. 2012 IEEE Internet Computing 15(6):97 - 102.

Hauman (2015) Developing ClojureScript with Figwheel. Clojure/West, Portland, Oregon, April 20-22 2015.

Brannigan, Erin (2007) Music Makes the Moves. RealTime issue 76, Dec 2006-Jan 2007. Available at `http://www.realtimearts.net/article/76/8266` (accessed [2019-09-22 Sun]).

Hickey, Rich (2016). Clojure core.async. Strange Loop, November 2013.

Hoare, C.A.R. (1978). Communicating Sequential Processes. Communications of the ACM 21/8, August 1978.

Redux: A predictable state container for JavaScript apps. Available at `https://redux.js.org/` (accessed [2019-09-22 Sun]).

Transient Data Structures. Available at `https://clojure.org/reference/transients` (accessed [2019-09-22 Sun]).

Visor in Practice: Live Performance and Evaluation

Jack Purvis
Victoria University of Wellington, New Zealand
jack.purvis@ecs.vuw.ac.nz

Craig Anslow
Victoria University of Wellington, New Zealand
craig@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington, New Zealand
kjx@ecs.vuw.ac.nz

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

Abstract

Visor is a new environment for live visual performance that was developed to demonstrate code jockey practice (CJing), a new hybrid performance practice that combines live coding and VJing to harness the strengths of both practices. CJing draws on live coding for the ability to improvise content at a low level by coding in textual interfaces. VJing is drawn on for its ability to manipulate content at a high level by interacting with GUIs and hardware controllers. Combining these aspects of both practices enables flexible performances where content can be controlled at both low and high levels. We build on previous work by reflecting on the use of Visor in live performances and evaluating feedback gathered from creative coders, live coders, and VJs who experimented with the environment. We conclude by discussing Visor's effectiveness and whether CJing effectively combines live coding and VJing along with areas for future work.

Introduction

The creation of visuals to accompany music is an essential part of any audiovisual experience. Live coding and VJing (video jockey practice) are live performance practices that offer the ability to improvise and manipulate visuals that synchronize with music in real-time. Live coding makes use of live programming techniques, enabling code to be executed at runtime with immediate feedback (Tanimoto 2013). Live coding is used to explore how algorithms can generate music or visuals and is typically performed live at events called Algoraves (Collins et al. 2003). Video jockeys (VJs) are live visual artists that mix content in real-time, often complimenting music played by disc jockeys (DJs) to create audiovisual marriages that engage the senses (Faulkner and D-fuse 2006). VJs typically perform by layering multiple video clips together, applying video effects, and interacting with effect parameters using hardware devices such as MIDI controllers.

Live coding focuses on writing code to improvise or manipulate content. This focus provides fine-grained, low level control of visuals,

but does not provide high level control, impairing usability as all interactions must occur through a textual interface. VJs instead focus on interacting with comprehensive graphical user interfaces (GUIs) and hardware controllers to improvise or manipulate content. This focus provides overarching, high level control of visuals, but does not provide low level control, preventing improvisation of content from scratch or the ability to make fine-grained adjustments to existing content. The CJing practice was introduced to overcome the limitations of live coding and VJing by combining the practices together (Purvis et al. 2019). In CJing, a performer known as a code jockey (CJ) interacts with code, GUIs, and hardware controllers to improvise or manipulate visual content in real-time. CJing harnesses the strengths of live coding and VJing to enable flexible performances where content can be controlled at both low and high levels. CJing has been demonstrated by Visor, a new live coding environment that embodies the practice (Purvis et al. 2019). Visor has been purpose-built following a practice-based, user-centered approach to offer features for both live coding and VJing to enable live visual performances. To determine the effectiveness of Visor and whether CJing can effectively combine live coding and VJing, in this paper we reflect on Visor's use in live performances and evaluate feedback gathered from creative coders, live coders, and VJs who experimented with the environment.

Background

CJING

Code jockey practice (CJing) is a new hybrid performance practice that was first proposed by Purvis et al. (2019). CJing harnesses the strengths of live coding and VJing to enable flexible performances while simultaneously removing limitations identified in each practice. In CJing, a performer known as a code jockey (CJ) interacts with code, GUIs, and hardware controllers to improvise or manipulate visual content in real-time. CJing is designed to complement live coding and VJing by providing a new approach that enables performers to utilise aspects of both practices in the same performance. For example, an aesthetic of CJing practice is the utilisation of live coding as a method to improvise 'visual instruments' on the fly. Once defined, visual instruments can be performed using GUIs and hardware controllers to generate live visuals.

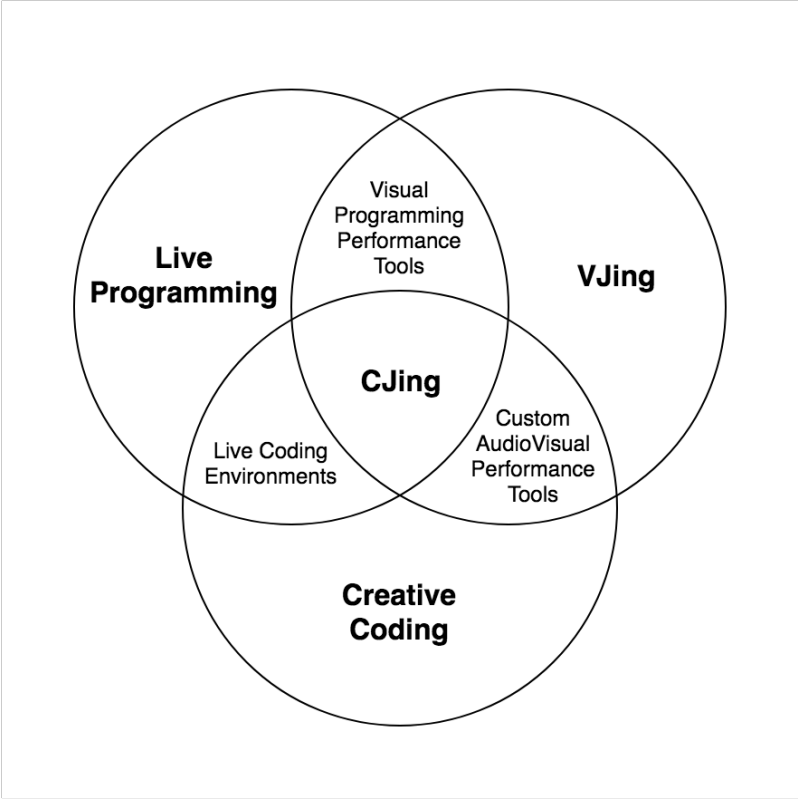


Figure 1: "Software that supports CJing lies at the intersection of creative coding, live programming, and VJing related software tools." (Purvis et al. 2019)

CJing practice is formulated from the broader subject areas of

creative coding, live programming, and VJing (Purvis et al. 2019). As shown in Figure 1, software that supports CJing is placed at the intersection of these subject areas. Purvis et al. (2019) proposes that CJing practice is based on three key ideas:

1. Code as a universal language: Creative coding should be used to produce content from scratch, while live coding enables the coded content to be manipulated on the fly at runtime. In CJing, code is the predominant content that is used to instruct the visual output, similar to how musical tracks are treated in DJing, and how video clips are treated in VJing.

2. Complete content control: CJing should allow for content to be manipulated at both low and high levels, enabling flexible control of the visual output during performances. The low level aspect is provided by live coding, allowing content to be improvised or edited using code. The high level aspect is provided by VJing, allowing content to be organised into layers and manipulated through effects, parameters, or hardware controls.

3. User interfaces as an abstraction: Code should be abstracted upon by user interfaces, providing high level functionality that would not be easy to achieve by simply writing code. Therefore, CJing software maintains a relationship between user interfaces and code, for example, providing contextual interfaces that detect when changes occur to the code and update themselves accordingly. At the same time, code should be able to access the state of interactive user interfaces.

VISOR

Visor[1] is a new hybrid environment that was developed with the primary goal being to embody CJing, exploring how live coding and VJing can be combined into a single environment to harness the strengths of both practices (Purvis et al. 2019). Visor achieves this goal by offering a number of core features to facilitate live performance: Live coding: Visor offers the ability to live code visuals with the Processing API (Reas and Fry 2006) in the Ruby language.

- State management: Visor supports a state management interface that automatically visualises and provides GUIs to update live coded variables. For example, numeric variables are presented as sliders with configurable ranges and boolean variables are presented as checkboxes, enabling parameters to be manipulated at a high level without the need to write any code.

- Layers: Visor offers the ability to organise code into multiple layers that are composited together into the final visual output using a variety of blend modes. Each layer acts as an independent Processing sketch with its own state and draw loop.

- FFT: Visor supports the fast Fourier transform (FFT) algorithm to generate a frequency spectrum from an audio input in real-time. The spectrum is visualised in the interface and made accessible in the code, enabling audio reactive visuals. Tap tempo: Visor offers the ability to set a tempo that can be referenced in the code to animate visuals to the beat of live music. The tempo can be set by repeatedly clicking a button in the interface or by using a keyboard shortcut.

- MIDI: Visor supports a framework for configuring inputs from external devices such as MIDI controllers using the MIDI protocol. The framework enables sliders, knobs, and buttons to become directly accessible in the code or mapped to state parameters. Visor supports a number of other notable features including support for multiple code tabs, support for multiple display outputs, a console, an in-app tutorial, and in-app documentation. A number of existing features were also refactored since Purvis et al. (2019), most notably, the REPL editor and

[1] `https://www.visor.live`

the draw loop editor were merged into a single editor. The implication of this is that the code that should be executed every frame should now be scoped within a method called draw, akin to how Processing traditionally operates. Figure 2 shows Visor in action by presenting the Visor interface and corresponding visual output.

Design Methodology

Visor was developed using a practice-based approach where the environment was tested in a performance context throughout development to ensure it met the needs of a performer (in this case, the first author). Unlike conventional software engineering processes, this approach considers the act of developing software as a form of craft research (Blackwell and Aaron 2015) and places emphasis on the need for 'research through design' (Gaver 2012). Using Visor in a performance context during development meant that the effectiveness of the environment could be evaluated iteratively: Existing features could be validated, features that needed improvement could be refined, and ideas for new features could be generated.

Practice-based approaches have been demonstrated by the Sonic Pi and Palimpsest environments (Aaron 2016; Blackwell 2014), both of which were developed with a consideration for craft practice and were discussed in the context of their use in live performance. Here we take a similar approach by reflecting on the use of Visor in live performances, as presented in Section 3. Visor is also evaluated following a more conventional user-centered approach (Abras et al. 2004) by analysing feedback collected from respondents of an online survey, similar to the approach taken to evaluate the Gibber live coding environment (Roberts et al. 2014). This evaluation is presented in Section 4.

Related Work

Figure 3: Mainstage of the Taniwha's Den 2019 music festival. The rendered visuals are projected across multiple screens around the DJ booth using multiple projectors. The VJ booth is situated behind where this photograph was taken.

There are a number of software applications that support features that coincide with Visor or the broader CJing practice. Sonic Pi (Aaron 2016) is a Ruby based live coding environment for creating music that focuses on simplicity to excel in computing education. Mother (Bergström and Lotto 2008) is an extension to Processing that enables VJing performances where multiple sketches can be layered together to instruct the final visual output. Auraglyph (Salazar 2017) is a live coding environment that supports visual programming of music using a touch screen interface, offering gestural manipulation akin to using a MIDI controller. Praxis LIVE (Smith 2016) is an audiovisual live coding environment and IDE that supports creative coding with Processing and offers GUIs that work hand in hand with coded content. Siren (Toka et al. 2018) is a hybrid system for the

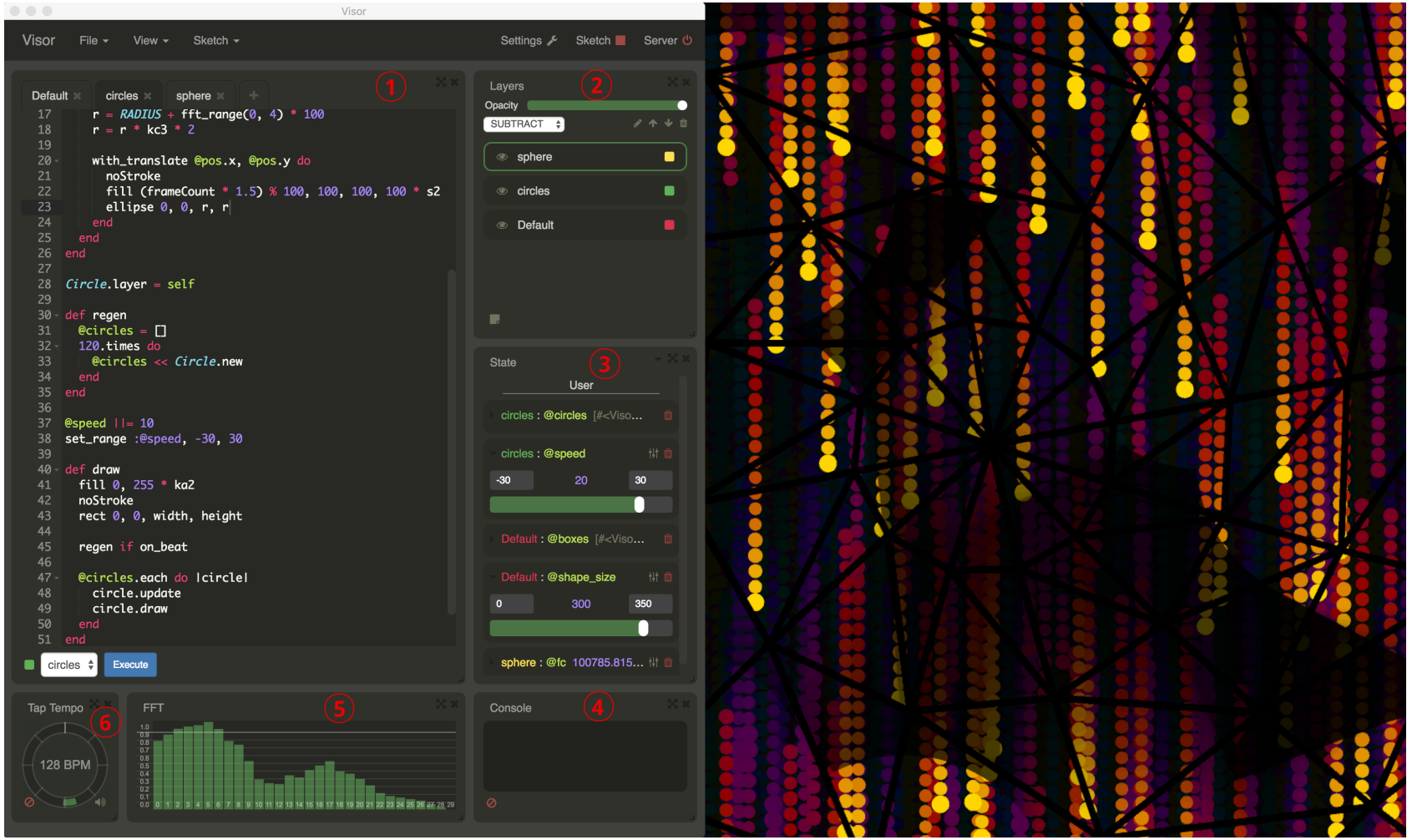


Figure 2: Visor in action. The interface (left) is made up of multiple GUI elements including a live code editor (1), layer interface (2), state management interface (3), console (4), FFT display (5), and tap tempo interface (6). The visual output corresponding to this state of the interface is also shown (right).

composition of algorithmic music and live coding performances that offers GUIs to interact with code. Resolume[2] is a widely used tool for VJing that supports video mixing, a variety of effects, and allows parameters to be animated to music in real-time using an FFT or tap tempo.

Live Performances

Visor was used by the first author in 20 live performances to reflect on the effectiveness of the environment as part of a practice-based approach. The intent of the performances was to provide visuals to accompany music performed by DJs, live coders, and other musicians, demonstrating Visor's effectiveness in a live context. The performances were conducted alongside a variety of collaborators at a variety of different events including gigs, algoraves, livestreams, exhibitions, research group meetings, private parties, and music festivals. By using Visor in real performances, we explored and demonstrated what it meant to perform with an environment that embodies CJing. We now describe the typical performance setup, approach to using Visor in live performances, and two notable issues concerning CJing that were observed.

Performance Setup

The typical performance setup consisted of a number of hardware and software components. A MacBook Pro laptop was used to run the Visor software. A Novation Launch Control XL was used as a MIDI controller, offering 8 sliders, 24 knobs, and 16 buttons. Where available, a line-in from the sound desk was used as input to Visor's FFT, otherwise, the laptop microphone was used. Projectors were usually provided by the venue and were either connected directly to the laptop or routed through another computer running the Resolume VJ software. Resolume was used to projection map the rendered output of Visor onto complex surfaces. An example of this is shown in two performances that were conducted at the Taniwha's Den 2019 music festival. In the first performance, the visuals were projected onto multiple screens using multiple projectors, as shown in Figure 3. In the second performance, the visuals were projected onto a large limestone cliff face, as shown in Figure 4. This offered a novel live coding and VJing experience.

Figure 4: Limestone cliff face that was used as a projection surface during the Taniwha's Den 2019 festival. Note the size of the people standing at the base of the cliff.

Performance Approach

The conducted performances were generally approached in one of two ways. The first approach was to live code from scratch and was most similar to a traditional live coding performance, due to starting with

[2] https://resolume.com/

[3] https://tinyurl.com/visor-toplap15/

an empty screen. This approach was used in 10 of the performances including the TOPLAP 15th Birthday[3] performance shown in Figure 5. This approach involved live coding the visual content throughout the course of an entire performance. This included the live coding of visual elements such as shapes, animations, and colours. Individual layers of content were created progressively and introduced, manipulated or removed at different times throughout the performance. Live coding also established mappings between parameters and MIDI variables, followed by the performance of these parameters on the MIDI controller. This approach showcased the performance aesthetic of the CJing practice where live coding can be used as a method to improvise visual instruments that are then performed using GUIs and hardware controllers.

The second approach was to perform with prepared code. This approach involved coding the visual content in preparation for the performance and was most similar to a traditional VJ performance, due to primarily making use of existing content. This approach was used in 10 of the performances including the Taniwha's Den[4] performance shown in Figure 3. This approach involved organising visual elements into layers where parameters of each layer were assigned to groupings of controls on the MIDI controller. These performances mostly focused on interaction with the MIDI controller as the content and MIDI mappings had been defined in advance. Live coding also occurred during these performances to improvise content or to manipulate the existing content, for example, to toggle predefined states and attach or detach parameters from the FFT and tap tempo.

In addition to these two approaches, during two performances, a collaborator was invited to perform alongside the author for a small section of each performance. This collaborator focused solely on interacting with the MIDI controller while the author focused on live coding and interacting with the GUI. Overall, these approaches demonstrate respectively how Visor can be used for live coding and VJing style performances. The crossover of these two approaches also highlights Visor's demonstration of the CJing practice where aspects of both live coding and VJing can be used together in the same performance.

[4] https://tinyurl.com/visor-taniwhasden-2019/

Reflection

Using Visor in live performances helped to identify general usability issues and aspects of the core features that could be improved. More notably, two issues were identified concerning the broader CJing practice. The first issue relates to the idea of user interfaces as an abstraction and highlights the need for careful consideration when designing the relationship between code and GUIs in CJing environments. This issue was observed when interaction with layers using both the GUI and the code would result in conflicting behaviour. For example, when the code to set the blend mode was specified (i.e. set_blend_mode), the blend mode set through the GUI could be unintentionally overridden by the executing code. To avoid this behaviour, the live performances almost exclusively used the GUI to set blend modes. This issue highlights the importance of the relationship between code and user interfaces in CJing environments: careful consideration should be taken when designing a flexible CJing environment where both live coding and VJing can be used to interact with specific features.

The second issue relates to the idea of complete control and was the need to switch contexts between live coding and using the MIDI controller. This issue was also observed in the user feedback presented in Section 4. As both contexts required almost full attention, it seemed impossible to live code and perform with the MIDI controller at the same time. This was mostly observed in the early stages of the "from scratch" performances. In these performances, parameters of existing content could not be tuned using the MIDI controller while the focus was placed on live coding new content. The opposite holds true for later in the performance when the focus was placed on the MIDI controller and live coding was mostly used to make minor adjustments to

the existing content. This issue emphasises that CJs must not only develop their skills in live coding and using the controller, but must also learn to strike an effective balance when working across multiple modalities. This highlights the importance of automated features such as the FFT and tap tempo which continuously produce dynamic visual effects without requiring the attention of the performer.

An approach to mitigating the friction caused by context switching in CJing was observed when a collaborator performed alongside the first author during two performances. Utilising two performers meant that one performer could focus on live coding while the other focused on the MIDI controller, enabling content to be improvised from scratch while the parameters of existing content were performed at the same time.

Evaluation

An online feedback survey was constructed to evaluate the effectiveness of Visor as part of the user-centered design process (Abras et al. 2004). The survey solicited feedback from people with creative coding, live coding, and VJing experience who had used Visor. The Visor users who participated in the survey were asked to complete a questionnaire that asked about their background, their usage of Visor, their outlook on Visor's core features, how difficult they found Visor to learn, the context in which they might use Visor, and what they liked or disliked about Visor in general. These questions were formatted as either Likert scales, multiple choice, or free form text fields.

ID	Ruby experience	Processing experience	Live coding experience	VJing experience	Visor Time	Visor Context
P1	Little	Pro	Little	Little	1-5 hours	Performance (Live coding)
P2	Little	Little	None	None	1-5 hours	Performance (VJing)
P3	Little	Pro	None	Little	1-5 hours	Creative coding
P4	Little	Pro	Pro	Pro	5-10 hours	Teaching
P5	Fair	Fair	Little	Little	10+ hours	Performance (VJing)
P6	None	Pro	None	None	1-5 hours	Creative coding
P7	None	Fair	None	None	10+ hours	Performance (VJing)
P8	Pro	Little	None	None	1-5 hours	Creative coding
P9	Pro	Little	Little	None	10+ hours	Creat. cod. & Perf.
P10	None	Pro	Little	None	10+ hours	Performance (Live coding)
P11	None	Fair	None	None	1-5 hours	Teaching

Table 16.1: Feedback survey participants background, estimated time spent using Visor, and the context in which they might use Visor.

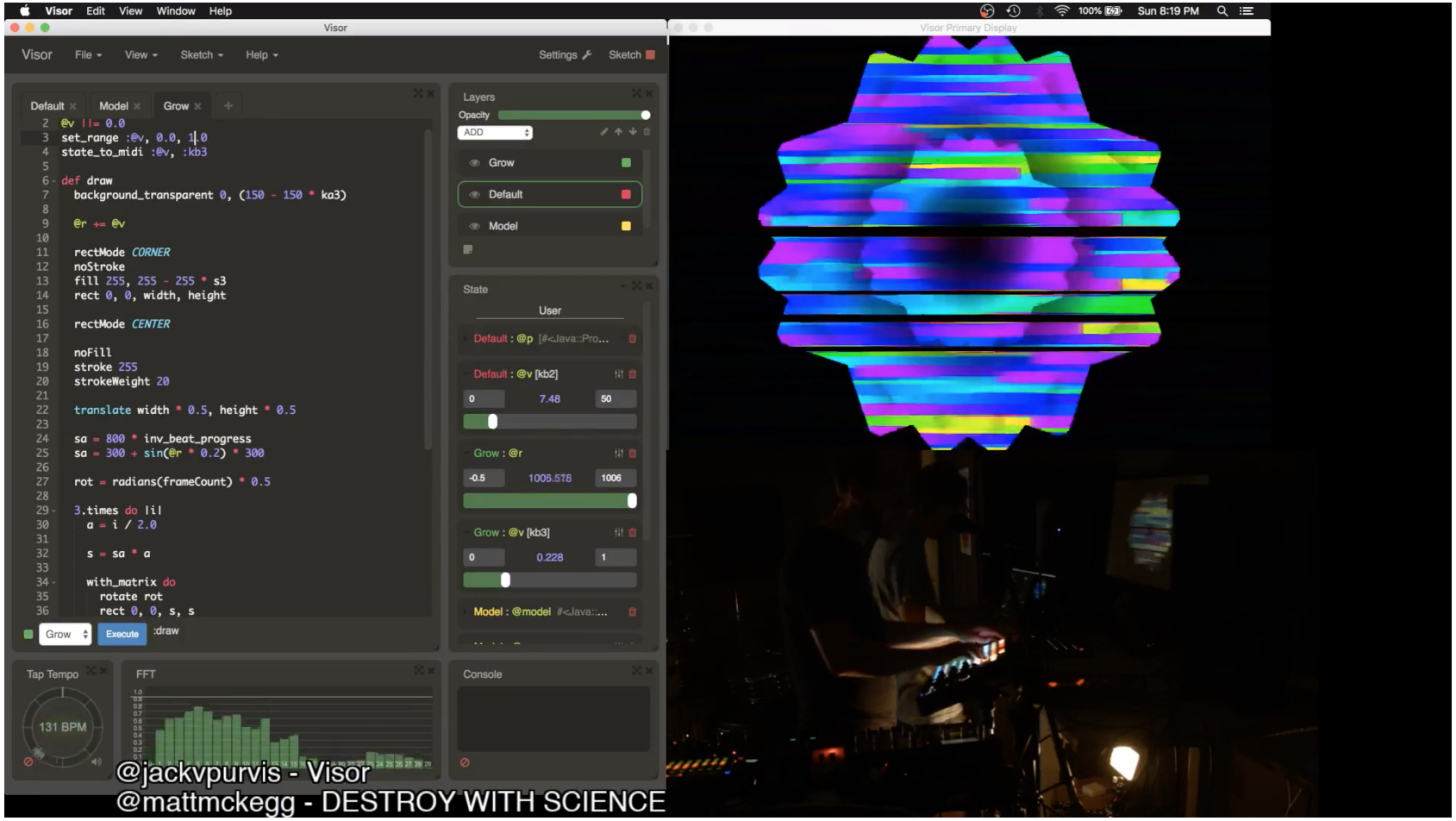


Figure 5: Screenshot from the livestream of the TOPLAP 15th Birthday performance. The Visor GUI (left) is displayed alongside the rendered visuals (top-right) and a camera recording of the physical performance by the first author and a musical collaborator, DESTROY WITH SCIENCE (bottom-right).

Participants

In total, 11 participants completed the feedback survey since it was launched in January of 2019. Participants were recruited through recruitment messages placed on the Visor website and within the software itself. Visor was advocated through various online forums, chat channels, and social media groups relating to live coding, creative coding, Processing, and VJing. The environment was also advocated through the author's existing networks of live coders, creative coders, and VJs. Visor itself has been downloaded more than 900 times since January 2019.

Participants were asked to provide background information with respect to their experience with general programming, Ruby, Processing, live coding, and VJing. To answer these questions, participants could choose from the following options: no experience, a little experience, a fair amount of experience, or professional experience. The results are shown in Table 16.1. All of the participants reported having more than three years of programming experience except for P7, who had 1 to 2 years of experience.

Results

The results are grouped based on Visor's usage, learning difficulty, core features, and ease of use. The results about Visor's usage and learning difficulty were reported based on a combination of multiple choice questions and comments received in free form questions. The remainder of the results were reported based on direct quotes from free form questions. These questions asked participants how effective they found each of Visor's core features and why, as well as what features of Visor they enjoyed most or least. Usage: The participants were asked to report how much time they had spent using Visor and the context in which they might use Visor. The responses to these questions are also presented in Table 16.1. All of the participants reported using Visor for at least one hour while five had used it for five hours or more. P1 and P10 reported that they would use Visor for live coding new material in performance; P2, P5, and P7 reported that they would use Visor to VJ with pre-prepared material; P3, P6, and P8 reported that they would use Visor for creative coding; P4 and P11 reported that they would use Visor in teaching; and P9 reported that they would use Visor for creative coding and in both performance contexts. This variety of responses that were received for this question indicates the potential versatility for Visor to be used by different audiences including creative coders, live coders, and VJs. Two of the participants also made additional comments on their current or intended usage of Visor:

> "I see a lot of potential is this program, I'm trying to learn everything as soon as possible, and have people interested already [sic] in applying it in real clubs. I haven't had this much fun with a program in a while." (P7)

> "I've used visor for two creative coding projects. In one of them, I used visor + a genetic algorithm gem that I've published to "evolve" visualizations ... The project got a great response and I don't think I would have been able to pull it off so smoothly without Visor." (P9)

Learning difficulty: The participants were asked to report how difficult they found Visor to learn. Seven of the participants disagreed or strongly disagreed that it was difficult (P1, P2, P4, P5, P7, P8, P11); three of the participants were neutral (P3, P9, P10); and one of the participants agreed that it was difficult (P5). Some of the participants commented on the effectiveness of the documentation. Three stated that it was useful for helping them get started (P4, P6, P9). It was suggested by another participant that the learning difficulty depended on the user's creative coding experience:

> "For someone with creative coding experience, I picked it up easily. For an absolute newbie coder, I'd put it on a par with something like Processing." (P3)

Live coding: Most of the participants reported enjoying the live coding experience in Visor. Reasons for this included its similarity to Processing (P1, P2), utilisation of Ruby (P4, P5, P9), and the fast iteration time that was provided (P1, P6). Some of the participants also reported issues with the live coding experience in Visor. One participant struggled to improvise quickly, but put it down to their lack of experience with the IDE (P4), another stated that it was tedious to have to execute different code tabs individually (P5), and one other mentioned that they would be more productive if they could use their own editor (P8).

State management: The state management interface was reported to be effective for a number of reasons. These reasons included the ability to set ranges on values (P2), slider interactions (P5, P11), visual confirmation for debugging (P3, P5, P6), and being able to change variable values without inspecting the code (P2, P7, P9). One participant also discussed specifically how the feature provided high level control of a coded sketch:

> "It was also useful for compartmentalizing the sketch into different key pieces I can control once they were setup." (P6)

Two of the participants also brought up a usability issue with the state management interface, reporting that it became cluttered as their programs got larger and there were no options for organisation (P1, P2). One participant with extensive live coding experience did not use the feature due to how it required them to take their hands off the keyboard and reach for the mouse (P4), something they were not accustomed to doing when live coding. This insight emphasises the issue of context switching that was also identified in the live performances described in Section 3. The participant went on to discuss how they would have used the state management interface, MIDI controller, and tap tempo more if they were conducting pre-composed performances.

Layers: Visor's ability to organise code into layers proved to be one of the most popular features amongst all of the participants. Reasons for the feature's popularity included how they enabled switching between scenes (P2, P7), combining sketches (P1, P2), organising content (P4), provided a way to develop or test a piece of code in isolation (P4, P9), provided more visual variety from less code (P8), and were fun to experiment with (P3, P6).

Supporting comments from two of the participants were:

> "Very useful. Especially if you're used to Photoshop, the metaphor for composing layers like that makes a lot of intuitive sense." (P1)

> "Blending modes especially were fun to experiment with since it was easy to make many variations with just a few sketches and it was also great to have a completely new sketch to branch into once the starting sketch got too complicated." (P6)

Usability issues with layers were also identified by some of the participants. For example, P4 raised a concern about the lack of a keyboard shortcut to easily switch between code tabs. FFT: A number of participants claimed that they found the FFT effective in Visor. Reasons for this included its ease of use (P1, P3, P5, P6), visualisation of the frequency spectrum (P2, P8), enabling audio reactive visuals (P7, P9), or that it was something they were accustomed to (P4). Two participants requested a need for more control over the FFT including the smoothing (P2), volume level (P5), and adding support for multiple channels (P5). One participant couldn't configure an audio input (P10).

Tap tempo: The tap tempo was reported by participants to be effective for a number of reasons including its visual design (P2), ease of use (P3), ability to sync visuals with a tempo (P3, P5, P6, P9), and it's availability in situations where neither MIDI or an audio input are available (P8). One participant's experience with the feature was:

> "Tap tempo and the 'beat' features was [sic] useful to quickly get something visually interesting that was synced

> to the music. As someone with limited musical experience, being able to tap to set the tempo was much more intuitive than typing a number." (P6)

One participant was unsure about the tap tempo and suggested adding options to manually tune the BPM and offset without the need to tap (P1). One participant also reported that they did not use the feature because they didn't initially recognise its use in the visual arts (P4).

MIDI: Visor's support for MIDI was a feature that could not be evaluated effectively as part of the feedback survey. Only one participant managed to use a controller successfully (P4). Another participant tried to use a MIDI controller but could not use it effectively due to an issue with Visor or the controller (P2). The remaining 9 participants did not use the MIDI feature. This is likely due to a lack of access to a controller, highlighting a disadvantage of this type of remote study. A better approach to testing Visor's MIDI support would be to conduct an in-person user study where a controller is provided for the participants to use.

Ease of use: One prominent theme in the results was the ease of use of some of Visor's features. For example, the audio input for the FFT was reported to be easily configured through the GUI, and the built-in methods to access the data were straightforward to use. In general, this aspect can be summed up from the following comments:

> "It was very quick to launch Visor and just start making something interesting and dynamic, and not have to worry about setting up different libraries." (P6)

> "I liked that [sic] many options for getting dynamic input (FFT, tap tempo, setting up buttons and sliders) and that it was straightforward to access them within the sketch. I found these features to be better to explore/control the sketch's style than typing up variables." (P6)

The last comment also touches on the participant's enjoyment of the high level functionality that Visor's features provided. This reiterates one of the motivations for CJing in that the high level control provided by features of VJing software can improve the usability of live coding.

Conclusions

Visor is a new environment for live visual performance that was developed to demonstrate CJing, a new hybrid performance practice that combines aspects of live coding and VJing, drawing on the strengths of both practices while simultaneously removing limitations identified in each practice. To determine the effectiveness of Visor and whether CJing can effectively combine live coding and VJing, we have reflected on Visor's use in live performances, as well as conducted an evaluation of feedback gathered from creative coders, live coders, and VJs who experimented with the environment.

The use of Visor in performances has demonstrated Visor's ability to effectively produce visuals in a live context, at least in combination with the first author's own performance skills. Two approaches to performance were described: live coding content from scratch, and performing with prepared content. These approaches demonstrate how Visor can be used effectively for conducting aspects of live coding, VJing, and both together in the same performance, demonstrating CJing. The feedback gathered from Visor users suggests that each of Visor's core features were effective for their intended purpose except for the support for MIDI, which could not be evaluated to the extent of the other features. A number of usability issues and suggested improvements were also identified. Overall, the feedback was highly supportive of Visor and participants generally enjoyed using the environment. The evaluation of Visor has demonstrated Visor's effective support for aspects of both live coding and VJing, improving the usability of live coding through high level user interfaces and providing fine-grained control of content while VJing. This showcases Visor's demonstration of CJing, and in turn, how CJing can be used to effectively combine live coding and VJing.

Two prominent issues with CJing were also identified that need to

be considered in future work. The first was the need for careful design of the relationship between code and user interfaces when designing CJing environments. The second was the issue of context switching that highlighted the need for performers to split their focus between live coding, interacting with GUIs, and using hardware controllers.

A number of opportunities for future work have also been identified. These include further development of the Visor software to improve usability and to offer more features to enhance the environment's performance capabilities. Ideally, we would then conduct a more comprehensive evaluation of the environment through a controlled user study. We also hope to explore how collaboration with live coders, DJs, VJs, and CJs can play a role in CJing practice. In addition, we hope to explore how CJing can be applied in other live coding environments and in particular, within the context of music.

Acknowledgements

To Victoria University of Wellington for the Victoria Masters by Thesis Scholarship. To the Faculty of Science for the Faculty Strategic Research Grant. To the participants who completed the feedback survey. To those who collaborated with or provided the first author with the opportunity to perform on various occasions.

References

Aaron, S. Sonic Pi performance in education, technology and art. (2016). In: International Journal of Performance Arts and Digital Media 12, 2, 171–178.

Abras, C., Maloney-krichmar, D., and Preece, J. User-centered design. (2004). In: Bainbridge, W. Encyclopedia of Human-Computer Interaction. Thousand Oaks: Sage Publications 37, 4, 445–456.

Bergström, I., and Lotto, B. (2008). Mother: Making the performance of real-time computer graphics accessible to non-programmers. In: re) Actor3: The Third International Conference on Digital Live Art Proceedings. pp. 11–12.

Blackwell, A. F. (2014). Palimpsest. In: J. Vis. Lang. Comput. 25, 5, 545– 571.

Blackwell, A. F, and Aaron, S. (2015). Craft practices of Live Coding Language Design. In: Proceedings of the First International Conference on Live Coding.

Collins, N., Mclean, A., Rohrhuber, J., and Ward, A. (2003). Live coding in laptop performance. In: Organised sound 8, 3, 321–330.

Faulkner, M., and D-fuse. (2006). VJ: Audio-Visual Art and VJ Culture: Includes DVD. Laurence King Publishing.

Gaver, W. (2012). What should we expect from research through design? In: Proceedings of the SIGCHI conference on human factors in computing systems, ACM. pp. 937-946.

Purvis, J., Anslow, C., and Noble, J. (2019). CJing Practice: Combining Live Coding and VJing. In: Proceedings of the International Conference on Live Coding (ICLC).

Reas, C., and Fry, B. (2006). Processing: programming for the media arts. In: AI & SOCIETY 20, 4, 526–538.

Roberts, C., Wright, M., Kuchera-morin, J., and Höllerer, T. (2014). Gibber: Abstractions for creative multimedia programming. In: Proceedings of the International Conference on Multimedia, ACM. pp. 67–76.

Salazar, S. (2017). Searching for Gesture and Embodiment in Live Coding. In: Proceedings of the International Conference on Live Coding (ICLC).

Smith, N. C. (2016). Praxis LIVE - hybrid visual IDE for (live) creative coding. In: Proceedings of the International Conference on Live Coding (ICLC).

Tanimoto, S. L. (2013). A Perspective on the Evolution of Live Programming. In: Proceedings of the International Workshop on Live Programming. pp. 31–34.

Toka, M., Ince, C., and Baytas, M. A. (2018). Siren: Interface for pattern languages. In: Proceedings of the International Conference on New Interfaces for Musical Expression (NIME). pp. 381–386.

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

 ICLC 2020, February 5-7, 2020, University of Limerick ,Limerick, Ireland

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.

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.

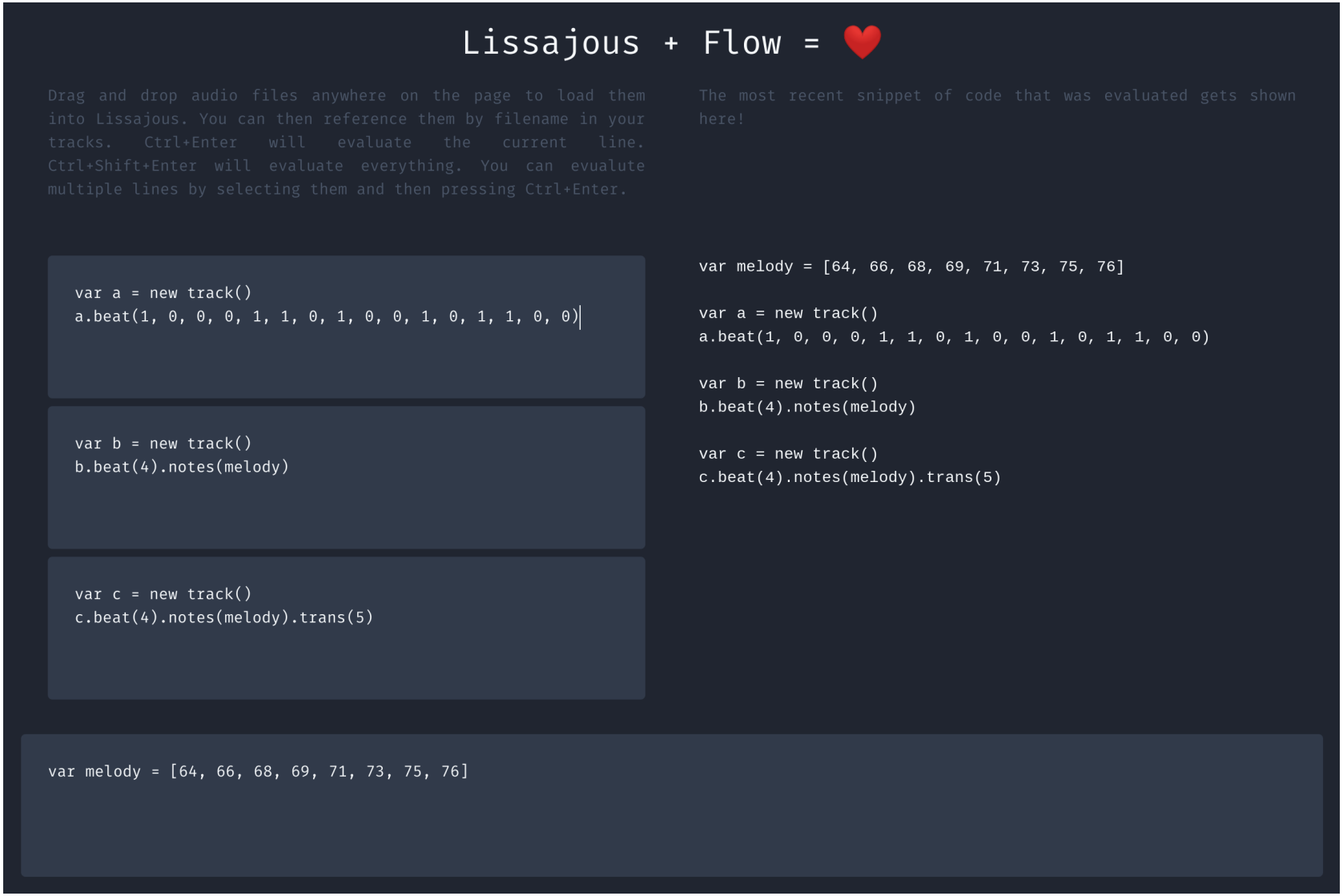


Figure 1: The Flow-Lissajous environment. (Left) The code editor. Here, as with many other live coding environments, lines of code are entered and evaluated. (Right) The last snippet of code that was evaluated. This section of the environment cannot be edited, instead always showing the most recent piece 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.

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.

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);
```

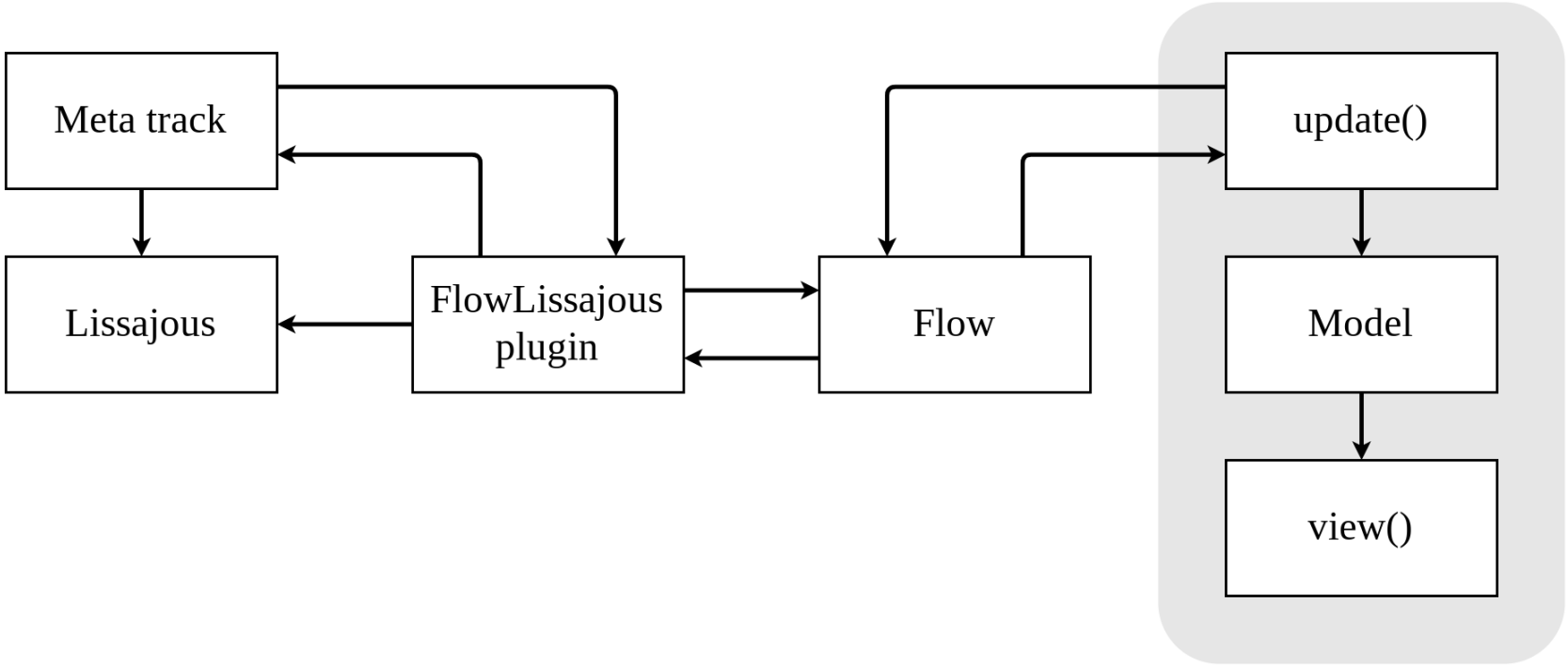


Figure 2: The overall architecture of the FlowLissajous environment. The grey shaded area represents the side-effect free portion of a Flow program. This is important to guarantee the same sequence of actions produces the same sequence of rendered views. Upon receiving an action, Flow calls the user-defined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

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

INSTALLATIONS

Anatomies of Intelligence

Joana Chicau
hello@jonathanreus.com

Jonathan Reus
hello@jonathanreus.com

Artist Statement

Anatomies of Intelligence is an artistic research initiative seeking tomake connections between the formats and collections of anatomicalknowledge and investigations into the "anatomy" of computational learningand prediction processes, datasets and machine learning models. (Read moreabout our method: `https://anatomiesofintelligence.github.io/about.html`)

The Catalogue / Explorations in Collection and PreparationWe began the project by building an online repository which gatherster-minologies and techniques for a critical examination of the "anatomy" oflearning and prediction processes, data corpora and models of ma-chinelearning algorithms. This catalog is a living research docu-ment into thehistory and present of anatomical science, as well as "anatomies" ofartificial intelligence algorithms and data stewardship practices. Thecatalogue is also itself a reflection on the process by which collectionsof anatomical specimens were constructed in the for-mative decades ofEuropean/Western anatomical science. By collect-ing our research exemplarsas a demonstrable archive, we attempt to explore the process of preparationand collection that reifies a collec-tion of information into a body ofknowledge, a collection that can be consumed and an ontology created. The catalogue itself has a built-in javascript API and a machine learningfunction which can be 'activated' by writing code on the web console. Thefunctional com-mands of the script utilizes terminologies from theanatomical theater which guides us through the process of clustering thedifferent catalog entries.

The Theatre / Staging Bodies of AI as Spectacle

The catalogue interface is also our platform to explore, throughperfor-mance, how such a collection and an artisanal algorithmic toolkit can-confront the idealized bodies of artificial intelligence — its fixedrep-resentational structures and opaque learning processes. Inspired by the layout of the 18th century anatomical theatre, theperformance is

divided into various interfaces:1 the catalogue entries where the clustering can be visualised, projectedon to a large table;2 the web console from the catalogue where the script runs;3 interfaces displaying code used to sonify and visualize the processesof analysis.For this proposal we intend to produce a video installation displaying thethree above mentioned interfaces which guide the audience through ananatomised live coding process.

Bio

Joana Chicau [PT/NL] is a graphic designer, coder, researcher — with a background in dance. Her trans-disciplinary project interweaves web programming languages and environments with choreographic practices. In herpractice she researches the intersection of the body with the constructed, designed, programmed environment, aiming at in widening the ways in which digital sciences is presented and made accessible to the public. She has been actively participating and organizing events with performances involving multi-location collaborative coding, algorithmic improvisation, open discussions on gender equality and activism. [web: `joanachicau.com`]

Jonathan Reus [US/NL] is a musician and artist who explores expanded forms of music-making and improvisational performance through technological artefacts. His practice is cross-disciplinary and research-based, involvingopen and iterative processes of collaboration with practitioners from across the arts, sciences and humanities. His work tries to confront and challenge the representational capacities of mathematical-logistical systems, algorithms, and infrastructure through a practice of invasive intuition and trust in the diversity of lived experiences. [web: `jonathanreus.com`]

Figure 1

Metastasis II

Feli Cabrera López
Estéticas Expandidas
rizza.av@gmail.com

Artist Statement

Metastasis II is a project born in the Electronic Arts (MFA)program of the Tres de Febrero National University, where I investigate the arts making processes and aesthetics from the limits of life. The work is an experiment which uses decomposing organic materia (meat), the states of the materia are sensed and interpreted and used for show/generate life signs (images, sounds,behaviors, displacements). The decomposing is evidence and appearance of another biosphere; the microorganisms that interact with the organic materia to allow its decomposing, generates visible color changes and environment ph transformations, which are analyzed and interpreted. After the image analysis of the decomposition, the results are communicated as data to the generative sound/visual organisms, so to influence their behavior and to generate an a live virtual audiovisual environment.This is a work that propose an alternative way to approach decomposing materia, in this case, as a life motor. And which present questions between the concepts of health, nature, life and artifice.The meat and its bacterial processes are in this work a kind of performers, they control the parameters of the growth on the artificial (audiovisual) life. The artificial life in this work is,in summary, structures composed of displaced lines in time and space,where the image residue (the growth) is enabler of the complex forms.This project looks to generate a series of works, textures, codes, asa virtual metaphor about biology in relation to its displacement,time and space; based in the idea of that every complex form, alive,is union and residue of successive accumulation of simple forms. And that the sensing work of this decomposing materia, is a enabler of ways/channels for the propagation of something that previously was territory only of the tangible-objects. In this way, the materia as data is propagated to another territories of the virtual world andallows to born the organisms in new organs of the digitality. This propagation allows, as metaphor, the fecundation of an object/work made from the limits of life and death in a techno-corporal territory, taking other life-beings as performers in a kind of non-human or post-human live-coding installation

Bio

Feli Cabrera López aka Efe Ce Ele `http://efeceele.com.ar/` Trans, Artist, Designer and Researcher; focused on human body, abstraction,perception, physics and semiotics. MFA on Technology and Aesthetics of Electronic Arts (UNTREF, Argentina). CoFounder Director and Curator of Estéticas Expandidas International Festival(Colombia).She research and produce in different formats such as video art, sound art,music, audiovisual concert, generative art, interactive art, installation, performance and electronic art. Currently investigating the possible connections between body, gender, identity, art and new media. Her work has been exhibited in different countries such as Argentina, Colombia, Ecuador, Germany, Greece, Italy and South Korea

Figure 1

Memorias

Jessica A. Rodríguez
McMaster University
rodrij28@mcmaster.ca

Artist Statement

MEMORIAS is a project installation with a personal approach to one's body and its memory system. This memory system is built through both individual and collective resemblances that are attached to one's specific historicity. MEMORIAS draws from two places: My work in Andamio, an artistic collective which experiments with electronic literature and audiovisual practices; and, my own participation in de development of Estuary, a hosting platform for live coding languages and networked music. The production of MEMORIAS involved three steps: 1) writing six autobiographical short-stories, 2) designing and programming six mini-languages using as conceptual base the autobiographical stories, and 3) conceptualizing her installation art piece as a mix of the six mini-languages into one space/time.

The artistic decision of having six stories relates to how I experience the world through Hearing, Writing, Watching, Reading, Seeing and Listening. Another artistic decision was to use Saramago's idea of seeing commas and points as musical pauses related to the voice, rather than as merely grammatical punctuation/separations. For the piece, this implies using the pause as a pivot for traveling or expanding to other ideas/images/resemblances, instead of finishing or cutting them. This pause also allows the sentences of the stories to have a certain degree of modularity to change the arrangement without losing meaning. These stories are written in two languages: Spanish and English. Yet, they are not translations from each other but versions that explore both similar and different ideas/resemblances.

Once the stories were finished the idea was to produce a third version: six mini-languages that could mix natural languages but at the same time triggered visuals or sound outputs through an ensemble environment. The conceptual part of the project involved the decisions regarding sintaxis, as well as visual and sound specification in each case, and the application part involved parsing existing languages in Estuary. Three of the mini-languages are parsing Tidal Cycles but aesthetically they're doing different things. "Write" and "See" uses voice samples (in English and Spanish), the recordings were cleaned

and divided into sentences as well as archived by the name of each text. In both mini-languages the most important function is the one that allows to call randomly the samples archive in each folder, as well as transformation that result in producing variations of each story. "Watch" uses instrumental samples (cello and paetzold) that are also divided and archive in different folders. "See", and "Listen" parse Punctual, the first uses the Audio part, and the second the visual Part. For these mini-languages, the parameters of the sound waves, filters and other forms are fixed within the code. Finally, "Hear" is parsing CineCer0, and it is limited to playing, moving and sizing pre-recorded and edited video samples.

Finally, the installation part uses Estuary's collaborative mode and the personalization of views in ensembles. The idea is that all mini-languages/stories coexist in the same space. Coming back to the idea of how we experienced the world, our remembrances, our bodies and our memory, the installation is called "How do I move through space/time". Using Estuary's platform allows the constant change of the sintaxis of each mini-language where being in space is not a possibility. These changes are made remotely during the exhibition time while Estuary is running in a server, always accessing the correct ensamble to be played, show, and/or changed. This allows more flexibility that one though using the random function within the code.

Credits: Voice in English Vic Wojciechowska (Canada); Voice in Spanish and text edition Rolando Rodríguez (Mexico); Cello Iracema de Andrade (Brazil-Mexico); Paetzold Alejandro Brianza (Argentina); Technical support David Ogborn (Canada) y Luis N. Del Angel (Mexico-Canada).

This piece was funded by the Program for Creations and Artistic Development in Michoacán (PECDAM'18). The project was also created as part of the Networked Imagination Laboratory [NIL] at McMaster University and through the personal collective project Andamio.in

Figure 1

Bio

Mexico. Visual composer and researcher. Ph.D. student in Communication, New Media and Cultural Studies at McMaster University. She has a Master of Arts with a wide work on musical/visual with

the new technologies research. She is the co-founder of andamio, a collaboration platform where she developed performances, educational projects, and research papers. Her practical work has focused in the collaboration with composers to produce audiovisual projects. https://andamio.in https://rggtrn.github.io/index.html

WORKSHOPS

Live Code Language Design with Sema

Francisco Bernardo
EMUTE, Lab Department of Music, University of Sussex
f.bernardo@sussex.ac.uk

Chris Kiefer
EMUTE, Lab Department of Music, University of Sussex
c.kiefer@sussex.ac.uk

Thor Magnusson
EMUTE, Lab Department of Music, University of Sussex
t.magnusson@sussex.ac.uk

There are compelling opportunities for empowering the live coding community with new artistic processes and outcomes that may arise from the integration of real-time interactive signal processing and machine learning technologies, in a scalable and accessible environment such as the web browser. This workshop is part of the MIMIC research project, which explores how machine learning and machine listening can be designed in a user-friendly way and provide for live coding, rapid prototyping and fast development cycles of musical applications. Participants will get familiar with our live coding language design environment and language design techniques, preparing them to work in the creation of their own live coding languages. Participants will learn how to express sound and music concepts using signal processing expressions. They will experiment with selected machine learning models and machine listening JavaScript classes and explore some of their direct benefits, including beat detection, pattern detection and generation.

Figure 1

Participants will gain understanding about workflows in our system that will allow then to develop their own live coding mini-language using language design techniques. For instance, they will learn how to create, inspect and change language specifications to generate language parsers. Participants will also explore practical techniques for performing with the live coding mini-languages they have constructed or customised, including live coding with interactive machine learning and pre-trained generative models. Additionally, the workshop will include a group discussion about participants' experience learning and using our system.

Introduction to music-making in Extempore

Ben Swift
Australian National University
ben.swift@anu.edu.au

Figure 1

This workshop will provide a hands-on introduction to making music in the live coding programming environment Extempore (`https://github.com/digego/extempore`). The workshop will cover:

- the fundamentals of making sound (starting from "hello sine")
- an overview of Extempore's built-in DSP libraries (including live coding with UGens)
- introduction to note-level live coding with Extempore's built-in synths & samplers

No Extempore or previous live coding experience necessary (although more experienced practitioners are welcome as well). Just bring a laptop with macOS, Windows or Ubuntu installed and we'll start from scratch. All the required software is freely-available, and a binary release of Extempore will be provided (so there's no need to build anything from scratch).

Hex Osc Shift Mod

Steven Yi

Rochester Institute of Technology

syyigm@rit.edu

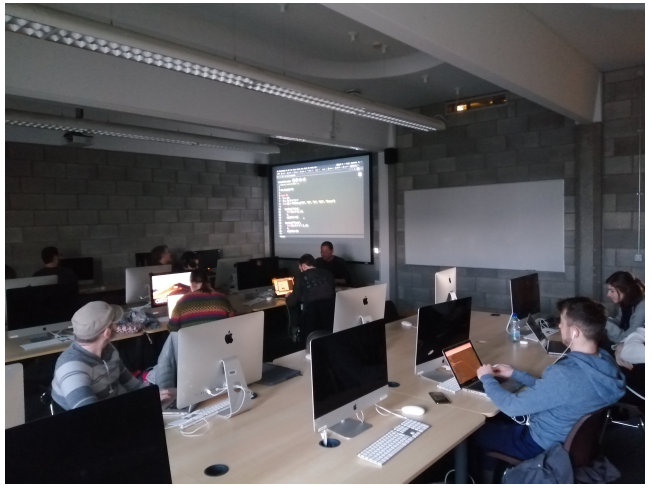


Figure 1

This workshop will explore creative uses of hexadecimal notation ("Hexadecimal Beats"), event-time oscillators, bitshifts, and modulus processing for live coding performance. We will explore the nature of each technique, see how it integrates with other techniques, and learn how to use it to generate note events as well as how to work with them to shape a performance. The workshop will use live.csound.com, a web environment for live coding with Csound and the csound-live-code library, to explore each technique. A laptop is required for participants to follow along and practice. No other installation of software is necessary except to have a WebAudio-compatible browser (e.g., Chrome, Firefox). While we will be using live.csound.com to explore the techniques in the context of music, the techniques and practical experience should be easily transferred to other live coding systems.

Approaches to Working in a Flexible Network, Reimagining the Ensemble.

Amble Skuse
University of Plymouth
amble.skuse@plymouth.ac.uk

Shelly Knotts
Durham University
knotts.shelly@gmail.com

In this workshop the participants will be introduced to ways of thinking about diversity as a practice applicable to all. We will cover the basics of Disability theory through a presentation, looking at barriers which are often presented to disabled people as a result of a "normal" approach to music making and performances. We will then explore how we can reimagine music making and performance as a collaborative negotiated experience in order to include all participants. This second aspect of the workshop will involve taking on disparate identities, writing a "needs assessment" for this new identity and working in small groups to create a music making and performance approach which suits those involved. This practice hopes to highlight the necessity of ground up decision making rather than "bolt on" diversity fixes.

This workshop is tailored to a live coding environment. Networked ensembles provide a key opportunity for disabled people to collaborate which "locational" ensembles do not. In our development of the laptop orchestra, and collaborating live online, we open the door to disabled and home based people in a way which other ways of music making do not. In this context it is useful for all people involved in networked ensembles to have an understanding of how we can create an environment which is open to and aware of disabled music makers.

Participants get into small groups, each participant takes an "identity" card.

Person 1:
You are a person with a chronic health condition. You spend 6 days out of 7 at your home.
You can only work every ow and again and you don't know when this will be, some days you are incredibly tired and cannot process information.
You have very little disposable income.
Your computer is 8 years old.
You have a wifi connection which is relatively stable.
You can concentrate on learning new software for about 20 minutes before needing a substantial rest.
You can communicate via telephone and online messaging.

Person 2:
You are a disabled person with upper limb difference.
You have two hands but limited finger movement which makes typing slow and laborious.
You can leave your home several days per week.
You prefer to use a tablet for software as swiping visual displays is easier than using text based software.
Your internet connection is stable.
You find learning new software easy.
You have a part time job which means you can only get online to play or perform in the evenings UK time.

Person 3:

You are a paraplegic person with no use of your limbs.
You use a head mouse to control your computer. This means moving a tube with your tongue to control the curser, and blowing into the tube as a click function. You have no double click or right click function on your head mouse.
You have a carer who comes in the morning to help you and you can ask this person to write e mails for you.
You don't have any software or coding experience and require quite a lot of support to understand how to use new programmes.
The easiest way for you to communicate is verbally.

Person 4:
You have Cerebral Palsy. Your body movements are often spasmodic and your speech is affected considerably.
You can control a mouse and type slowly.
Your mental and cognitive functions are not affected although sometimes you get very tired.
You are excited about the possibility of working with a new music making platform.
You have a carer who comes in the mornings to help with your daily care and so you are not available during this time.
The easiest mode of communication is text.

In groups of 4 the participants explore each other's identities through a series of questions.

1. When is the best time for you to meet and play?
2. What is the best way for you to communicate with the group?
3. What is the best way for you to use a computer?
4. What problems do you experience using software and computers?
5. What would make working with this software simpler for you?
6. What is putting you off working in this way?

Then as a group they look at various live coding platforms to see which platform would suit the ensemble.

As a group they must negotiate a working plan which allows everyone to work in a way which suits them. The groups then report back to each other around different issues they found with various live coding platforms, and how they overcame them as a group. What groups dynamics arose and how did they overcome clashes of need in the group?

With this new knowledge and experience, the groups suggest ways in which live coding software and ensembles could be adapted or reimagined to be more accessible to disabled users. Background to Workshop

SOCIAL MODEL of Disability

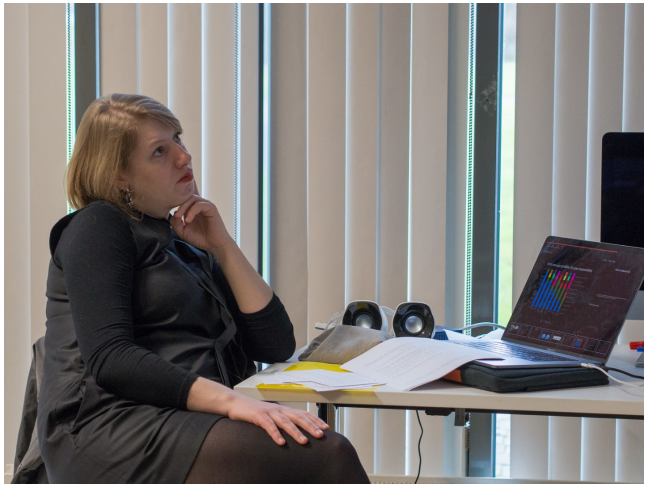


Figure 1

The Social Model of Disability describes how disabled people are not disabled by their bodies, but by a society which creates environ-

ments in which we cannot function. This is the process of society and structures actively disabling people. As systems and built environments are often created by able bodied people for non disabled people, the needs of disabled people are rarely taken into account, and thus the structures created include barriers which prevent disabled people from gaining access. For example, the person who uses a wheelchair is disabled by the decision not to include lifts and ramps to a building. It is the design of the environment which disables them. Following this theory, a person with a fatigue condition is disabled from taking part in a project because of the long days scheduled. The design of the project has disabled the person.

Often when we think of access and inclusion we think "what do we have to add on to our usual arrangements to make it accessible? Is it a ramp? Is it captioning?" While these thoughts are helpful in terms of adjusting existing situations, they negatively deflect the need for inclusion back onto the disabled person. We think we have to make adjustments because of the disabled person, rather than because of the initial design failing to consider everyone who may use it. This philosophically brings us back to centring the disabled person as the problem, an obstacle to be overcome, an adjustment which gets in the way of the "normal" flow of things. This recentring of the disabled person as other and problematic, reinforces negative views of disability from mainstream society. However, if we build society with everyone in mind from the outset, we will not need to make adjustments, and we will not draw attention to those who have different needs to the majority.

The extension of the Social Model approach is the "nothing about us without us" concept which holds that those for whom a service, system or environment is designed, must have a contributing say as to its design. By incorporating diverse voices in the design of something, we can more reasonably design something which fits those it will serve. Something which is designed for a group of individuals with complex needs therefore, will need to be designed with each individual in mind by consulting those people and asking what they need to flourish.

Open, high and low: writing classes in SuperCollider

Hernani Villaseñor Ramírez
Graduate Music Program, UNAM
hernani.vr@gmail.com

SuperCollider is a well known software in the context of sound live coding. It is free, open source and have a huge community around it. From artistic perspective, usually source code during a live coding performance is written in a high level. A lot of black boxes operates to facilitate the live coder to express their ideas with a high degree of abstraction. During a live coding performance we write code based on messages and methods for objects, classes and plug-ins to generate sound, in this moment we don't have to know how these technological devices are made, just how they function. But live coding, as well as improvisation, doesn't come from nowhere, live coders explore the software and its possibilities before the performance. So, What if we explore not just the function of the software but some mechanisms of its structure?, What if we open some black boxes to understand better the tool and our practice? Furthermore, What if we design some of these black boxes? This workshop aims to explore classes and objects in SuperCollider written in sclang to show participants how to start to write a class. This is not an expert level workshop, but for those who want to be more confident with technological development or curious of the SuperCollider software structure. The aim is to write a simple class and put it in the program extensions.

The idea behind this workshop is to begin to understand the software structure in relation to an artistic practice, and formulate some questions regarding the relationship between computational art and technological development in the context of the open.

Figure 1

Open, high and low: writing classes in SuperCollider

David Ogborn
McMaster University
ogbornd@mcmaster.ca

Jamie Beverley
University of Toronto
jamie_beverley@hotmail.com

Alejandro Franco
McMaster University
francoba@mcmaster.ca

Alex MacLean
McMaster University
maclea11@mcmaster.ca

Luis N. del Angel
McMaster University
navarrol@mcmaster.ca

Jessica Rodriguez
McMaster University
rodrij28@mcmaster.ca

In this workshop, we will demonstrate and collectively explore numerous different approaches to collaborative audio-visual live coding, taking particular advantage of the multiple languages targeting visual output within the Estuary collaborative live coding platform. As this platform unites heterogenous languages within a single, zero-installation, web-based platform, it is particularly well-suited to collaborative work between musicians and visualists (as well as people who are both musicians and visualists!). Synchronization, data sharing, and ensemble transparency (the ability to see what others are working on) are default starting conditions with Estuary, rather than "special features/modes". The development of languages targeting visual outputs has been a particular focus of recent work within Estuary. The Punctual language allows WebGL fragment shaders to be created from economical Haskell- like notations that can also, simultaneously, be translated into Web

Figure 1

Audio API graphs and sound output. The CineCer0 (pronounced "sin–ay– ser-oh") language allows video files to be projected temporally and geometrically, targeting similar functionality to that of CineVivo, again with an economical Haskell-like notation. An even newer, third graphics language within Estuary is projected to be available for this workshop, modelled on The Force and like that environment making it possible to write fragment shaders in "raw" GLSL.

All of the above languages can be used simultaneously within the

networked collaborative interfaces provided by Estuary, and as such, in close connection with other languages that target sound output more exclusively. This is strongly supportive of experiments with different ways of relating sound to visuals and vice versa, and consequently doing such experiments collectively will be a core activity within the workshop.

Participants are strongly encouraged to bring their own laptops with either the Chrome or Chromium browser installed, as well as some type of headphones for the headphone jam. No previous experience with live coding, Estuary, or any of the languages used here is assumed, so beginners are welcome and will likely take away techniques they can put to use immediately. At the same time, the workshop's focus on an emerging area of interest in live coding research (better understanding the possibilities of combined audio-visual work) will hopefully provide something to challenge any live coding "veterans" in the group.

`https://intramuros.mcmaster.ca` (Estuary, use Chrome/Chromium)
`https://dktr0.github.io/Punctual` (Punctual, use Chrome/Chromium)

Giving live to autonomous agents

Ulysses Popple
ulysses.popple@gmail.com

Autonomous agents created using machine learning are a growing presence in our world, so we should use them for art! Participants of this workshop will livecode virtual worlds and populate them with intelligent agents.

The workshop will start with a brief overview of tools and terminology. Training (outside the scope of this workshop) is accomplished using Unity's ml-agents and simplified with **??**PersonalityQuarks. The primary livecode interface is Arcadia with a few helpful wrappers from baisless fabric.

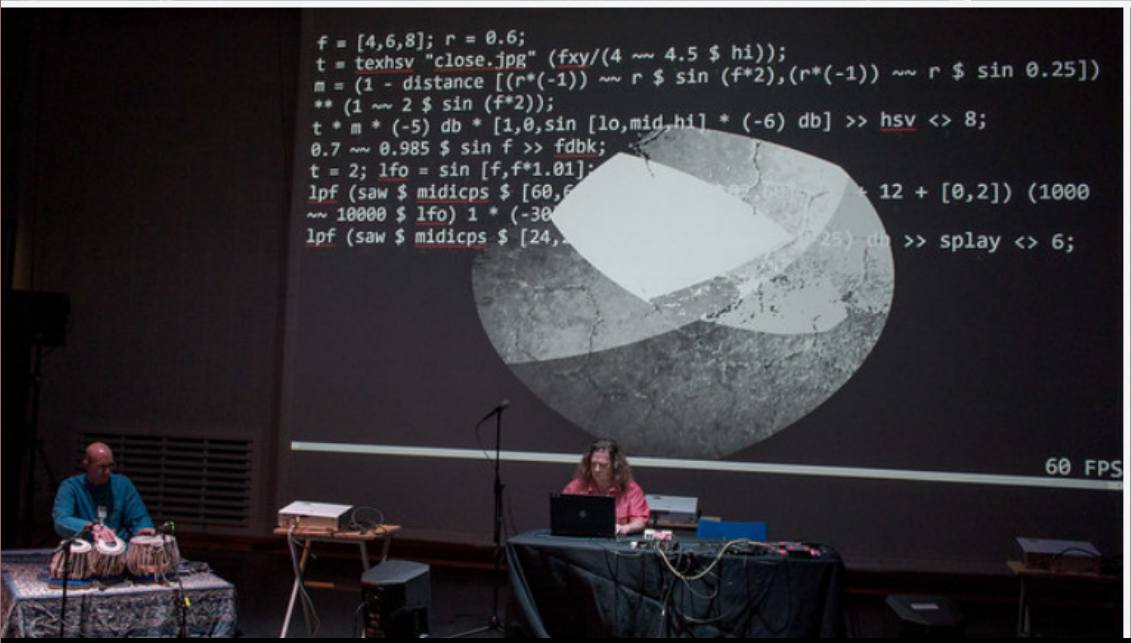
The first half hour - hour will be the overview and setting up tools. Participants should download Unity and a github repository with all the important parts set up (link to be provided later), but there's always going to be complications.

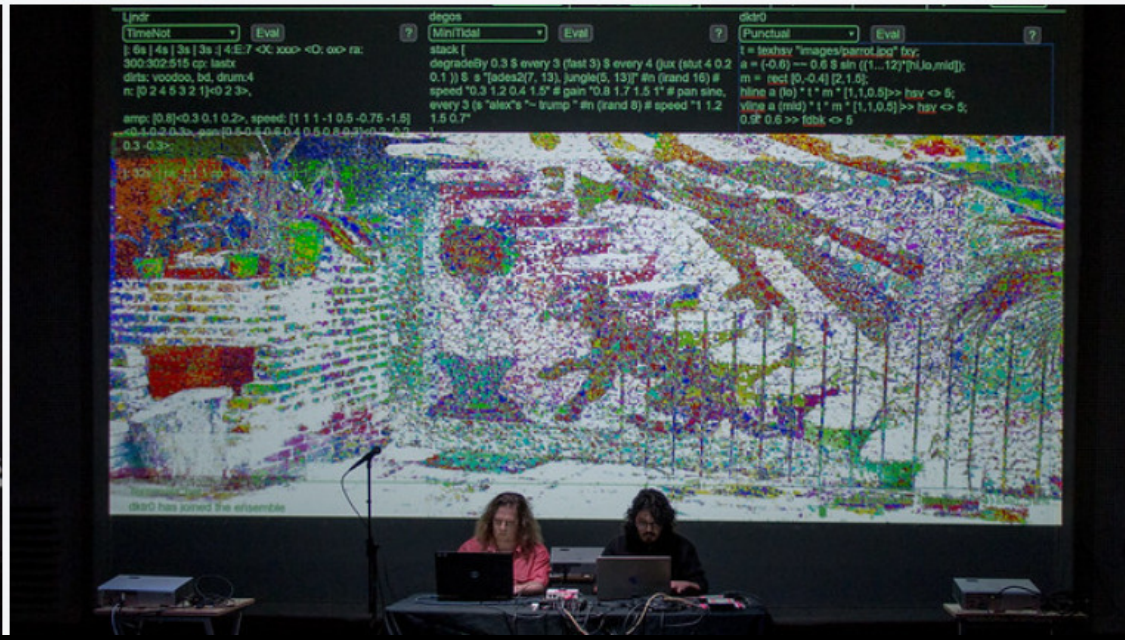
The remainder of the time will be spent on exploring the environment. Participants will start by creating an agent prefab and then spawning it. Then they'll experiment with multiple agents, then modifying the environment for agents to interact together, and finally (if there's time) customizing their environment with colors and lights. By the end of the workshop, participants should have an understanding of how to use autonomous agents in a live performance, and have the tools to train their own agents and create their own environments.

This will be a moderately advanced workshop, and participants will preferably have some familiarity with lisp and 3D environments.

Figure 1

LUNCH CONCERTS





Wednesday 5th - 13:00-14:00

GLSL Anywhere

Shawn Lawson
Rensselaer Polytechnic Institute

Bio

Shawn Lawson is an artist researching the computational sublime. He performs under the pseudonym Obi-Wan Codenobi where he live-codes real-time computer graphics with his open source software. He has performed or exhibited in England, Scotland, Spain, Denmark, Russia, Italy, Korea, Portugal, Brazil, Turkey, Malaysia, Iran, Canada, and the USA. He received grants from NYSCA and the Experimental Television Center, and he has been in residence at CultureHub and Signal Culture. Lawson studied at CMU and ÉNSBA. He received his MFA in Art and Technology Studies from SAIC. He is a Professor in and Head of the Department of Art at RPI.

Thursday 6th - 13:00-14:00

Very Long Cat

David Ogborn* & Shawn Mativetsky
*McMaster University & McGill University

very long cat is a tabla and live coding duo that rehearses and performs over the Internet (between Hamilton and Montréal). For ICLC2020 we propose a co-located improvisation highlighting the capacity of the Punctual browser-based live coding language to produce generative visuals, incorporating the analysis of audio signals from the tabla as well as transformed photographic textures.

The Punctual browser-based live coding language has been developed asa "core" language of the Estuary collaborative live coding platform, with an emphasis on economical notations to describe how successive definitions of the same computational entity relate to each other intime (ie. "transitions"). The language allows graphs of oscillators, filters, mathematical operations, etc to be directed to both audioand visual outputs (as Web Audio API graphs, and WebGL fragmentshaders, respectively). In earlier performances, very long cat made heavy use of the JITlib affordances of SuperCollider as well as the jacktrip software for network audio – our proposed performance for ICLC 2020 is a chance to showcase the evolution of that practice, now based heavily on zero-installation web technologies (such as Punctualand Estuary).

Bio

Dynamic performer Shawn Mativetsky is considered one of Canada's leading ambassadors of the tabla, and is a pioneer in bridging theworlds of Western and Indian classical music. Called an "exceptional soloist" by critic Réjean Beaucage, Whole Note's Andrew Timar adds that "as a leading disciple of the renowned Sharda Sahai, he has serious street cred." Shawn Mativetsky is highly sought-after as both performer and educator, and is active in the promotion of the tabla and North Indian classical music through lectures, workshops, and performances across Canada and internationally. Based in Montreal, Shawn teaches tabla and percussion at McGill University. His first solo recording, Payton MacDonald: Works for Tabla, was released in 2007, and Cycles, his recording of Canadian compositions for tabla,was released in the fall of 2011. His most recent release, Rivers, is a solo tabla album rooted in the rich traditions of the Benares style of tabla playing. `https://www.shawnmativetsky.com/`

David Ogborn / dktr0: `http://www.dktr0.net` hacker, composer, artistprogrammer, live coding and guitar performer; lead developer

of numerous software projects used in network music and live coding, including EspGrid, extramuros, Punctual, and Estuary; a founding member of the Cybernetic Orchestra; director of the Networked Imagination Laboratory http://nil.mcmaster.ca, and the Centre for Networked Media and Performance (CNMAP) at McMaster University; Associate Professor in McMaster'sDepartment of Communication Studies and Multimedia, teaching in the undergraduate Multimedia program, the MA in Communication and New Media, and the PhD in Communication, New Media, and Cultural Studies https://csmm.mcmaster.ca.

Friday 7th - 13:00-14:00

SonoTexto: live coding sound environment

Hernani Villaseñor
UNAM

For ICLC 2020 I propose a live coding performance in which I will record some fragments of sound ambience of the concert hall or spaceand live code them using sclang, for that purpose I will use the SuperCollider Class SonoTexto. SonoTexto is a technological object and a performance which come froman old question of computational art regarding the relationship between artistic practice and technological development. In this case, the structure of the SuperCollider software is explored to understand some technological devices that conform it as classes in order to open some black boxes that represent the high level of abstraction during a live coding performance. That is to say, to explore the source code of SuperCollider to understand how a class works and how is written. Writing classes in SuperCollider permits not just understand the technological device itself but to modify themodes of production of own artistic practice.The idea behind SonoTexto is to record small fragments of the sound environment during a live coding performance to control these sounds with code, that is to say, SonoTexto, as performance, is a metaphor of live coding the sound ambience. In this regard, some seconds of sound are recorded in buffers with the built-in microphone of the computer or one connected to a sound card. Then, the buffers are reproduced, modified and processed through source code organized in patterns, routines or proxy space. SonoTexto, as a SuperCollider class, has three methods:.boot call a script that have the Buffers and the SynthDefs to record and reproduce sound; .rec records the sound into the Buffers and .writew rites the content of the Buffers to the hard disk if the coder wants to keep the recorded sound. SonoTexto is available here: https://github.com/hvillase/sonotexto

Bio

Hernani Villaseñor is a Mexican musician interested in sound, code and improvisation. He is currently a PhD student at the Music Graduate Program of the National Autonomous University of Mexico. His current research is about the implications of writing source code indifferent levels and layers to produce music and sound. He is also interested in artistic research and the relation of art and technology. As a musician he performs and improvises computer music with source code as interface in a range of styles from techno toexperimental sound. He has collaborated with different artists in thefield of cinema, experimental video, photography and installation. He has performed in many venues and participated in diverse conferences in countries of America and Europe. As an organizer he have co-organized three international symposiums dedicated to music and code called /*vivo*/ and many concerts for the Centro Multimedia CENART in Mexico. www.hernanivillasenor.com

Maths That Wiggle Air

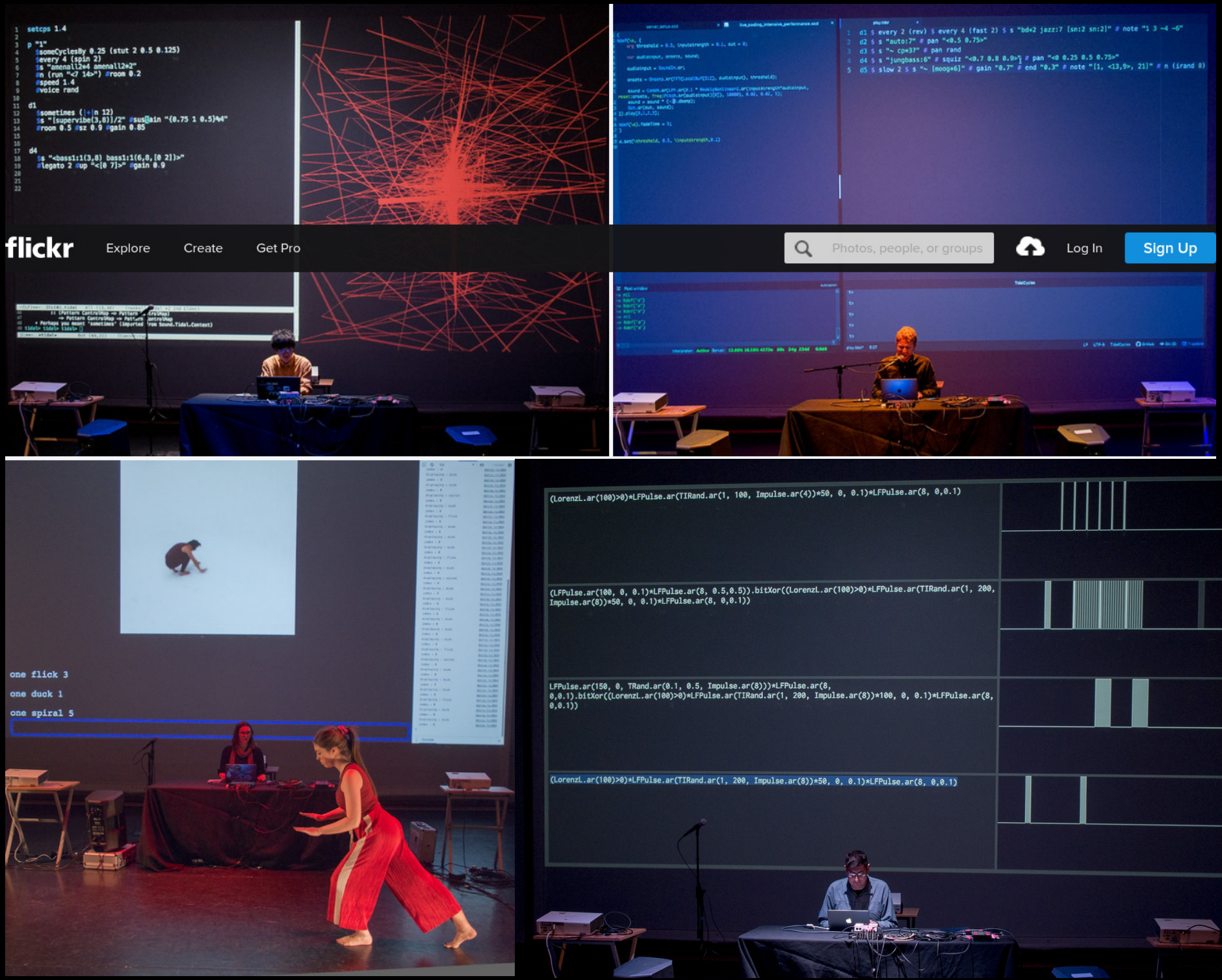
Dimitris Kyriakoudis (w1n5t0n)
Independent, lnfinite Monkeys

Music is made of sound that changes over time. Sound is made of waves that wiggle air particles over time. Those waves in turn can be made of other waves, that change over time too. Mathematical functions, in particular some of the simplest algebra taught in schools, turn out to be particularly well-suited for looping, twisting, and bending the flow of time to create and compose those waves.This performance uses such mathematics to improvise a choreography of air particles. The instrument of that improvisation is Time Lines, a functional reactive language embedded in Haskell, used to control the parameters of both hardware and software modular synthesizers using numerical functions of time. Time Lines itself doesn't synthesize the sound, it just repeatedly asks the question: 'If the musical time right now is t, in seconds,then what value should each parameter of every sound process have?'. By building upon multiple different layers of abstraction, the performer constructs a series of equations that, in parallel, shape the linear flow of time into each answer for every moment in time.The resulting waves are then indexed through at a 1:1 rate and sent to various modular synthesis and effects processes, which live in SuperCollider and hardware analog circuits.

Bio

Dimitris Kyriakoudis, occasionally known as w1n5t0n, is a researcher,musician, and computational artist among the lnfinite Monkeys. He studied music at a young age by reading and writing black squiggles on pieces of white paper, playing them using an array of even more discrete, but equally black and white, on-off switches. That turned out to be a bit too limiting, so now he can be found looking up obscure error messages and making funny noises using functional programming and mathematics he can barely understand. He develops TimeLines, a live coding modular synthesizer and sequencer based on mathematics and FRP, and is obsessed with keyboard technique and code editing ergonomics.

EVENING CONCERT
I



Wednesday 5th of February 2020
19:30
Theater 1 Irish World Academy UL

Very Long Cat

David Ogborn* & Shawn Mativetsky
*McMaster University & McGill University

very long cat is a tabla and live coding duo that rehearses and performs over the Internet (between Hamilton and Montréal). For ICLC2020 we propose a co-located improvisation highlighting the capacity of the Punctual browser-based live coding language to produce generative visuals, incorporating the analysis of audio signals from the tabla as well as transformed photographic textures.

The Punctual browser-based live coding language has been developed asa "core" language of the Estuary collaborative live coding platform, with an emphasis on economical notations to describe how successive definitions of the same computational entity relate to each other intime (ie. "transitions"). The language allows graphs of oscillators, filters, mathematical operations, etc to be directed to both audioand visual outputs (as Web Audio API graphs, and WebGL fragmentshaders, respectively). In earlier performances, very long cat made heavy use of the JITlib affordances of SuperCollider as well as the jacktrip software for network audio – our proposed performance for ICLC 2020 is a chance to showcase the evolution of that practice, now based heavily on zero-installation web technologies (such as Punctualand Estuary).

Bio

Dynamic performer Shawn Mativetsky is considered one of Canada's leading ambassadors of the tabla, and is a pioneer in bridging theworlds of Western and Indian classical music. Called an "exceptional soloist" by critic Réjean Beaucage, Whole Note's Andrew Timar adds that "as a leading disciple of the renowned Sharda Sahai, he has serious street cred." Shawn Mativetsky is highly sought-after as both performer and educator, and is active in the promotion of the tabla and North Indian classical music through lectures, workshops, and performances across Canada and internationally. Based in Montreal, Shawn teaches tabla and percussion at McGill University. His first solo recording, Payton MacDonald: Works for Tabla, was released in 2007, and Cycles, his recording of Canadian compositions for tabla,was released in the fall of 2011. His most recent release, Rivers, is a solo tabla album rooted in the rich traditions of the Benares style of tabla playing. `https://www.shawnmativetsky.com/`

David Ogborn / dktr0: `http://www.dktr0.net` hacker, composer, artistprogrammer, live coding and guitar performer; lead developer of numerous software projects used in network music and live coding, including EspGrid, extramuros, Punctual, and Estuary; a founding member of the Cybernetic Orchestra; director of the Networked Imagination Laboratory `http://nil.mcmaster.ca`, and the Centre for Networked Media and Performance (CNMAP) at McMaster University; Associate Professor in McMaster'sDepartment of Communication Studies and Multimedia, teaching in the undergraduate Multimedia program, the MA in Communication and New Media, and the PhD in Communication, New Media, and Cultural Studies `https://csmm.mcmaster.ca`.

Three Ravens

Francisco Bernardo, Chris Kiefer & Thor Magnusson
Emute Lab, School of Music, University of Sussex

We present a live coding musical performance that illustrates live coding systems created with Sema1, our new Web-based live coding language design and performance system. Using three bespoke languages, Kiefer, Bernardo and Magnusson will collaborate on a co-located networked musical performance where the music is live coded in real-time. Each of the languages serves as an instrument in the ensemble of three. This performance brings together three connected live coding languages upported by Sema. The second iteration of Sema(to be presented at ICLC2020) enables users to create their own mini-language. We have created three distinct mini-languages that serve as instruments of a musical ensemble. We explore the extent to which these mini-languages enable expressive live coding performance. In a live coding fashion, the screens will be projected onto the wall enabling the audience to follow the performance as it is played on the pidgin languages.

Bio

Thor Magnussonis a worker in rhythm, frequencies and intensities. His research interests include musical improvisation, new technologies for musical expression, live coding, musical notation and digital scores, artificial intelligence and computational creativity, programming education, and the philosophy of technology. These topics have come together in the ixiQuarks, ixi lang, and the Threnoscope live coding systems he has developed.

Chris Kiefer is a computer-musician and musical instrument designer, specialising in musician-computer interaction, physical computing, and machine learning. He performs with custom-made instruments including malleable interfaces, touch screen software, interactive sculptures and a modified self-resonating cello. Chris is an experience live-coder, performing under the name 'Luuma'. He performs with Feedback Cell and Brain Dead Ensemble, and has released music with ChordPunch, Confront Recordings and Emute.

Francisco Bernardois a computer scientist, an interactive media artist,and a multi-instrumentalist. His research is focused on human-computer interaction approaches to toolkits that broaden and accelerate user innovation with interactive machine learning. In 2008, Francisco designed BEN, a language that augments Bluetooth naming for mobile interaction with intelligent and ubiquitous environments. Francisco has performed with different acts (e.g. FRANTICØ,:papercutz), and most recently, with his solo project MNISTREL

Voodoo Suite

Diego Villaseñor, Alejandro Franco Briones* & David Ogborn*
Independent, McMaster University*

We are proposing a performance using the libraries and platforms developed by the authors: Nanc-In-A-Can, TimeNot and FluentCan which emphasise poly-temporality and other forms of complex time relationships. The name of the performance is Voodoo Suite in reference to the long-scale music work by the Cuban-Mexican mambo composer and performer Damaso Pérez Prado. This piece is characterised by the counter position of various musical styles relevant for the author: mambo, African traditional music and Jazz. In this spirit, we intend to produce a multilinguistic livecoding act that not only oscillates between beat-oriented music and polytemporal experimental textures but also manages to express simultaneously similar or complementary compositional and algorithmic ideas with different live coding languages, notations and platforms. Moreover,the poly-temporal and rhythmic ideas expressed here will have a literary (words), visual and sonic output.

Bio

Alejandro Franco Briones My interests focus on experimenting with sonic phenomena in order to develop alternative ways to structure and perceive music/sound art.Two of the major focal point for my work as a composer/sound artists are the development of a rhythm oriented music, and the interactions and the multi lateral flow of in formation which occur between the composer,the programmer,thecode score, the instrumentalists, the performer and the audience. By the act of listening, I attempt to make evident the interplay that occurs between the audible and the muted qualities of musical reality.

Diego Villaseñor de Cortina I am a composer, improviser, philosopher and programmer.My work focuses on the exploration of the possibilities of modular composition, the interaction of heterogeneous entities and the creation, exploration and intervention of opensystems. Alongside Alejandro Franco Briones I am the coauthor of Nan-in-a-Can, a SuperCollider library for the creation of temporal canons. I am also a member of the free improvisation collective Ruido13, from Mexico City, with whom he has worked on concepts such as, the deconstruction of the musical instrument leading sound synthesis with acoustic instruments objects, of various forms of rhythmic texturing.

Weathery

Gerard Roma [0001]
University of Huddersfield

0001 is a project by Gerard Roma focusing on small-scale digital lo-fi experiments. In "Weathery", the project explores a more rigorous approach to one-bit synthesis. The sound generators are driven (where to?) by chaotic oscillators.

Bio

Performer: 0001

Livecoding with integrated visualization of code information and sound impression

Hiroki Matsui
Tokyo University of Technology

This performance presents an integrated audio visualization that combines code information and music abstraction contained in TidalCycles. The system called RIPPLE (Real-time Image Production Platform for Livecoding Environment) integrates two types of information. The numerical values that compose themusic quantitatively are obtained from the code. The impression of the sound is obtained in real-time by machine learning. In the machine learning module, I use the statistical model trained by myself to visualize the sound in a reflection of the coder's impressions and ideas. I try to generate visuals that are organically connected to music by sending the estimated results to openFrameworks.

Bio

Born in Shimane, Japan. He received a bachelor's degree in media science from Tokyo University of Technology (TUT) in 2019. He is currently studying live coding, creative coding, audio and speech science at the graduate school of Bionics, Computer and Media Sciences, TUT.

Bionic March: Live Coding with Voice and Machine Listening

Alex MacLean
McMaster University

This performance will combine live coding, voice, and machine listening. The live coding will be done in TidalCycles and accompanied by live vocals. A synth running in SuperCollider will be controlled by both the sound created by typing on the computer's keyboard and the live vocals with the use of onset and pitch detection, the machine listening component. This piece was first workshopped at the 2019 live coding intensive in the Networked Imagination Laboratory, and for a performance at ICLC 2020 will be augmented by working with samples from field recordings and generative visuals. The work makes use of Nick Collins' work on machine listening in SuperCollider (Nick Collins. 2015. "Live Coding and Machine Listening". Proceedings of the First International Conference on Live Coding).

Bio

Alex (aka Monalex as a live coding artist) is a musician, songwriter, audio/live sound engineer, and software developer from Northern Ontario. He has played in many bands over the years and told even more to turn their amps down on stage. His current band, Deepsea Challenger, is a progressive rock band based in Hamilton, ON that Alex sings, writes, and plays guitar for. He holds both a diploma in Music Industry Arts from Fanshawe College and a degree in Computer Science from Western University and has worked for several years as a DevOps and Cloud Engineer in the broadcast industry. More recently, as an MA candidate in Communication and New Media at McMaster University, he is investigating assistive applications of machine learning for the performing arts.

Terpsicode

Kate Sicchio, Marissa Forbes and Zeshan Wang
Virginia Commonwealth University

This performance is a duet for dancer and coder using Terpsicode, a developing programming language for live coding dance performance scores. It allows a visual score for a dancer to be created in real time as the work unfolds. The live coder is using the new programming language Terpsicode, designed specifically for this use. The code patterns images which are then projected into the performance space and interpreted by the dancer.

Bio

Dr. Kate Sicchio is a choreographer, media artist and performer-whose work explores the interface between choreography and technology with wearable technology, live coding, and videosystems. Her work has been shown internationally in many countries including the US, Germany, Australia, Belgium, Sweden, and the UK at venues such as PS122 (NYC), Banff New Media Institute (Canada), V&and A Digital Futures (London), and Artisan Gallery (Hong Kong). She is currently Assistant Professor of Dance and Media Technology at Virginia Commonwealth University.

Zeshan Wang is a motion and 3D artist interested in the relationship between how people represent themselves physically and digitally. They are currently a student at Virginia Commonwealth University majoring in Kinetic Imaging and Computer Science.

Cibo V2

Jeremy Stewart & Shawn Lawson
Rensselaer Polytechnic Institute

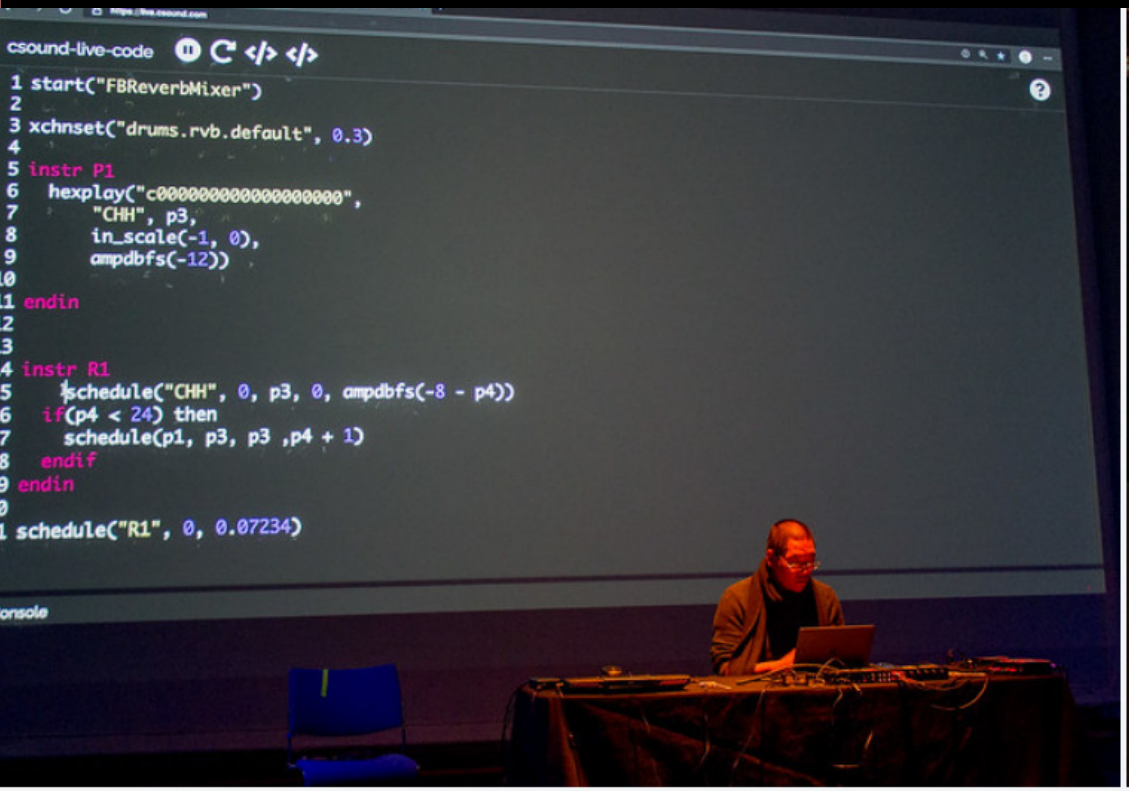
The Proposal is for Cibo V2, a machine learning agent, to perform solo. This version of the Cibo V2 agent is vastly different with multiple changes to the deep neural network structure, performers contributed learning data sets, additions to the visual representation of how the agent manipulates code. Updates to agent architecture are outlined in our technical paper submission.

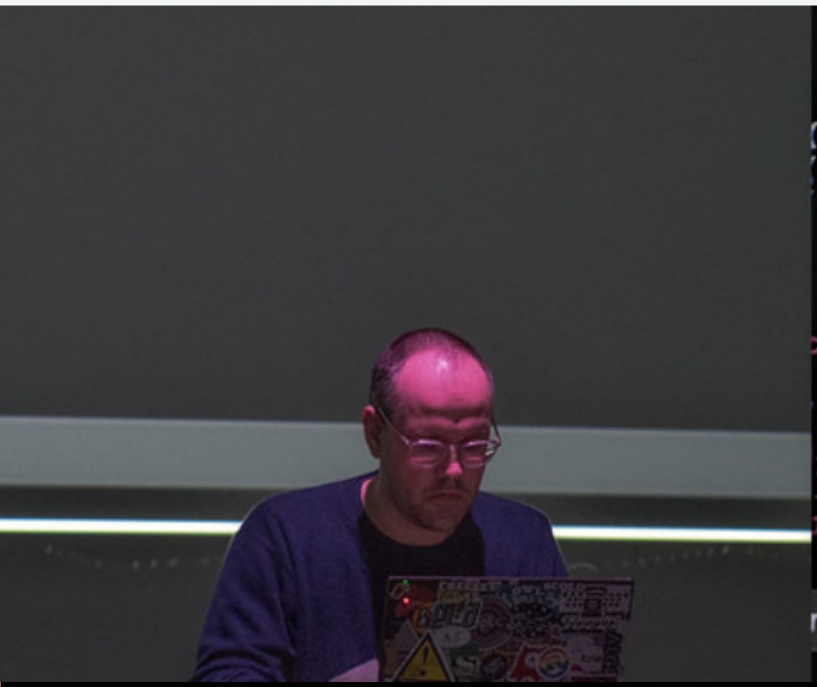
Bio

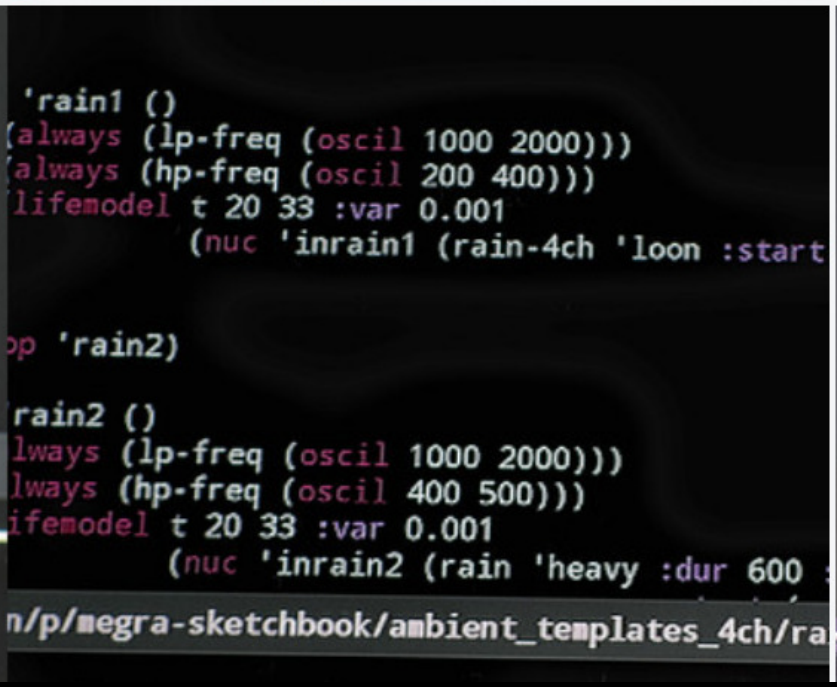
Shawn Lawson is an artist researching the computational sublime. He performs under the pseudonym Obi-Wan Codenobi.

Jeremy Stewart is a multimedia artist and performer researching the affective potential of distributed media systems through the creation of improvisational performances, artificial intelligence (A.I.) software, and wearable hardware. Cibo V2 is the second iteration of their machine learning agent that live-codes TidalCycles solo. The training set for this performance comes from blind elephants, kindohm, and bgold; Jeremy Stewart, Mike Hodnick, and Ben Gold respectively.

EVENING CONCERT II







Thursday 6th of February 2020
19:30
Theater 1 Irish World Academy UL

Perang Gagal: a Series of Inconclusive Battles

Dr J Simon van der Walt
Royal Conservatoire of Scotland
with Prof. Mel Mercier and students of the Irish World Academy of Music and Dance - University of Limerick

This performance juxtaposes livecoded 'gamelan' music created in SuperCollider with music played live on real gamelan instruments. The title refers to a traditional set-piece scene from the Javanese wayang kulit shadow-play repertoire. Taken literally, this scene represents a series of skirmishes between characters from opposing armies, none of whom emerge clearly victorious. Metaphorically it might be taken to represent a transition between youth, as represented by the first pathet nemscene, to the middle age of pathet sångå, while in contemporary performance it is an entertaining crowd pleaser that allows the dhalang puppeteer to show off his skill in manipulating the puppets.In this performance, the puppet battles will be imaginary, and there is no attempt to mimic the particular series of srepeg and sampak musical forms that accompany this scene. Our intention is rather to capture the lively and rambunctious atmosphere, while perhaps hinting at the challenges and tensions inherent in combing performance by live instrumentalists with livecoded music.

Bio

Dr J Simon van der Waltis Glasgow-based composer and performing artist. Over the course of his career has created a varied and original body of work, ranging from score-based composition to installation, sound art, performance, and devised musik theater. His chief current preoccupations are Indonesian gamelan music, livecoding, and reconstructing thecareer of his fictional alter ego Edward 'Teddy' Edwards, unsung hero of British light music electronica. He is Head of MMus Programmes at the Royal Conservatoire of Scotland.

Professor and Chair of Performing Arts at the Irish World Academy of Music and Dance, University of Limerick, since 2016, Professor Mel Mercier was formerly Associate Professor of Music and inaugural Head of the School of Music and Theatre at University College Cork. He is a traditional percussionist, composer and educator, with an international reputation as an innovative performer, rooted in Irish traditional music and committed to collaborating across music genres and art forms. A renowned, award-winning composer, he has composed the music for many critically acclaimed, award-winning theatre productions and installations that have been presented at theatres and venues in Ireland, UK, Europe and America. Awards include: Irish Times Theatre Award for Best Soundscape for the Gare St Lazare production of Beckett's How It Is –Part I(2018); Irish Times Theatre Award for Best Soundscape for the Corcadorca production of Caryl Churchill's Far Away(2017); the Gradam Cheoil Awardfor Collaboration on CONCERT with Colin Dunne and Sinead Rushe (TG4, 2018); the New York Festival Bronze Medal Awardfor his radio documentary, Peadar Mercier(RTÉ Doc on One, 2017); theNew York Drama Desk Awardand Tony Awardnomination for his sound score for Colm Tóibín's Testament of Mary(Broadway 2012); and a nomination for an Irish Times Theatre Award for the Abbey Theatre/Fibín production of Paul Mercier's Sétanta. In 2002, he was nominated for a Drama Desk Award for the Abbey Theatre/Broadway production of Medea

eCossystem

Char Stiles
Robotics Institute at Carnegie Mellon University

Danielle Rager
Center for the Neural Basis of Cognition

arsonist (Danielle Rager) and Char Stiles live code audio and visuals to create a simulated ecosystem using cellular automata. The flux of the ecosystem is reflective of the ability to dynamically alter rules when live coding. The simulated ecosystem's state feeds back to alter the music and the visuals, leading to the genesis and destruction of synthesized life forms and their sounds. Starting with a Shepard tone and a sunrise, the performance begins. It builds to a crescendo as this world driven by an alternation of Stephan Rafler's smooth life in GLSL. There is an appearance of Primordial Particle Systems, as well as reaction diffusion algorithms throughout the performance. The rules of these systems are conducted by the sounds from arsonist. There is a narrative of creation and deterioration into chaos. Inside of this structured ecosystem waves of natural and synthesized sounds disturb and govern the digital landscape.eCosystem explores how live coding can enable a musician and visualist to perform and improvise based upon a symbiotic relationship through the medium of simple rules with emergent behaviors.

Bio

Char Stiles is a researcher and digital artist. Using computational systems and algorithms she is producing pieces that spans disciplines such as video, dance, interactive installation, performance and online works. Right now she is at the Robotics Institute at Carnegie Mellon University as a research associate. She has received awards from the Carnegie Museum of Art, exhibited internationally and talked and gave workshops at Carnegie Mellon University, Massachusetts Institute of Technology and New York University. Her portfolio is CharStiles.com.arsonist is the solo musical project of Pittsburgh native, classically-trained violinist and producer

Danielle Rager. Collaging raucous electronics, manipulated vocals, and her mother-tongue of lush string arrangements, arsonist has kindled organic, texture-rich digital soundscapes in live performances across the US. The audio/visual set that arsonist recently performed at the Mattress Factory with digital artist Char Stiles incorporates intricate, flexible sequencing of beats and found sounds with the live coding environment Tidal Cycles, marrying her music with the algorithmic nature of her daytime work as a Department of Energy Graduate Fellow in Carnegie Mellon University's Program for Neural Computation. Rager is also a member of the improvisatory, electroacoustic duo Diaphony and the DJ duo The New Hip Tiki Scene, winners of Secret Thirteen's 2017 New Blood mix competition.

CTRL+Z

Zeshan Wang Kinetic
Virginia Commonwealth University

This performance is simply a simulated interaction between programmer and programme. Using TidalCycles, the program will compose as the want but a concurrently running Python script will be running and attempting to play its own composition. This can be seen as a more benign version of previouswork (CTRL-Z) where the script would be constantly closing the composition window to impede the programmer's process.

Bio

Zeshan Wang is a motion and 3D artist interested in the relationship between how people represent themselves physically and digitally. They are currently a student at Virginia Commonwealth University majoring in Kinetic Imaging and Computer Science.

Improvisation

Steven Yi
Rochester Institute of Technology

Experiments in hexadecimal beats, event-rate oscillators, bit shifts, and modulus processing.

Bio

Steven Yiisa composer, performer, programmer, and Tai Chi practitioner. He is a developer of Csound and author of various computer music programs including Blue, Pink, Score, and csound-live-code. He is an Assistant Professor at the Rochester Institute of Technology in the School of Interactive Games and Media.

INFERNO

Giovanni Muzio (Kesson)

The Amazon fires have captured the attention of the world, and for good reasons: the destruction of one of the world major carbon stores, considered as "the lungs of the world", could strike a devastating blow to the fight against the climate change, and to the home of indigenous communities. Amazon fires increased by 84% compared to the same period last year, according to the satellite images by the Brazilian National Institute for Space Research. Yet, cattle ranching, logging and the production of soybeans was not only the plague for the deforestation in the Amazon Rainforest, but it affects other parts of the world, like the Cerrado (one of the world's most biodiverse regions), which is 50% deforested, or the Rainforest in Indonesia. Still, is not everything: the biggest number of fires, in the month of August was in Russia, followed by the Democratic Republic of the Congo and Brazil, according to the satellite images of NASA and VIIRS.INFERNO is a dramatic journey through the blazes around the globe to raise awareness about forest fires, not only in the Amazon but, for many reasons, across the globe. An audiovisual performance coded live, fetching both archived and real time data of the fires around the world from NASA database and their Near Real Time API. The performance is coded in the Processing environment, in REPL/Hot Swap mode, it includes some customizations (GLSL shaders, data fetching algorithms...) and it communicates on the fly with Max/MSP and Ableton Live for the sounds and music.

Bio

Giovanni Muzio (kesson) is an artist and researcher, graduated in New Media Arts. In his work he defines time as an abstract dimension, where matter is created and evolves continuously in imaginary landscapes. He works mostly with algorithms, mathematical formulas, in the intersection of order and chaos. The unpredictability of randomness and fortuity is mixed with the order and aesthetics of maths, in a combination that leads to poetic hallucinations. His aesthetics is minimal nevertheless it is also inspired by the uncanny quality of complexity, synthesized in geometric shapes with few colours, lines of lights, real data converted in abstract geometries, inspired by the cyberpunk and Sci-Fi imaginary.

Autobiographical Improvisation on 4 Channels

Niklas Reppel

An autobiographical performance based on found sounds, some of which have accompanied me since my early childhood, some of which have been recorded during the recent years. Using granular sampling, the sounds are stretched and extended, the time information is altered if not removed, and the sounds turn into a potentially endless texture. The Mégra (`https://github.com/the-drunk-coder/megra`) system and its simple life modeling algorithm generate subtle, if not subliminal,variations. The performer is given time and space to listen, select and layer the sounds in an improvisational manner, melting down a lifetime into a continously evolving stream of sound. Interaction is calm and reduced.The multichannel setup creates additional acoustic space to listen to, distribute and get lost in the sound, adding spatial wideness to the impression of timelessness.

Bio

Niklas Reppel (*1983, Witten a.d. Ruhr, Germany) is a live coder and audio software developer currently based in Barcelona. Oftentimes he is more involved in making live coding tools than actually using them. Coming from a multifaceted musical background that features everything from jammy rock bands to contemporary chamber music ensembles, improvisation and eclecticism play a major role in his current live coding works. For more information, see:`https://parkellipsen.de`

00000

15 MINUTE REHEARSAL

Rafaele Andrade — Jonathan Reus

This concert is a last-minute collaboration between Rafaele Andrade and Jonathan Reus, combining instruments of their own making.Rafaele will perform on her KNURL, a hybrid/reprogrammable cello, and Jonathan will perform using the live catalog interface from the project Anatomies of Intelligence, created along with Joana Chicau.

Bio

Rafaele Andrade, who grew up in Curitiba, Brazil, is a creative musician working for sustainability and ethic. Her background involves composition, conducting, sonology. She plays cello and currently designs an electric cello with a live code mode integrated on the instrument. When she was only seventeen, Andrade was already assembling and producing an orchestra of Brazilian popular music, and by the time she was 22, she was curating a UNESCO project to promote a group of Latin American women composers in order to showcase their work by radio globally (Rádio Delas). This latest project is an extension of her previous work, but also a shift into exploring the potential of acoustic instruments to be enhanced through built-in electronic components, as well as the potential for music to be a shared endeavor between performers and audience members. She is at the moment responsible for Mant, a worldwide project of sustainable live sound art, a project that brings environmental practices on stage, which respect and include all the energy sources available at the environment. Currently, she lives in the Hague, The Netherlands, performs in Europe and Latin America, and hopes to reach even wider audiences through her projects.
Jonathan Reus [US/NL] is a musician, educator and researcher in the field of electronic music instruments, sonic interaction design and critical computing. His artistic work deals primarily with themes relating

to computing culture, software and the capacities of mathematical-logistical systems to capture and represent the world. Musical performance is, for him, a method and means of being material. Together with Sissel Marie Tonn, he is one half of Sensory Cartographies, an artistic project that speculates on new forms of mapping through wearable technologies and techniques of observation and augmented attention.

ALGORAVE

Friday 7th of February 2020
19:30
Flannery's Bar -Upper William Street - Limerick City

7.30 Alejandro Albornoz [co(n)de Zero], Christian Oyarzún [voodoochild], Juan AndrésJaramillo Silva [Noisk8], (Universidad Austral deChile) - Austral Online Streaming – A remote transmission

7.45 Christian Oyarzún Roa aka voodoochild/ (Universidad Austral deChile) - n3_M3510:

8.00 Celeste EstebanBetancur Gutiérrez (ITM - Medellín) , Raul BenitoRevollo Sierra (ITM – Medellín) - Deus Ex Machina by Machina

8.15 Flor de Fuego (indipendent artist), Rapo (indipendent artist) – Free Software Cumbia

8.30 Ben Swift (Australian National Giovanni Muzio (Kesson) - .hyperkosmoUNiversity) Help Wanted: Open Port Algorave Set Streaming

8.45 Gergely Bencsik , Tímea Fekete (...) - No Title

9.00 Chris Kiefer (University of Sussex, Emute Lab) - DJ FPGA

9.15 Alex McLean (Deutsches Museum) - Prototype

9.30 Antonio Roberts (independent), Maria Witek (University of Birmingham) - mxwx + hellocatfood

9.45 Alo Allik (independent), Elizabeth Wilson (C4DM, Queen MaryUniversity) - digital selves + alo

10.00 Charles Roberts (Worcester Polytechnic Institute) - Untitled

10.15 NSFW (independent) osmia/Carla Sophie Tapparo

10.45 Sarah Groff Hennigh-Palermo, Melody Loveless, Kate Sicchio (...) - Codie Coding Coders

10.30 Shelly Knotts (Durham University) - AlgoRIOTmic Grrrl!

11.00 Maxwell Neely-Cohen (Independent), Zach Krall (Parson School of Design U.S.) - No Title

11.15 Shawn Lawson (Rensselaer Polytechnic Institute), Ryan Ross Smith (Monash University) - Codenobi & Wookie

11.30 Aleksandr Igorevich Yakunichev, Violetta Postnova (...) - No Title

11.45 Hiroki Matsui (Tokyo University of Technology) - Livecoding with integrated visualization of code information and sound impression

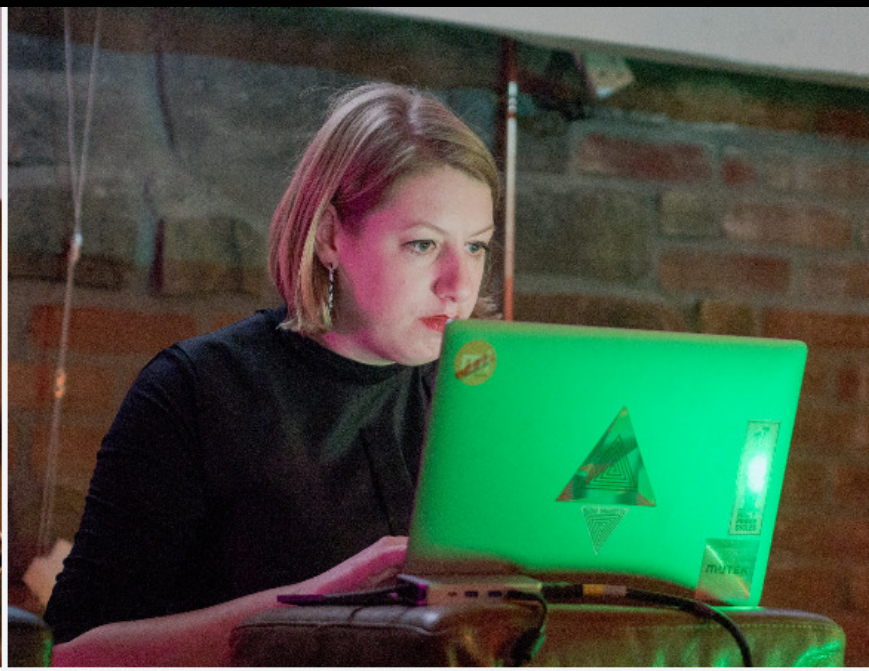
12.00 Timo Hoogland (HKU University of Arts Utrecht) - c26h21n3o3+o2

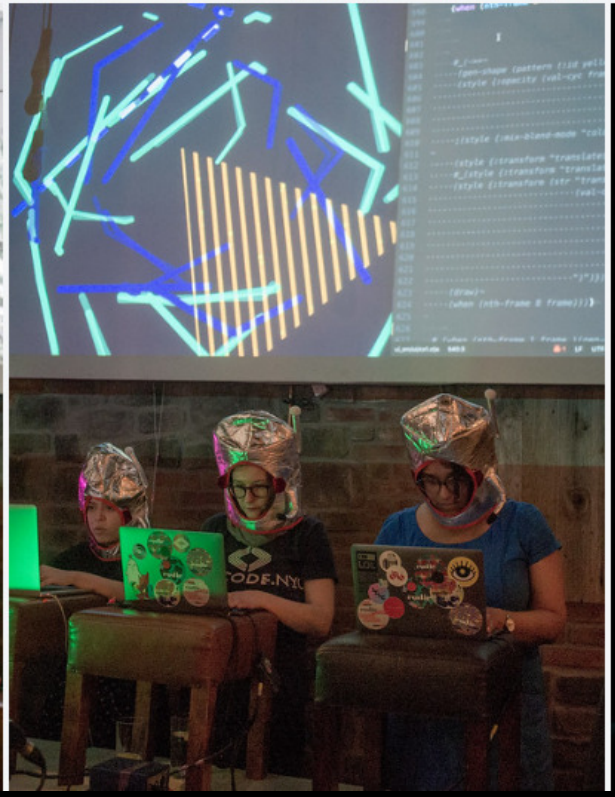
12.15 Niklas Reppel (...) - No Title Søren Peter (darch) VISOR - Blending Live Coding and VJing

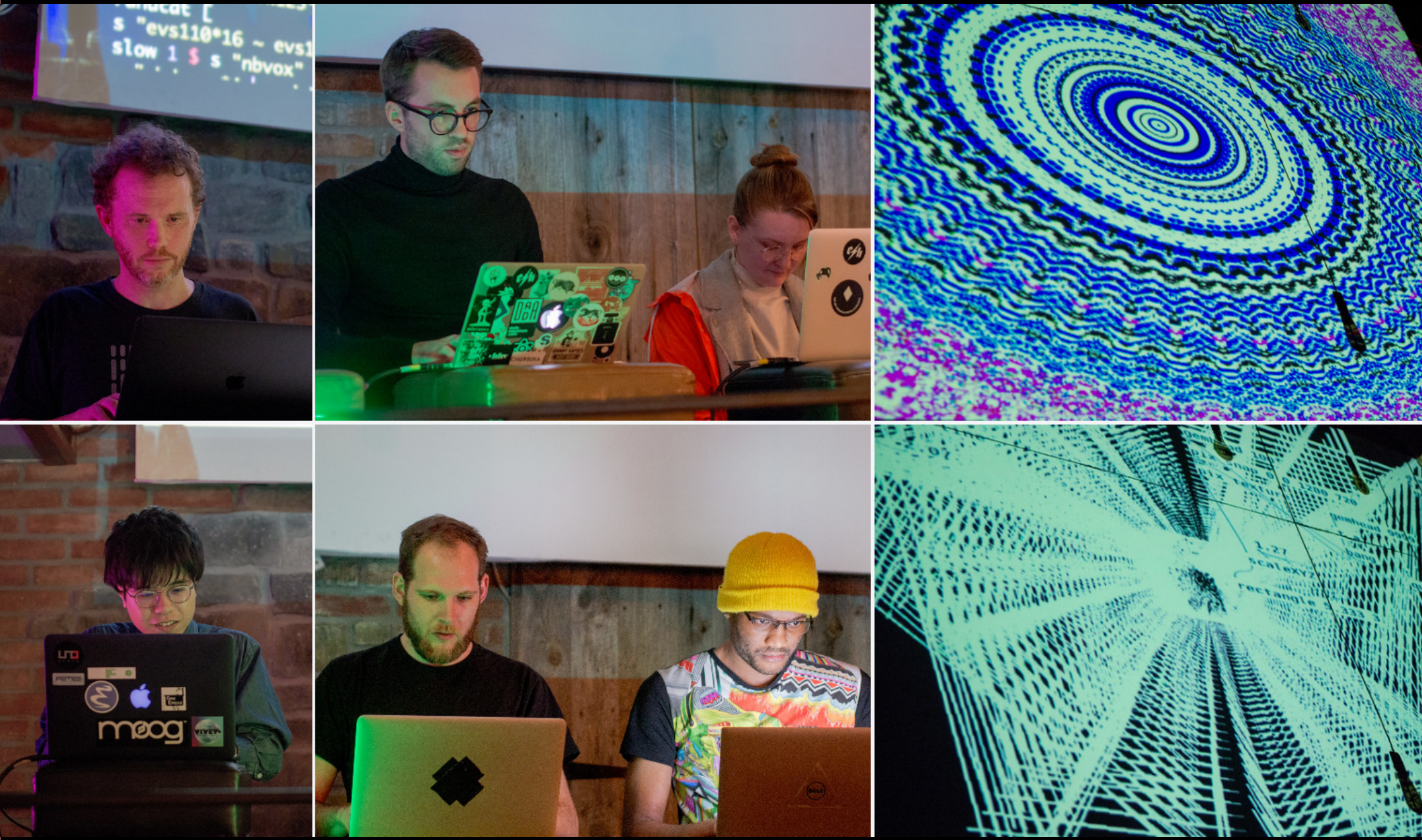
12.30 Ulysses Popple & Melody Loveless (...) - Baisless Fabric

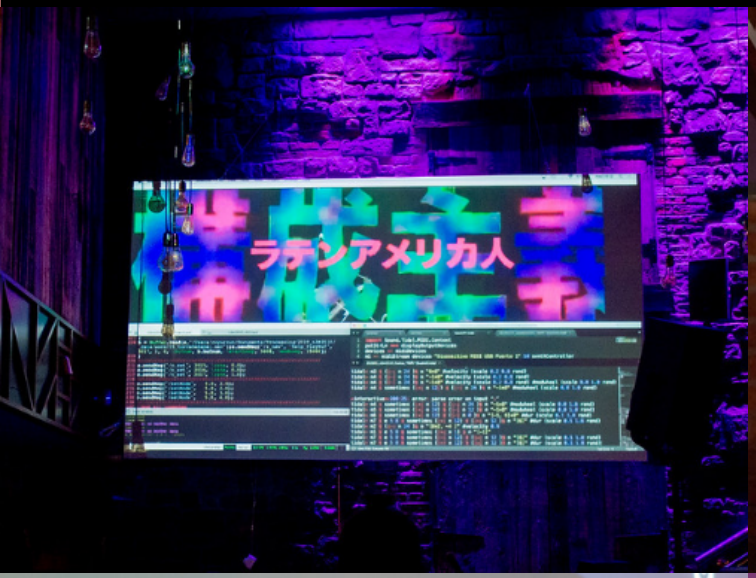
12.45 Alison Merri Amet (TOPLAP France) - Merristasis

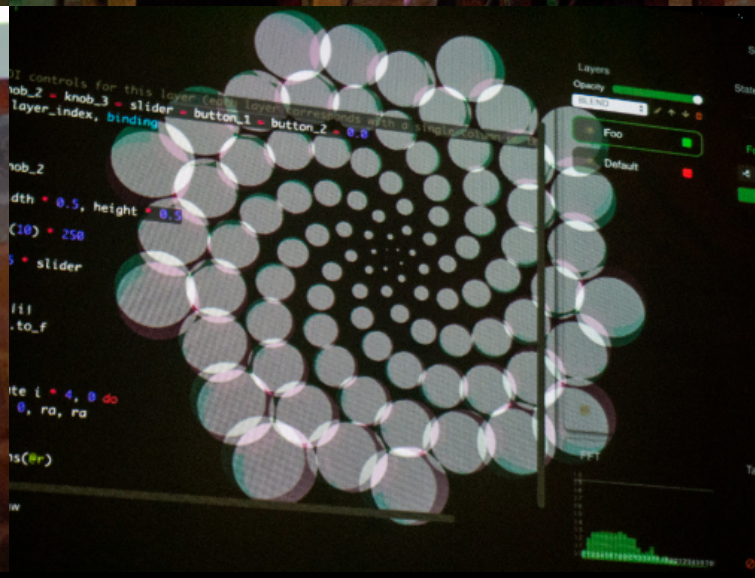
1.00 Jamie Beverley (University of Toronto,McMaster University) Jack Purvis (Victoria University of Wellington, New Zealand) - Dead Algorave












(c) 2020 Robin Parmar