Proceedings of the
First International Conference on Live Coding

Leeds UK, 13-15th July 2015

Proceedings of the First International Conference on Live Coding

Editors Alex McLean, Thor Magnusson, Kia Ng, Shelly Knotts, and Joanne Armitage

Published by ICSRiM, School of Music, University of Leeds, Leeds LS2 9JT

13th July 2015

ISBN 978 0 85316 340 4

Papers

Live Coding and Machine Listening

Nick Collins
Durham University, Department of Music
nick.collins@durham.ac.uk

ABSTRACT

Live coding control of machine listening processes, or more radically, machine listening control of live coding, provides an exciting area of crossover between two research frontiers in computer music. This article surveys the state of the art, and reports a number of experimental projects that point to potentially productive further directions.

1. Introduction

Live coding has established itself as a viable and productive method of computer music performance and prototyping, embracing the immedicacy of modern computer programming languages (Collins et al. 2003; Ward et al. 2004; Blackwell and Collins 2005; Brown and Sorensen 2009; Collins 2011; McLean 2011; Magnusson 2014). New avenues in interfacing for programming are being embraced, including tangible computing of unit generator graphs and musical sequences (see for example Mónica Rikić's *buildacode*), natural language processing (e.g., Craig Latta's Quoth) and computer vision (e.g., Nik Hanselmann's *bodyfuck* (2009) brainfuck interface). Performance ventures range from the current vogue for algoraves, showcasing generative and interactive creation of electronic dance music (Collins and McLean 2014) to contemporary dance and live arts.

Complementing existing work, this paper explores the interaction of live coding with machine listening, from listening functionality within existing performance systems towards the use of live audio analysis as programming interface itself. Although there are some existing projects which have utilised machine listening components, this area of computer music has received less attention in live coding, perhaps because of its technical challenges. Machine listening is not a solved problem, and whilst such elements as realtime monophonic pitch tracking, percussive onset detection and timbral feature detection are by now a staple part of computer music environments, more involved polyphonic pitch tracking, auditory scene analysis and live music information retrieval inspired work remains at the cutting edge (Casey et al. 2008; Klapuri and Davy 2006). It is implausible to claim any parity of machine algorithms to human listening at the present time, though many interesting artificial listening resources have been developed.

The paper proceeds after some further review of the confluence of live coding and machine listening within live code performance, to experiments of the author pointing to further possibilities for interaction.

2. Machine listening as resource within a live coding performance system

Musical live coding performance systems have predominantly explored deterministic and probabilistic sequencing of samples and synthesized sound events, driven from an interpreted or jit-compiled textual or graphical computer music language. Audio input, and particularly live analysis of audio input, is much rarer than direct synthesis.

Nonetheless, the work of Dan Stowell on beat boxing analysis (Stowell 2010) and associated MCLD live coding and beat boxing performances is a good example of the co-occurence of machine listening and live coding (Stowell and McLean 2013). Matthew Yee-King has also embraced dynamic live code control of listening systems, notably in improvised collaboration with Finn Peters (Yee-King 2011) with Markovian and evolutionary systems reconfigured on-the-fly to track a saxophonist-flutist. The feedback system between live coder and dancer set up by McLean and Sicchio (2014) utilises both computer vision (tracking the dancer) and machine listening (onset detection on the audio output), perturbing graphical placement of code elements within McLean's wonderfully idiosyncatic Texture language to affect choreographic instruction and audio synthesis code respectively. Such feedback loops of audio and video analysis were anticipated by the now defunct audiovisual collaboration klipp av, who live coded and live remapped audiovisual algorithms simultaneously in SuperCollider and Max/MSP (Collins and Olofsson 2006).

2.1. Machine listening as controller alongside live coding

Machine listening unit generators are available to the live coder, and highly beneficial in creating more involved synthesis patches, as well as feature adaptive processing of live audio input. Examples are provided here within the SuperCollider audio programming language, much used for live coding and well suited to live analysis and synthesis. For instance, an abstract feedback loop can easily be constructed where an audio construct's output is analyzed, the analysis result used to feedback into control of the audio, and parameters of this graph (or the complete graph) made available for live coding manipulation:

```
(
a = {arg feedbackamount = 0.5;

    var feedback, sound, sound2, freq;

    feedback = LocalIn.kr(2);

    //source sound, feedback into first input (frequency)
    sound = Pulse.ar(
                ((1.0-feedbackamount)* (LFNoise0.ar(2).range(10,1000)))
                    +
                (feedbackamount*feedback[0]),
                feedback[1]
            );

    freq = Pitch.kr(sound)[0]; //pitch detection analysis of output sound frequency

    LocalOut.kr([freq,(freq.cpsmidi%12)/12.0]); //send feedback around the loop

    sound

}.play
)

a.set(\feedbackamount, 0.98) //change later
```

Triggering sound processes from analysis of a driving sound process, through the use of onset detection, provides a typical use case (derived rhythms can be productive, for instance via onset detection interpretation of a source sound). The very typing of a live coder can spawn new events, or combinations of voice and typing drive the musical flow:

```
(
a = {arg threshold = 0.5, inputstrength = 0.1;

    var audioinput, onsets, sound;

    audioinput = SoundIn.ar;

    //run onset detector on audioinput via FFT analysis
    onsets = Onsets.kr(FFT(LocalBuf(512),audioinput),threshold);

    //nonlinear oscillator is retriggered by percussive onsets
    sound = WeaklyNonlinear2.ar(inputstrength*audioinput,reset:onsets, freq: Pitch.kr(audioinput)[0] );

    sound

}.play
)

a.set(\threshold, 0.1, \inputstrength,0.01)
```

```
a.set(\threshold, 0.8, \inputstrength,0.7)
```

It is possible to experiment with a much larger number of interesting processes than onset and pitch detection, whether timbral descriptors, or more elaborate sound analysis and resynthesis such as utilising a source separation algorithm to extract the tonal and percussive parts of a signal.

2.2. The Algoravethmic remix system

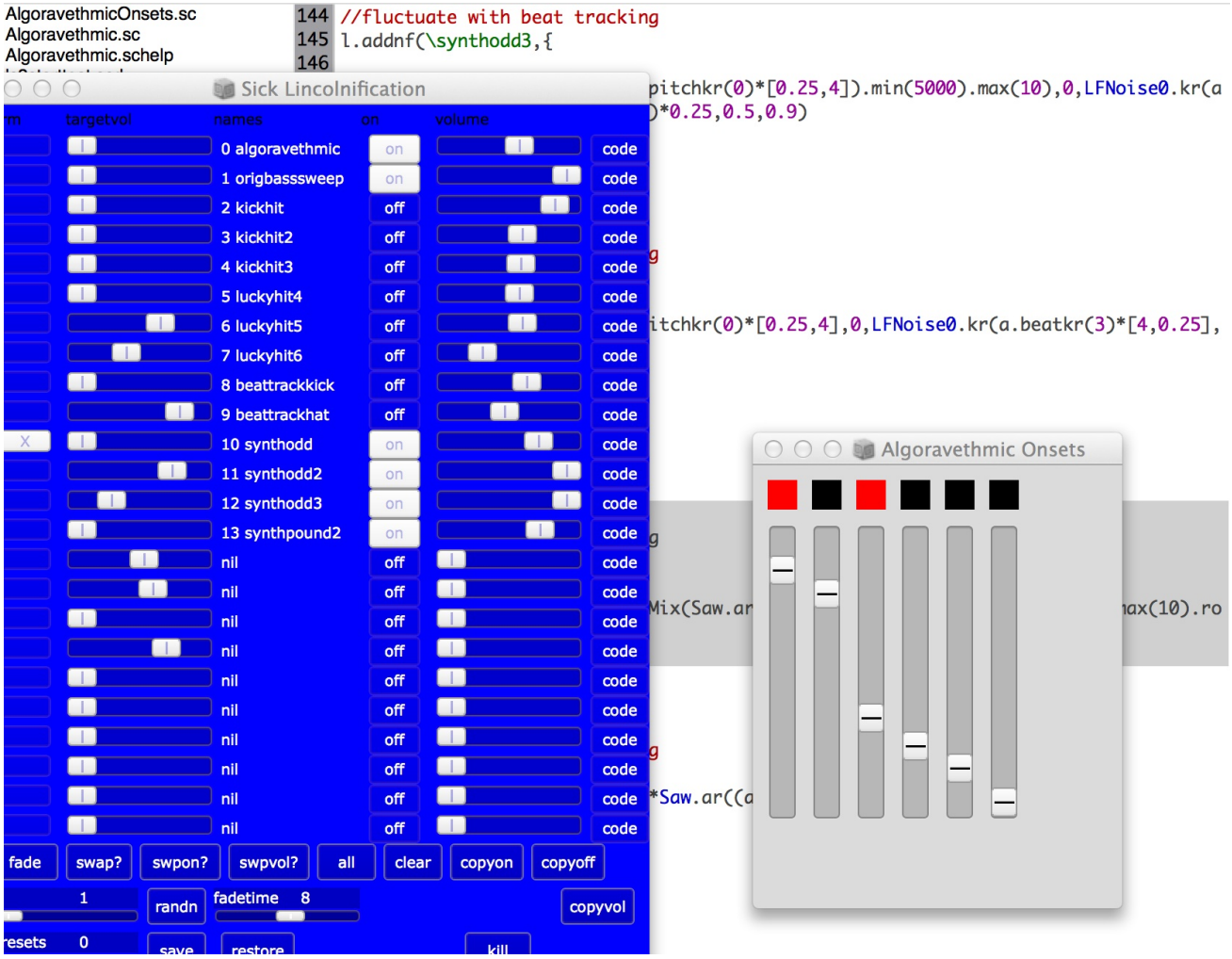


Figure 1: *Algoravethmic session in progress with live coding mixer and onset detection GUI*

An example of a more involved relationship between machine listening algorithms and live coding control is the *Algoravethmic* system (for SuperCollider), which empowers live remixing of an existing source work. The audio from the piece to be remixed is tracked by a variety of feature detectors, including beat tracking, onset detection, timbral and pitch content, and the results fed to control busses for re-use in feature-led synthesis and sound processing . The live coder works on patches exploiting the source-driven data to produce their remix. The live remix process is carried out within a live coding mixer, so that many different coded layers can be interwoven, including processing of the original source audio according to feature-adaptive effects and re-synthesis driven by feature data (Verfaille and Arfib 2001; Park, Li, and Biguenet 2008).

Line by line commands from a typical session might look like:

```
a = Algoravethmic();

(
```

```
//fluctuate with beat tracking
l.addnf(\origbasssweep,{
    BLowPass.ar(
        DelayN.ar(InFeedback.ar(a.busnum,1),0.02,0.02)*5,
        SinOsc.ar(a.beatkr(3)/4).range(200,3000)
        )
    })
)

//l contains the live coding mixer system, to be sent new sounds on the fly

//drum triggering
(
l.addnf(\kickhit,{
    PlayBuf.ar(1,~dubstepkicks[1],trigger:a.onsetkr(4))*3
})
)

l.addnf(\wobblyacid,{Blip.ar(a.pitchkr(0),a.timbrekr(3).range(1,100));})

//original track, delayed to help sync after machine listening delay
l.addnf(\original,{DelayN.ar(InFeedback.ar(a.busnum,1),0.02,0.02)*5})
```

where the beatkr, pitchkr, onsetkr and timbrekr are accessing different feature detector outputs on particular control busses.

Figure 1 illustrates the live coding mixer and a GUI for the bank of onset detectors (with threshold sensitivity controls) used for triggering new percussive events from the track to be remixed. Two of these detectors are adaptations of code from (Collins 2005; Goto 2001) specifically for kick and snare detection, implemented in custom SuperCollider UGens.

3. Machine listening as programming interface

The aforementioned graphical language perturbations of McLean and Sicchio (2014) provide a rare precedent for machine listening actually influencing the code interface itself. In this section two examples consider machine listening control of the instruction state list and memory contents of a running sound synthesis program. This should be understood as the radical adoption of machine listening as the basis for writing computer programs. In the two examples here, the programs so generated act within novel small scale programming languages specialised to musical output, but it is possible to envisage a machine listening frontend as the entry point to a more general purpose programming language.

3.1. TOPLAPapp variations in the web browser

TOPLAPapp is a promotional app for the TOPLAP organisation by this author, originally released as a free and open source iPhone app, and now available within the web browser in a javascript implementation thanks to Web Audio API. Javascript has enabled some immediate experiments extending the core TOPLAPapp instruction synthesis engine, for example, including a convolution reverb.

A version of *TOPLAPapp* with machine listening control has been devised (Figure 2). Machine listening with Web Audio API is a little more involved due to permissions requirements, but still feasible give or take a user having to allow a served web page access to their microphone. Indeed, the core code calls come down to:

```
navigator.getUserMedia({audio:true}, initAudio, function(e) {
            alert('Error getting audio');
            console.log(e);
        });

function initAudio(inputstream) {
```

```
audiocontext = new AudioContext();

audioinput = audiocontext.createMediaStreamSource(inputstream);

node = audiocontext.createScriptProcessor(1024, 1, 1); //mono in, mono out

audioinput.connect(node);

node.onaudioprocess = processAudio; //sample by sample processing function

}
```

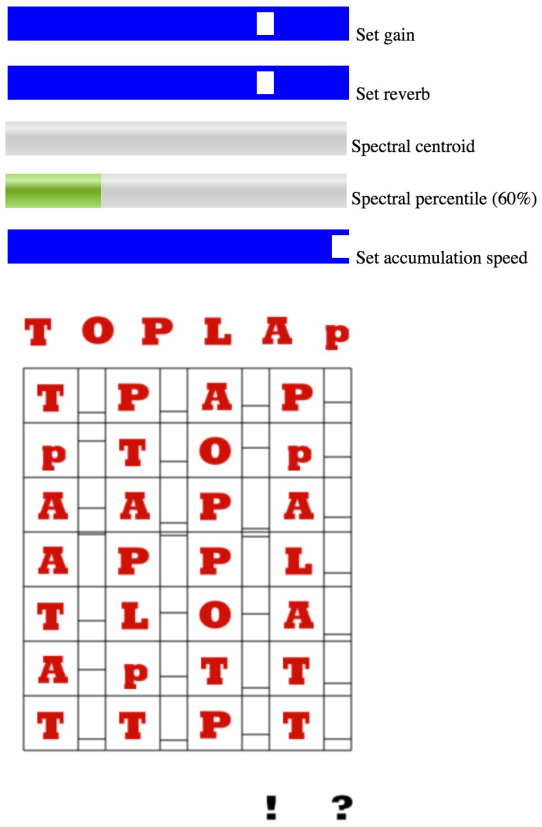


Figure 2: *TOPLAPapp with machine listening front end*

As a prelude to live spectral feature extraction, Nick Jones' fft.js code was co-opted; spectral centroid and 60% spectral energy percentile were then straight forward to calculate. The features were accumulated over multiple (non-overlapping) FFT frames to form feature means, where the duration between means is available as a user set parameter. For each new mean, one state in *TOPLAPapp*'s limited instruction set is updated, and one associated state value, according to:

```
stategrid.states[statepos] = Math.floor(5.9999*spectralcentroidmean);

stategrid.setSlider2(statepos,spectralpercentilemean);

statepos = (statepos +1 )%(stategrid.numunits);
```

Aside from vocal control of the state engine mediated via human ears, the system can also be sent into direct feedback, for example, by the simple expedient of having the built-in mic in a laptop allow feedback from built-in speakers, or via a virtual or physical patch cable. The reader can try out the process here with a recent compatible browser such as Chrome, and is encouraged to make a variety of brightnesses/frequency ranges of noise to reach different states and levels. The app can find itself restricted to only one state if left unstimulated by input, but also has allowed access to novel sonic outputs that the previous drag and drop state construction interface was too slow and deliberative to quickly explore, and the random function didn't tend to find.

3.2. Timbral instructions

A further experiment in machine listening control of a program was conducted in SuperCollider, with instruction states again determined from extracted features (in this case, perceptual loudness and spectral centroid). As the program stepped through the instruction list, it in turn influenced a memory array of floating point numbers, which are themselves used as source data for a monophonic pitched sequence (Figure 3).

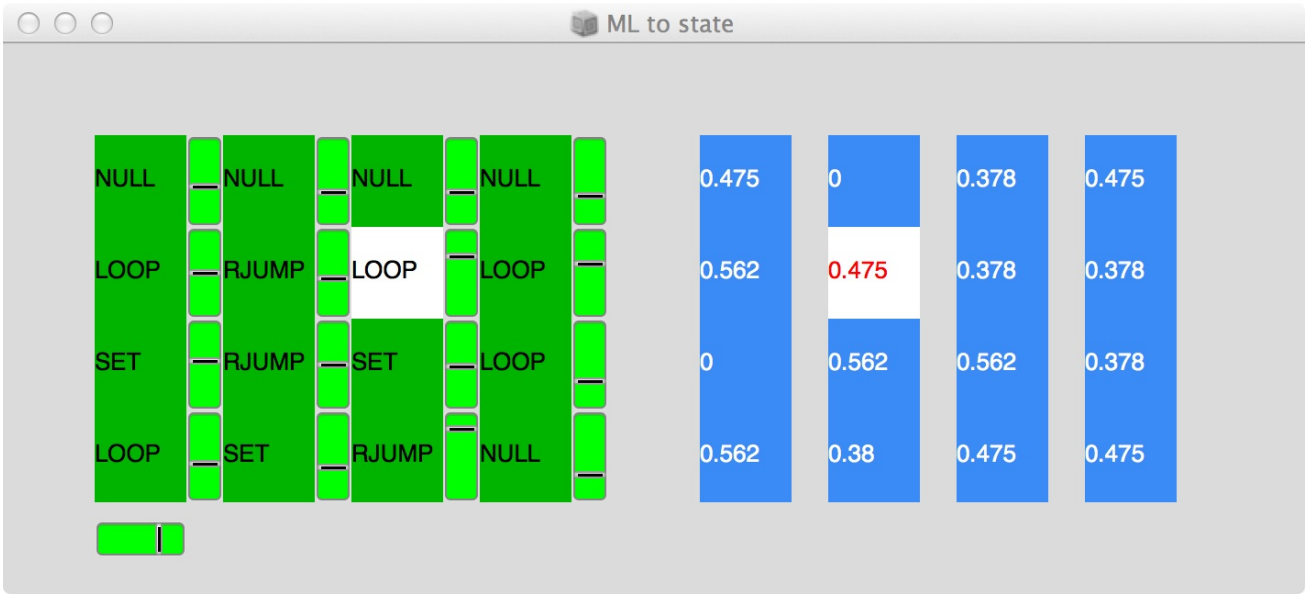


Figure 3: *Machine listening to instruction state engine to memory sequencer demonstration screenshot*

Each instruction slot has a corresponding state value (from 0.0 to 1.0 for easy remapping) to influence the result of the instruction applied to the memory. The (cyclic) instruction list is stepped through (clocked) at the same rate as the memory block (a slider lets a user set the clock rate), though their relative positions can diverge. The list of instructions in pseudocode are:

0) NULL : do nothing
1) JUMP : move in memory block to a position given proportionally by the state value
2) RJUMP : move in memory to a new random position
3) LOOP : loop on current memory location 2 to 8 times
4) SET : set memory contents at current location to state value
5) COPY : store state value for later use
6) PASTE : set memory contents at current location to last stored value
7) RAND : randomise memory contents at current location

The abstract relationship of the musical result to the machine listening is that from the contents of the memory block to the limited instruction set state engine which modulates it. Although there is little sense of reproducible controllability, there is a certain virtue to the unpredictable outputs easily obtained through audio input.

3.3. Further possibilities

The examples elucidated above are by no means exhaustive and are only meant to promote some of the potential interactions of machine listening and live coding.

A possibility not yet seriously exploited in live coding practice is the use of speech recognition to drive a live coding language. Hacking something together is perfectly possible using current generation tools; indeed, the dictation and

speech system preference within recent versions of OS X allows access to speech recognition within any text environment, including SuperCollider's IDE.

Live coding the fundamentals of a machine listening algorithm itself (aside from basic control parameters like a threshold or window size) remains a further tantilizing possibility. Aside from work to construct machine listening algorithms out of existing UGens (for instance, a SuperCollider SynthDef for a custom onset detector), just-in-time compilation facilities such as those in Extempore, Max/MSP's Gen, chuck's Chugins, or the ClangUGen for SuperCollider, provide routes to the live coding of sample by sample analysis DSP.

Are there worthwhile musical purposes, or would effort in coding of such live analysis DSP just be time wasted that could have been spent on details of the audio output? Well, changing the very nature of the computational listening experience underlying a current musical texture would seem an interesting avenue. Live coding offers unconventional takes on computer music performance, and for performance action to engage with an audio input tracking process as a dependent component within a signal chain has potential in this light.

A further option, feasible within the research domain, is an automated machine listening critic. Machine listening and learning over larger audio databases empowers this capability, and a specific corpus of live coding performances seems a feasible target. Deployment in live coding performance remains speculative at present; performance systems with live critics trained from MIR processes over larger audio databases have already been built by this author, though the on stage re-coding of such a critic is a challenge to come.

More unfettered speculation might ponder the future aesthetics of live coded machine listening work. Value may well be placed on individualised languages, perhaps through speech recognition or other sound taxonomic classification enabling coding to head to the territory of a unique constructed language between a given human and machine symbiont. A partnership of human and computer may build such a language as they grow up together; the virtuosity acheivable through such an interface, given substantial investment of learning, would be considerable, though it may only ever be intended for private use. For the form of a performance, the blank slate starting point frequently used in live coding extends straight forwardly from the blank page awaiting text to silence awaiting sound. Continuing the analogy, the full slate performance beginning from pre-composed code can be paralleled by an existing sound, analyzed via machine listening, the sound's features initialising the live code system which must be performed within; this may provide an alternative method of handover from performance to performance or performer to performer. Existing feedback systems incorporating audio, visual and textual material can themselves be extended through additional pathways, allowing generalised functions of multiple audiovisual and textual sources called via iteration or more complicated recursion. This article has said less concerning the multi-performer situation of multiple channels of machine listening information passed around a network, but interesting complexity lies therein. Whilst human listening remains the essential research challenge and touch point, there is also potential for aesthetics of novel machine listening systems divergent from the human ear; ultrasound and infrasound live coding, sensor data/live coding conglomerates, and the future artificially intelligent live coders with alternative auditory capabilities may provide some inspiration to researchers and artists, even if such systems will confound human audiences in further directions!

4. Conclusion

This paper has covered the conflation of live coding and machine listening, exploring different levels of re-configuration for such live analysis processes. There are a number of precedents, and this paper has also explored a number of projects including example code meant to illustrate the potential.

5. References

Blackwell, Alan, and Nick Collins. 2005. "The Programming Language as a Musical Instrument." *Proceedings of PPIG05 (Psychology of Programming Interest Group).*

Brown, Andrew R, and Andrew Sorensen. 2009. "Interacting with Generative Music Through Live Coding." *Contemporary Music Review* 28 (1): 17–29.

Casey, Michael, Remco Veltkamp, Masataka Goto, Marc Leman, Christophe Rhodes, and Malcolm Slaney. 2008. "Content-Based Music Information Retrieval: Current Directions and Future Challenges." *Proceedings of the IEEE* 96 (4) (April): 668–696.

Collins, Nick. 2005. "DrumTrack: Beat Induction from an Acoustic Drum Kit with Synchronised Scheduling." *Icmc*. Barcelona.

———. 2011. “Live Coding of Consequence.” *Leonardo* 44 (3): 207–211.

Collins, Nick, and Alex McLean. 2014. “Algorave: Live Performance of Algorithmic Electronic Dance Music.” *Proceedings of New Interfaces for Musical Expression*. London.

Collins, Nick, and Fredrik Olofsson. 2006. “Klipp Av: Live Algorithmic Splicing and Audiovisual Event Capture.” *Computer Music Journal* 30 (2): 8–18.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. “Live Coding Techniques for Laptop Performance.” *Organised Sound* 8 (3): 321–29.

Goto, Masataka. 2001. “An Audio-Based Real-Time Beat Tracking System for Music with or Without Drum-Sounds.” *Journal of New Music Research* 30 (2): 159–71.

Klapuri, Anssi, and Manuel Davy, ed. 2006. *Signal Processing Methods for Music Transcription*. New York, NY: Springer.

Magnusson, Thor. 2014. “Herding Cats: Observing Live Coding in the Wild.” *Computer Music Journal* 38 (1): 8–16.

McLean, Alex. 2011. “Artist-Programmers and Programming Languages for the Arts.” PhD thesis, Department of Computing, Goldsmiths, University of London.

McLean, Alex, and Kate Sicchio. 2014. “Sound Choreography<> Body Code.” In *Proceedings of the 2nd Conference on Computation, Communcation, Aesthetics and X (XCoAx)*, 355–362.

Park, Tae Hong, Zhiye Li, and Jonathan Biguenet. 2008. “Not Just More FMS: Taking It to the Next Level.” In *Proceedings of the International Computer Music Conference*. Belfast.

Stowell, Dan. 2010. “Making Music Through Real-Time Voice Timbre Analysis: Machine Learning and Timbral Control.” PhD thesis, School of Electronic Engineering; Computer Science, Queen Mary University of London. http://www.mcld.co.uk/thesis/.

Stowell, Dan, and Alex McLean. 2013. “Live Music-Making: a Rich Open Task Requires a Rich Open Interface.” In *Music and Human-Computer Interaction*, 139–152. Springer.

Verfaille, Vincent, and Daniel Arfib. 2001. “A-DAFx: Adaptive Digital Audio Effects.” In *International Conference on Digital Audio Effects (DAFx)*. Limerick.

Ward, Adrian, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. 2004. “Live Algorithm Programming and a Temporary Organisation for Its Promotion.” In *Proceedings of the README Software Art Conference*, 243–261.

Yee-King, Matthew John. 2011. “Automatic Sound Synthesizer Programming: Techniques and Applications.” PhD thesis, University of Sussex.

Patterns of User Experience in Performance Programming

Alan F. Blackwell
University of Cambridge Computer Laboratory
Alan.Blackwell@cl.cam.ac.uk

ABSTRACT

This paper presents a pattern language for user experiences in live coding. It uses a recently defined analytic framework that has developed out of the Cognitive Dimensions of Notations and related approaches. The focus on performance programming offers particular value in its potential to construct rigorous accounts of the experiences of both performers (the live coder) and audiences. Although developed as an account of live coding, the findings promise to be relevant to a wider range of performance programming contexts, which could benefit from analysis in terms of live coding, if a systematic framework of this kind were available. The paper provides a detailed analytic commentary, drawing on the broadly diverse body of prior live coding practice, but using music live coding as a central running example. The findings demonstrate the advantage of rigorous analysis from an independent theoretical perspective, and suggest the potential for future work that might draw on this pattern language as a basis for empirical investigations of user experience, and as a theoretical grounding in practice-led design of new live coding languages and tools.

1. INTRODUCTION

The concept of *software patterns* – construction techniques that can be applied in many situations – is extremely well known in the software engineering industry, regularly applied by professionals and widely taught to undergraduates (e.g. Larman 2004, Holzner 2006). However, Blackwell and Fincher have argued (2010) that there has been a fundamental misunderstanding in the way this concept is being developed and applied. They observed that the software engineering researchers who originally coined the term had not intended it to be a catalogue of construction techniques, but rather based it on an analogy to the writing of architect and design philosopher Christopher Alexander (1978).

Alexander created the term *pattern language* to describe his systematized observations of the ways that people actually use and inhabit their built environment. He argued that buildings are shaped as much by their users as by materials and construction, and that it is useful for architects to have a systematic description of the things users experience. An example is the pattern "light on two sides of a room". This might be achieved, for example, when an architect, builder or home renovator installs glass windows in two different walls. However, the essence of the pattern is a description of the experience of light, not a prescription for achieving it (for example, by specifying how to place windows, what they are made out of, or even whether they are windows at all). Alexander's pattern language is a systematic collection of experience descriptions, not a set of rules for construction.

The goal of this paper is to apply an analogous approach to the study of the human context in which live coding is done, in order to achieve a systematic collection of descriptions of the user experiences that constitute live coding. Such a collection is potentially valuable both as a design resource (drawing attention to new opportunities for live coding tools or languages) and a critical resource (providing a vocabulary with which to describe the tools and languages of today).

The application of Alexander's pattern language to software was originally proposed by Kent Beck and Ward Cunningham (1987) and popularized in the form that is familiar today by the "Gang of Four": Gamma, Helm, Johnson and Vlissides (1994). The original analogy between software engineering and architecture was clearly expressed by Richard Gabriel as follows:

> *Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently. It should be clear that, in our context,* ***a "user" is a***

> ***programmer*** *who is called upon to maintain or modify software; a user is not (necessarily) the person who uses the software. In Alexander's terminology, a user is an inhabitant* (Gabriel 1993, emphasis added)

Subsequent extension of this work to user interface patterns (e.g. Tidwell 2005) has abandoned this insight from Beck, Cunningham and Gabriel that Alexander's ideas could be used to understand contexts where 'a "user" is a programmer' as in the quote above. In most user interface '"patterns", the user does *not* get to be a programmer, and does not have the creativity and control that arises from programming. Blackwell and Fincher (2010) argue that this situation may have arisen from misunderstanding of the original analogy, which has become more widespread as other "patterns" publications rely on secondary sources rather than the original publications by Beck and Cunningham. However, live coding offers an ideal opportunity to draw away from the (mis)understanding that may be implicit in the UI patterns community, and instead refocus on the creative power of programming, and on source code as the primary medium of interaction, rather than secondary to some other UI. Live coding is especially interesting because of the contrast it offers to mainstream software engineering (Blackwell and Collins 2005). By considering Alexander's (and Gabriel's) application of design patterns, the goal of this paper is to understand what it is that makes live coding distinctive, in relation to more conventional software engineering, and to other kinds of user experience.

The remainder of this paper is structured as follows. This introduction is fleshed out, first with a more precise characterization of performance programming, followed by more detail of Blackwell and Fincher's critique of conventional software and user interface "patterns", and then an introduction to their alternative pattern language as it applies to live coding. The main body of the paper describes that pattern language in detail, as the main contribution of the paper. A closing discussion reviews the validity and potential value of this approach.

1.1 The User Audience of Performance Programming

This paper takes a particular interest in the frequent interpretation of live coding as performance programming. Although live coding can be interpreted in other ways (in particular, as a software engineering method, as an educational strategy, or as a creative medium), the performance context introduces the key dynamic between the performer and audience. This dynamic immediately highlights many of the trade-offs that are inherent in the varieties of user experience captured by pattern languages. In particular, it is understood that performers and audiences have *different* experiences – a factor that is often neglected in the implicit universalism of software engineering methodology, and in mainstream theoretical research into principles of programming languages.

The performance context addressed by this paper can be understood primarily in terms that are familiar conventions of art-coding – where the code is written by a performer on stage, with a projection screen displaying the code as a stage backdrop, and the audience are either dancing (at an algorave), or sitting (in a concert/recital setting). This conventional layout might be varied slightly, for example when the performance accompanies a conference talk, when the live coder is providing background music at a function, or in a mixed genre variety show or cabaret context. However, the observations in the paper are also applicable to other presentation formats that maintain the performer/audience distinction, but replicated in digital media. In particular, video of live coding performances, whether recorded in one of the settings already described, or created specifically for online distribution, attracts larger audience numbers than concerts and algoraves.

The findings from this paper are also relevant to a further, perhaps even more widely viewed, online genre – the programming screencast. Screencasts are often recorded from a live 'performance', but typically focus on demonstrating software skills, or the features of software development tools, rather than on recording a stage performance. There is already substantial overlap between these genres and live coding as understood at this conference, for example in Sam Aaron's tutorial introduction *How to Hack Overtone with Emacs*[1]. Of course screencasts of art-coding are a minority interest by comparison to "performances" of programming with mainstream tools, whether tutorials such as *Data Driven Programming in Haskell*[2] or exhibitions of skill

[1] https://vimeo.com/25190186
[2] https://www.youtube.com/watch?v=045422s6xik

by popular coding celebrities such as Minecraft developer Notch[3]. The recent launch of a dedicated site (www.livecoding.tv) for performances of this kind suggests that this trend will continue to increase.

In the remainder of the paper, I assume for simplicity of presentation that the performer is creating music through live coding. I therefore use examples from music to illustrate the kinds of information structure involved in code. However, it is intended that these observations should apply equally to other live coding environments controlling other media – for example, my own Palimpsest system for live coding of visual imagery (Blackwell 2014), as well as the games, data analytics and software engineering examples above.

1.2 Patterns of User Experience

A key assumption of the presentation in this paper is that conventional uses of the term "pattern language" in software engineering and in human-computer interaction are based on a *misunderstanding* of Beck, Cunningham and Gabriel's original analogy between programming and architecture (Blackwell and Fincher 2010). This paper is therefore *not* primarily concerned with "software design patterns" as presented in Gamma et al (1994) and many other texts. Blackwell and Fincher (2010) argue that those texts are concerned mainly with construction techniques, rather than user experiences. This paper is particularly *not* concerned with "user interface patterns", as presented in many HCI texts and resource sites (e.g. http://ui-patterns.com/) which collect examples of specific construction techniques by analogy to software design patterns. As noted in the introduction to this paper, Blackwell and Fincher (2010) argue that this literature continues the misunderstanding of pattern languages, through a focus on construction of user interfaces rather than the user experience of programming. Although it has been necessary to cite this literature in order to avoid further misunderstanding, this paper will not make any further reference to either software patterns or user interface patterns literatures.

Blackwell and Fincher's proposal for the creation of a pattern language of user experience in software was modeled closely on the Cognitive Dimensions of Notations (CDs) framework originally proposed by Thomas Green (1989), greatly expanded by Green and Petre (1996), and incrementally developed with many collaborators since then, including the present author and many others. Several alternative perspectives on CDs have been proposed, introducing corresponding approaches to tangible user interfaces (Edge 2006), collaborative representations (Bresciani 2008) and several others. Blackwell and Fincher argued that the goal of CDs was far more closely related to Alexander's pattern language, in that it aimed to provide a vocabulary with which the designers of programming languages and other complex notations could discuss the usability properties of those notations. An example CD was "viscosity" – the experience of using a system in which small changes are unreasonably hard to achieve, like "wading through treacle". The vocabulary was grounded in previous literature, but was a "broad brush" description rather than focusing on details of the kind that can be measured in controlled laboratory studies, or time-and-motion style exploration of efficiency in use. The popularity of the framework arose in part because labels such as "viscosity" captured intuitive feelings that were recognized by many programmers as aspects of their everyday experience, but aspects that they had not previously had a name for.

In the 25 years since Green originally proposed the Cognitive Dimensions of Notations framework, it has been highly influential as an approach to understanding the usability of programming languages. The paper by Green and Petre (1996) was for many years the most highly cited original paper to be published in the Journal of Visual Languages and Computing, and a 10th anniversary issue of that journal reviewed the many ways in which the framework had been applied, including its application in mainstream commercial programming products such as Microsoft's Visual Studio (Dagit et al. 2006, Green et al. 2006). This paper cannot review that literature in detail, but it extends to practical design guidance for programming languages and APIs, evaluation methods for programming tools, theories of user interaction, requirements capture methods and others.

Based on this substantial and diverse body of research, the author has developed a teaching guide, which synthesizes a pattern language of user experience from all these different sources, variants and extensions of the original CDs framework. This guide has been used in contexts that are intentionally distinct from the programming language research that has been the main focus of CDs until now. The most substantial application to date has been in a graduate course within a professional master's programme for sustainability leadership[4]. The goal of that course has been to highlight the role played by representation systems, when global challenges are being negotiated in the presence of technology. It is taught to students

[3] https://www.youtube.com/watch?v=rhN35bGvM8c

with no prior experience of interaction design, using a case study of mobile GPS usage by indigenous people negotiating with logging companies in the Congo Basin (Lewis 2014). This course has provided an opportunity to expand the concept of a representational pattern language to the widest possible extent, encompassing considerations of physical context, language, culture, colonialism, embodiment, ecosystems, distributed and locative media, and many others.

1.3 Interrogating User Experience in Performance Programming

Starting from a pattern language that has been independently developed to analyse the user experience of novel digital interventions, this paper now analyses performance programming from that perspective. The goal of the analysis is to explore the ways in which performance programming exhibits (or does not exhibit) patterns that are also seen in other digital media contexts. These patterns are expressed, as in the Cognitive Dimensions of Notations framework, in terms of interaction with an *information structure*, where the structure is (in the digital music context) the configuration of synthesisers, samples, filters, instruments and other software components that generate sound through digital means.

In keeping with mainstream practice of analysis using CDs, the patterns are used as a discussion tool, providing a structured basis for systematic consideration of different aspect of user experience. In classical CDs analysis, the characteristic user activities are first defined, together with a profile of dimensions that are most salient for that kind of activity. Each dimension is then considered in turn, in order to observe how the particular combination of notation and environment will support the identified activities. This consideration also includes trade-offs where improvement on one dimension might result in detriment for another, and possible work-arounds that users would engage in. Each of these elements has been included in the following analysis, using appropriate analogies to the broader class of user experience patterns now identified.

The specific form in which the patterns are expressed below, using reference numbers to identify particular patterns, allows the reader to cross-reference this analysis with a forthcoming publication in which the patterns of user experience have been applied in another non-programming context, to the design of novel diagrammatic representations (Blackwell in press). An appendix in that publication can also be consulted for more detailed derivations that relate each pattern to the previous bodies of research from which is has been developed, in particular drawing on the various iterations and variants of the Cognitive Dimensions of Notations framework.

2. PATTERNS OF ACTIVITY IN PERFORMANCE PROGRAMMING

As with the architectural analyses of Christopher Alexander, patterns in the user experience of live coding can be described from different perspectives. This section contrasts the different modes in which people interact with code, expressed as types of activity that are characteristic of *Interpretation*, *Construction* and *Collaboration* activities. In each case, these are activities that have been observed in other areas of notation and representation use, and this pre-existing descriptive framework is used to interrogate different aspects of performance programming, stepping aside from conventional descriptions that are already familiar in the critical discourse of live coding.

Each of these kinds of activity, as with the more limited set of activities that has previously been described in the Cognitive Dimensions framework, is associated with a different profile of user experience patterns that are found to be particularly salient in the context of that activity. These profiles are briefly indicated, in the following discussion, using reference numbers referring to the experience patterns that will be described later in the paper. This numbering scheme is consistent with (Blackwell in press), to allow further citations of previous work.

2.1 Interpretation activities: reading information structure

In the performance programming setting, the audience interprets the information structure that the performer constructs. Interpretation activities include: **Search**: The audience often follow the current cursor position, but may also search for the code that was responsible for producing a particular audible effect. This activity is enabled by patterns VE1, VE4, SE3, TE4 below. **Comparison**: The audience constantly compares different pieces of code – both diachronically (the state of the code before and after a change), and

[4] Master of Studies at the Cambridge Institute for Sustainability Leadership
http://www.cisl.cam.ac.uk/graduate-study/master-of-studies-in-sustainability-leadership

synchronically (comparing expressions on different parts of the screen in order to infer language features by comparison). This activity is enabled by patterns VE5, SE4, ME4, TE2 below. **Sense-making**: Live coding audiences often have no prior expectation of the code that is about to be produced, so they must integrate what they see into a mental model of its overall structure. This activity is enabled by patterns VE2, VE3, SE1, ME1, ME3, TE3, TE5 below.

2.2 Construction activities: building information structure

The performer is defining, in various layers, an abstract syntax tree, a synthesiser configuration, or a musical structure with audible consequences that will be perceived by listeners. The following activities represent different approaches to the creation of this structure. **Incrementation:** The structure of the code has been established, but a piece of information is added – e.g. another note, a synthesiser parameter. This activity is enabled by patterns IE1, PE6 below. **Transcription:** The structure is already defined, and the performer needs to express it in the form of code. Perhaps it has been rehearsed or pre-planned? This activity is enabled by patterns ME2, IE2, IE3, IE5, PE2, PE5 below. **Modification:** Is a constant of live coding – many live coders commence a performance with some code that they may modify, while even those who start with a blank screen restructure the code they have written. This activity is enabled by patterns SE2, ME5, IE4, TE1, PE1, CE1 below. **Exploratory design:** Despite the musically exploratory nature of live coding, it is rare for complex data structures or algorithms to be explored in performance – primarily because an executing program can only gradually change its structure while executing. As a result, live coding is surprisingly unlikely to exhibit the kinds of "hacking" behaviour anticipated when the distinctive requirements of exploratory software design were highlighted by the Cognitive Dimensions framework. This activity is enabled by patterns TE5, PE3, PE4, CE2, CE3, CE4 below.

2.3 Collaboration activities: sharing information structure

Descriptions of collaboration were added relatively recently to the Cognitive Dimensions framework (Bresciani et al 2008), perhaps because the stereotypical expectation is that programming is a relatively solitary activity. However, the performance programming context immediately introduces a social and "collaborative" element, if only between the performer and audience. The activities that have been identified in previous work include: **Illustrate a story**: Originally intended to describe situations in which a notation is used to support some other primary narrative, this provides an interesting opportunity to consider the audience (and performer) perspective in which the music is the primary medium, and the code, to the extent it is shared, is there to support this primary function. This activity is enabled by patterns VE2, VE4, IE6, TE1, CE3 below. **Organise a discussion:** At first sight, this seems a foreign perspective to live coding. An algorave is not a discussion. But one might consider performance settings such as conferences, or comments on online video, in which the audience do respond to the work. Does the code play a part? Furthermore, on-stage improvisation among a group of performers can be analysed as discursive – does the code support this? This activity is enabled by patterns ME5, IE2, TE2, PE3, PE4, CE4 below. **Persuade an audience**: What is the visual rhetoric embedded in code? Many notation designers are reluctant to admit the rhetorical or connotative elements of the representations they create, and these design elements of live coding languages are often left relatively tacit. This activity is enabled by patterns VE3, SE4, ME2, ME6, IE5, TE3, TE5 below.

3. DESIGN PATTERNS FOR EXPERIENCE IN USE

This section offers a comprehensive description of patterns in user experience that result from, or are influenced by, the design of different notations or tools. As discussed in the previous section, different aspects of the performance programming context are enabled by different subsets of these patterns. This set of patterns can potentially be used either as a checklist of design heuristics, for those who are creating new live coding tools, or as a means of reflecting on the capabilities of the tools they already have. In particular, there are many design decisions that carry *different* implications for audiences and performers, which would appear to be an important critical consideration for the design and evaluation of performance programming technologies.

3.1 Experiences of Visibility

Visibility is essential to performance programming – the TOPLAP manifesto demands "show us your screens."[5] It is therefore completely appropriate that this group of patterns so often comes before all others. **VE1: The information you need is visible:** The constraints imposed by the performance programming context are extreme – a single screen of code is projected or streamed, in a font that must be large enough for an audience to see. **VE2: The overall story is clear:** Can code be presented in a way that the structure is apparent, or might it be necessary to leave out some of the detail in order to improve this overall understanding? **VE3: Important parts draw your attention:** When there is such a small amount of code visible, perhaps this isn't much of a problem. However, it seems that many live coders support their audiences by ensuring that the current insertion point is prominent, to help in the moment engagement with the performance. **VE4: The visual layout is concise:** At present, the constraints already described mean that the importance of this pattern is unavoidable. **VE5: You can see detail in context:** This seems to be an interesting research opportunity for live coding. At present, many live coding environments have adopted the buffer-based conventions of plain text editors, but there are better ways of showing structural context, that could be borrowed from other user interfaces, IDEs or screen media conventions.

3.2 Experiences of Structure

In the running example used in this paper, it is assumed that the "structures" implicit in the performance programming context are the structures of music synthesis. However, as already noted, this running example should also be interpreted as potentially applying to other kinds of live-coded structured product – including imagery, dance and others. In all these cases, the understanding inherited from the Cognitive Dimensions framework is that the information structure consists of 'parts', and relationships between those parts. **SE1: You can see relationships between parts:** Are elements related by a bounded region, or lines drawn between them? Do multiple marks have the same colour, orientation, or size? Are there many verbal or numeric labels, some of which happen to be the same, so that the reader must remember, scan and compare? **SE2: You can change your mind easily:** This is a critical element supporting the central concern with improvisation in live coding. The problems associated with this pattern were captured in Green's most popular cognitive dimension of "Viscosity – a sticky problem for HCI" (1990). **SE3: There are routes from a thing you know to something you don't:** Virtuoso performers are expected to hold everything in their heads. Would it be acceptable for a live coder to openly rely on autocompletion, dependency browsers, or API tutorials when on stage? **SE4: You can compare or contrast different parts:** The convention of highly constrained screen space means that juxtaposition is usually straightforward, because everything necessary is already visible, allowing the performer to judge structural relationships and the audience to interpret them.

3.3 Experiences of Meaning

In many programming languages, the semantics of the notation are problematic, because the programmer must anticipate the future interpretation of any given element. In live coding, meaning is directly available in the moment, through the concurrent execution of the program being edited. As a result, there is far greater freedom for live coding languages to explore alternative, non-literal, or esoteric correspondences. **ME1: It looks like what it describes:** Live coders often use this freedom for a playful critique on conventional programming, with intentionally meaningless variable names, ambiguous syntax and so on. **ME2: The purpose of each part is clear:** Within the granularity of the edit/execute cycle, the changes that the audience perceive in musical output are closely coupled to the changes that they have observed in the source code. As a result, "purpose" is (at least apparently) readily available. **ME3: Similar things look similar**: At present, it is unusual for live coders to mislead their audiences intentionally, by expressing the same behaviour in different ways. This could potentially change in future. **ME4: You can tell the difference between things:** In contrast, live coding languages often include syntax elements that may be difficult for the audience to distinguish. A potential reason for this is that the resulting ambiguity supports a richer interpretive experience for the audience. This is explored further in CE2 and CE3 below. **ME5: You can add comments:** A simplified statement of the cognitive dimension of secondary notation, this refers to those aspects of the code that do not result in any change to the music. In performance, the text on the screen offers a side-band of communication with the audience, available for commentary, meta-narratives and many other intertextual devices. **ME6: The visual connotations are appropriate:** The visual aesthetic

[5] http://toplap.org/wiki/ManifestoDraft

of live coding environments is a highly salient aspect of their design, as appropriate to their function as a performance stage set. The muted greys of Marc Downie's *Field*, the ambient radar of Magnusson's *Threnoscope*, the vivid pink of Aaron's *Sonic Pi*, and the flat thresholded geometry of my own *Palimpsest* are all distinctive and instantly recognisable.

3.4 Experiences of Interaction

These patterns relate to the user interface of the editors and tools. It is the performer who interacts with the system, not the audience, so these are the respects in which the greatest tensions might be expected between the needs of the two. **IE1: Interaction opportunities are evident:** This is a fundamental of usability in conventional user interfaces, but the performance of virtuosity involves the performer giving the impression that all her actions are ready to hand, perhaps in a struggle against the obstacles presented by the "instrument". Furthermore, the performer may not want the audience to know what is going to happen next – for example, in my own performances with Palimpsest, I prefer the source images that I intend to manipulate to be arranged off-screen, on a second monitor that is not visible to the audience, so that I can introduce them at the most appropriate points in the performance. The question of choice is always implicit, but a degree of mystery can enhance anticipation. **IE2: Actions are fluid, not awkward:** Flow is essential to both the experience and the illusion of performance. For many developers of live coding systems, this is therefore one of their highest priorities. **IE3: Things stay where you put them:** Predictability is a virtue in most professional software development contexts, but in improvised performance, serendipity is also appreciated. As a result, the representations are less static than might be expected. An obvious example is the *shift* and *shake* features in Magnusson's ixi lang, whose purpose is to behave in opposition to this principle. **IE4: Accidental mistakes are unlikely:** On the contrary, accidents are an opportunity for serendipity, as has been noted since the earliest discussions of usability in live coding (e.g. Blackwell & Collins 2005). **IE5: Easier actions steer what you do:** In many improvising genres, it is understood that the affordances of the instrument, of the space and materials, or of the body, shape the performance. However, easy licks quickly become facile and trite, with little capacity to surprise the audience. As a result, every new live coding tool can become the starting point for a miniature genre, surprising at first, but then familiar in its likely uses. **IE6: It is easy to refer to specific parts:** Critical commentary, education, support for user communities, and the collaborative discourse of rehearsal and staging all require live coding performers to talk about their code. Although ease of reference is seldom a salient feature early in the lifecycle of a new tool, it becomes more important as the audience expands and diversifies.

3.5 Experiences of Thinking

The kinds of thinking done by the performer (creative decisions, planning the performance, analysing musical structures) are very different to those done by the audience (interpretation, active reception, perhaps selecting dance moves). Both appear rather different to conventional professional programming situations, which far more often involve reasoning about requirements and ways to satisfy them. **TE1: You don't need to think too hard**: Cognitive load, resulting from working memory and dependency analysis, is less likely to be immediately challenging in live coding situations because both the code and the product are directly available. However, audiences often need to interpret both the novel syntax of the language, and the performer's intentions, meaning that they may (unusually) have greater cognitive load than the performer – if they are concentrating! **TE2: You can read-off new information**: Purely textual languages are unlikely to exhibit this property, but visualisations such as Magnusson's *Threnoscope* exhibit relations between the parts beyond those that are explicit in the code, for example in phase relations that emerge between voices moving with different velocity. **TE3: It makes you stop and think**: The challenge noted for audiences in TE1 here becomes an advantage, because it offers opportunities for richer interpretation. **TE4: Elements mean only one thing**: Lack of specificity increases expressive power at the expense of abstraction, often resulting in short-term gains in usability. In live coding, the performer is usually practiced in taking advantage of the available abstraction, while the audience may appreciate interpretive breadth. **TE5: You are drawn in to play around:** Clearly an advantage for performing programmers, and less problematic in a context where risk of the program crashing is more exciting than dangerous. However, playfulness is supported by the ability to return to previous transient states (IE3).

3.6 Experiences of Process

Many conventional software development processes include assumptions about the order in which tasks should be done – and this is less common in live coding. However, it is useful to remember that rehearsal and composition/arrangement do involve some different processes that may not be involved in performance situations (for example, Sam Aaron notes in another paper at this conference that he keeps a written diary while rehearsing). **PE1: The order of tasks is natural:** Every user may have a different view of what is 'natural', although live coding audiences appreciate an early start to the music. **PE2: The steps you take match your goals:** Live coding is a field that shies away from prescription, and 'goals' often emerge in performance. Nevertheless, IE5 results in recognisable patterns that may not have been conceived explicitly by the performer. **PE3: You can try out a partial product:** Absolutely inherent in live coding – a 'finished' live coded program seems almost oxymoronic! **PE4: You can be non-committal:** Another high priority for live-coded languages, which often support sketch-like preliminary definitions. **PE5: Repetition can be automated:** Every performing programmer is aware of developing personal idiomatic conventions. Occasionally the language is extended to accommodate these automatically, but more often, they become practiced in the manner of an instrumental riff, or encoded through muscle memory as a 'finger macro.' **PE6: The content can be preserved:** Some live coders habitually develop complex libraries as a jumping-off point for improvisation (e.g. Andrew Sorensen), while others prefer to start with an empty screen (e.g. Alex McLean). Whether or not code is carried forward from one performance to another, it might be useful to explore more thoroughly whether the temporality of the performance itself can be preserved, rather than simply the end result.

3.7 Experiences of Creativity

In analysis of conventional software engineering, this final class of experience might appear trivial or frivolous. However, when programming is taking place in a performing arts context, 'creativity' (of some kind) is assumed to be one of the objectives. This suggests that these patterns will be among the most valued. **CE1: You can extend the language:** Live coders constantly tweak their languages, to support new performance capabilities, but this most often happens offline, rather than during performance. Language extension in performance is conceivable, and may offer interesting future opportunities, but would greatly increase the challenge to audience and performer in terms of TE4. **CE2: You can redefine how it is interpreted:** The predefined semantics of most mainstream programming languages means that open interpretation may be impractical. But in live coding, opportunities for increased expressivity, and serendipitous reinterpretation, suggest that this experience is likely to be more highly valued. **CE3: You can see different things when you look again:** This is a key resource for inspiration in improvised performance contexts, to generate new ideas for exploration, and to provide a wider range of interpretive opportunities for the audience. As a result, this represents one of the most significant ways in which live coding languages differ from conventional programming languages, where ambiguity is highly undesirable. **CE4: Anything not forbidden is allowed:** Many programming language standards attempt to lock down the semantics of the language, so that all possible uses can be anticipated and tested. In creative contexts, the opposite is true. An execution result that is undesirable in one performance may be completely appropriate in another, and digital media artists have often revelled in the glitch or crash, as a demonstration of fallibility or fragility in the machine.

4. DISCUSSION

This paper has investigated the nature of user experience in live coding, and also in other contexts that might more broadly be described as performance programming, characterized by the presence of an audience observing the programming activity. This investigation has proceeded by taking an existing analytic framework that describes patterns of user experience, in the style of an architectural pattern language, and has explored the insights that can be obtained by considering performance programming from the perspective of this framework.

4.1 Validity of the Analysis

The validity of this analysis relies on two precautions: Firstly, the existing framework is applied without modification or adaptation. Rather than selecting elements from the framework by convenience or apparent surface relevance, the structure and content of the existing framework has been applied without modification. This provides an opportunity for disciplined consideration of all possible factors, whether or

not they might have been considered to be relevant at first sight. This first precaution has demonstrated some novel advantages, for example by drawing closer attention to the user experience of the audience in performance programming situations. The value of this precaution can be understood by contrast with previously less successful attempts to apply the Cognitive Dimensions framework, in which researchers ignored, or redefined, particular dimensions that were inconvenient or inconsistent with their assumptions (as discussed in Blackwell & Green 2000).

The second precaution is that the framework chosen for application is intentionally tangential to the existing analytic approaches that have been applied in live coding, thus offering a form of qualitative triangulation. There have been previous analyses of live coding from the perspective of performance studies, and of programming language design. Although these prior analyses have resulted in a useful emerging body of theory, there is a tendency for such bodies of theory to be self-reinforcing, in that analytic design discourse is grounded in the body of theory, and empirical investigations of its application reflect the same assumptions. In contrast, the framework applied here was developed for analysis of visual representation use, in situations ranging from central African forests to Western business, education and domestic settings. Although drawing heavily on the approach pioneered in the Cognitive Dimensions of Notations, this framework has subsequently been adapted for an audience of visual representation designers, with many changes of terminology and structure (Blackwell in press). One consequence is that the terminology may appear less immediately relevant to live coding than is the case with familiar Cognitive Dimensions such as secondary notation. But the consequent advantage is to defamiliarise the discussion of user experience in live coding, allowing us to achieve a systematic analysis from a novel perspective, and an effective triangulation on the understanding of interaction in live coding.

4.2 Applying the Analysis

The next phase of this work is to apply the results of this analytic investigation as a starting point for empirical study and practice-led design research. The immediate opportunity for empirical study is to use the findings from this investigation as a coding frame[6] for the analysis of qualitative descriptions of user experiences in performance programming, whether taken from audiences or from performers. The framework will be beneficial to such situations, by providing a theoretically-grounded organisational structure rather than an *ab initio* categorisation. There is also a possible advantage for the framework itself, in that new sources of experience reports, from situations as different as algoraves and screencast videos, may identify new patterns of experience that have not previously been noticed in other programming and formal representation contexts.

The second opportunity is to use the results of this analysis as a resource for design research. This pattern language of user experience in performance programming retains the potential of the original Cognitive Dimensions framework, that it can be employed as a discussion vocabulary for use by language designers. It points to design opportunities, as noted at points throughout the above analysis. It offers the potential for reflection on the consequences of design decisions. It draws attention to known trade-offs, although this paper does not have space for the detailed listing of those trade-offs described elsewhere (Blackwell in press). Finally, it offers the potential for use as an evaluative framework, for example in the manner of Heuristic Evaluation, or as a basis for the design of user questionnaires, in the manner of the Cognitive Dimensions questionnaire (Blackwell & Green 2000).

The discussion of live coding in this paper has primarily drawn on music live coding systems to illustrate the relationship between structure and representation. This choice was made in part because the author is more familiar with music than with other performing arts. However, it is intended that this analysis can be applied to many other arts contexts, and the author's preliminary investigations have already included study of dance (Church et al. 2012) and sculpture (Gernand et al. 2011), as well as experiments in live-coded visual imagery (Blackwell & Aaron 2014). Current live coding research by others, for example McLean's performances of textile craft practices such as knitting and weaving, is also likely to benefit from this analytic approach.

5. CONCLUSIONS

Systematic description of user experience is a valuable design resource, as noted in the development and subsequent popularity of Christopher Alexander's pattern language for the design of the built environment.

[6] Note that the term "coding frame" is used here as a technical term in qualitative data analysis, which has no relationship to "coding" of software.

The idea of the pattern language has been appropriated enthusiastically in software engineering and in user interface design, although in a manner that has lost some of the most interesting implications in studying experiences of computation. Live coding, and performance programming more broadly, represent distinctive kinds of user experience. The user experiences of performance programming can be analysed in terms of the distinct ways that performers and audiences use and interpret the representations that they see. As with Cognitive Dimensions of Notations, different types of experience result from the ways that information structures are represented and manipulated. In live coding contexts, these information structures must be mapped to the structural elements by which art works are constructed and interpreted. The resulting framework offers a basis for the analysis of live coding experiences, as well as for reflexive critical understanding of the new languages and tools that we create to support live coding.

Acknowledgments

Thank you to Thomas Green and Sally Fincher, for valuable review and contributions to the development of the Patterns of User Experience framework.

REFERENCES

Alexander, C. (1978). *The timeless way of building*. Oxford University Press.

Beck, K. and Cunningham, W. (1987). *Using pattern languages for object-oriented programs.* Tektronix, Inc. Technical Report No. CR-87-43, presented at OOPSLA-87 workshop on Specification and Design for Object-Oriented Programming. Available online at http://c2.com/doc/oopsla87.html (accessed 17 September 2009)

Blackwell, A.F. (2014). Palimpsest: A layered language for exploratory image processing. *Journal of Visual Languages and Computing* **25**(5), pp. 545-571.

Blackwell, A.F. (in press) A pattern language for the design of diagrams. In C. Richards (Ed), *Elements of Diagramming*. To be published June 2015 by Gower Publishing.

Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of the Psychology of the Programming Interest Group (PPIG 2005),* pp. 120-130.

Blackwell, A.F. & Fincher, S. (2010). PUX: Patterns of User Experience. *interactions* 17(2), 27-31.

Blackwell, A.F. & Green, T.R.G. (2000). A Cognitive Dimensions questionnaire optimised for users. In A.F. Blackwell & E. Bilotta (Eds.) *Proceedings of the Twelth Annual Meeting of the Psychology of Programming Interest Group* , 137-152

Blackwell, A.F. and Aaron, S. (2014). Take a little walk to the edge of town: A live-coded audiovisual mashup. Performance/presentation at CRASSH Conference Creativity, Circulation and Copyright: Sonic and Visual Media in the Digital Age. Centre for Research in the Arts, Social Sciences and Humanities, Cambridge, 28 March 2014.

Bresciani, S., Blackwell, A.F. and Eppler, M. (2008). A Collaborative Dimensions Framework: Understanding the mediating role of conceptual visualizations in collaborative knowledge work. *Proc. 41st Hawaii International Conference on System Sciences (HICCS 08),* pp. 180-189.

Church, L., Rothwell, N., Downie, M., deLahunta, S. and Blackwell, A.F. (2012). Sketching by programming in the Choreographic Language Agent. In *Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2012),* pp. 163-174.

Dagit, J., Lawrance, J., Neumann, C., Burnett, M. Metoyer, R. and Adams, S. Using cognitive dimensions: Advice from the trenches. *Journal of Visual Languages & Computing*, 17(4), 302-327.

Edge, D. and Blackwell, A.F. (2006). Correlates of the cognitive dimensions for tangible user interface. *Journal of Visual Languages and Computing*, 17(4), 366-394.

Gabriel, R.P. (1993). Habitability and piecemeal growth. *Journal of Object-Oriented Programming* (February 1993), pp. 9-14. Also published as Chapter 2 of *Patterns of Software: Tales from the Software Community*. Oxford University Press 1996.
Available online http://www.dreamsongs.com/Files/PatternsOfSoftware.pdf

Gamma, E. Helm, R. Johnson, R. and Vlissides, J. (1994). *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Gernand, B., Blackwell, A. and MacLeod, N. (2011). *Coded Chimera: Exploring relationships between sculptural form making and biological morphogenesis through computer modelling*. Crucible Network.

Green, T.R.G. (1989). Cognitive Dimensions of Notations. In L. M. E. A. Sutcliffe (Ed.), *People and Computers V.* Cambridge: Cambridge University Press.

Green, T.R.G. (1990). The cognitive dimension of viscosity: A sticky problem for HCI. In *Proceedings of the IFIP TC13 Third Interational Conference on Human-Computer Interaction (INTERACT '90),* pp. 79-86.

Green, T.R.G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7, 131-174.

Green, T.R.G., Blandford, A.E., Church, L., Roast, C.R., and Clarke, S. (2006). Cognitive dimensions: Achievements, new directions, and open questions. *Journal of Visual Languages & Computing*, 17(4), 328-365.

Holzner, S. (2006). *Design Patterns For Dummies*. Wiley.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall.

Lewis, J. (2014). Making the invisible visible: designing technology for nonliterate hunter-gatherers. In J. Leach and L. Wilson (Eds). *Subversion, Conversion, Development: Cross-Cultural Knowledge Exchange and the Politics of Design*, 127-152.

Tidwell, J. (2005). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly.

Coda Lisa: Collaborative Art in the Browser

Felienne Hermans
Delft University of Technology
f.f.j.hermans@tudelft.nl

Rico Huijbers
Amazon
huijbers@amazon.com

Abstract

This paper introduces Code Lisa: a collaborative programming environment in the browser that allows users to program one cell in a grid of cells. The cells can react to each other, but also to the environment, represented by sensor values on an EV3 Lego Mindstorms robot. By programming reactions to each other and to the sensors, a group of Coda Lisa users can together create a living, interactive art work. Users program the cells using JavaScript, and can experiment locally with their cell in a live editor created for this purpose, which shows a preview of the cell's behaviour. Once ready, the cell's program can be submitted to the server so it can be rendered in the shared grid of cells, by a second Coda Lisa client application: the viewer. In addition to the implementation of Coda Lisa, this paper also describes several plans for future development.

1. Introduction

In recent years, live programming has attracted the attention of many, in all sorts of different domains, from education [1] to music programming [4]. The liveness of programming languages and environments enables quick feedback, which in turn powers creativity. Modern web browsers make it easier than ever to support live coding, because they are sophisticated platforms for rendering and scripting.

Programming environments which enable users to create drawings or more elaborate artwork are common, starting in the eighties with the Logo programming language to modern day with sophisticated tools like Processing [5].

In this paper we propose Coda Lisa, a program that enables making art through collaborative live programming. Coda Lisa is a live coding environment that enables users to collaboratively create a living artwork. Code Lisa users control a single *cell* in a grid, where other cells are drawn by other Coda Lisa users. By reacting to other cells, as well as to the physical environment around the players by reading sensor values, cells can be programmed to behave interactively, and together form one artwork as shown in the Figure below.

This paper describes the implementation of Coda Lisa and proposes a number of future extensions to be developed.

2. Implementation

2.1. Editors

The central component of Coda Lisa is a canvas divided into a grid of cells, whose contents are controlled by the users. This is the view shown in Figure 1.

Coda Lisa users control their cell by implementing two JavaScript functions: an `init` function and a `draw` function. The `draw` function can be declared as taking 1 to 3 arguments which will be explained below.

In the most basic form, `draw` takes a `cell` argument that represents the cell that an agent program is about to draw into. Users write code that will set color values for the pixels in the cell. For example, a red circle can be created with this piece of code:

```
function draw(cell) {
    for (var y = 0; y < cell.h; y++) {
        for (var x = 0; x < cell.w; x++) {
            var color;

            if (dist(x, y, 50, 50) < 10)
                color = new Color(255, 0, 0);
            else
                color = new Color(255, 255, 255);

            cell.set(x, y, color);
        }
    }
}
```

When run locally, in a user's browser, the code runs in the central cell of a grid of 9 cells, as shown by the picture below:

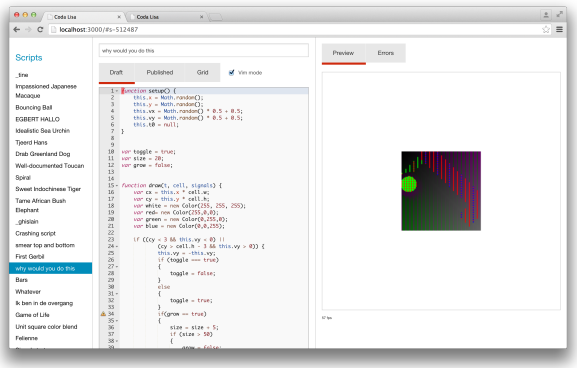


Figure 1: *The Coda Lisa editor: left is a list of all available programs, in the middle the JavaScript code can be entered, right, a preview of the cells behavior is shown*

The above interface is a Coda Lisa client application called the *editor* and is live: On every code change, the functions are compiled, and when compilation succeeds the `init` and draw functions on the data object are replaced, while the draw function keeps being executed at 30Hz.

To allow the user to have persistent data between live code reloads, we store data on a long-lived object that is outside the user code, and we call `init` as few times as possible. We use the prototypal facilities of Javascript to move the functions onto the object after they have been compiled.

`init` is in executed in the context of the data-holding object the first time the agent is run, and again on-demand when the user wants to change the `init` code. The context allows the user code to use `this.` to access long-lived variables.

When a user is satisfied with his cell's programming, she can submit the code to the Coda Lisa server. The execution of the user programs is then run by a second client application called the *viewer*, which executes all programs together on a big screen. This is when the user's program can start to interact with the cells created by other Coda Lisa users. When a user submits the code, the viewer calls the `init` method which is run once, and then calls the `draw` function at 30 Hz. The `draw` function has a *time* parameter that users can read (where time is given in fractional seconds), which allows users to create animated cells. For example, by changing the code of the previous example slightly, we create a red circle that moves left at a rate of 10 pixels per second:

```
function draw(cell, t) {
    for (...) {
        var x0 = modulo(50 - t * 10, cell.w);
```

```
        if (dist(x, y, x0, 50) < 10)
            color = new Color(255, 0, 0);

        // ...
    }
}
```

2.2. The environment

In addition to the simple variant of the draw function in which just time is used as a parameter, users can also have their cell react to the environment. The first environmental variable is the color values of the canvas in the last iteration. The `cell` object that is passed to the `draw` function has a `get` function, which will return the color of any pixel, relative to the cell's own coordinate system. By mixing past and future colors in the cell's own region, the program can create *blending* effects.

An agent cannot only write but also read pixels outside its own cell's boundaries, but an agent is not allowed to write outside its own cell. This reading outside the cell gives a user the opportunity to respond to the pixels in neighbouring cells. She can, for instance, create a *reflection* cell that mirrors a neighboring cell. To test their programs in this manner in the 3x3 grid in their local editor, the cells surrounding cells by default come pre-loaded with an agent program that responds to mouse events by painting pixels.

Finally, users can incorporate sensors values from the real world into their programs. For this, we have built a Lego EV3 Mindstorms robot with touch, color, sound and distance sensors. Using a simple request-response protocol over a serial connection to the EV3 brick, the sensor values are exposed to Coda Lisa programs in a third argument of the `draw` function.

For example, the following program draws a circle whose radius depends on the value of the distance sensors, with smooth transitions by blending the pixels of the previous frame:

```
function draw(cell, t, sensors) {
    for (...) {
        var color;

        if (dist(x, y, 50, 50) < sensors.dist * scale)
            color = new Color(255, 0, 0);
        else
            color = new Color(255, 255, 255);

        color = color.mix(cell.get(x, y), 0.7);

        cell.set(x, y, color);
    }
}
```

3. Related work

We took our inspiration for Coda Lisa from various existing systems. We for example looked at Processing, a programming language originally developed as a tool to teach kids to program, which evolved into a tool for artists and researchers [5]. Similar environments are Gibber [2], and Lich.js [3], both browser based live coding language for music and graphics.

Coda Lisa shares the idea of being a programming environment for art with these systems, but, while they do support some form of collaboration, they are not solely meant for it, like Coda Lisa is.

4. Conclusions & Future extensions

This paper describes a live programming environment in the browser, which lets users live program agents to control cells in a grid. The cells can vary depending on time and their previous color values, but can also react to colours of adjacent cells, or EV3 sensor values. In this way, Coda Lisa users collaboratively program a living, interactive artwork.

This paper presents a first version of Coda Lisa, which to date only has been demonstrated and used at one event. We foresee the following future extensions.

4.1. More environmental variables

A first extension would be to add more environmental variables in the form of *live data streams*, like stock tickers, weather data or seismographic information.

Furthermore, we could allow people to connect their mobile devices to the shared canvas, not only to act as an additional viewer, but also to send the devices sensor values, like gyroscopic sensors, to Coda Lisa, or to use the devices camera as input for a certain cell.

Finally, it would be great to extent Coda Lisa such that it can be played on a device with touch capabilities and have cells react to the screen being touched.

4.2. Better debugging and more directness in the editor

When users programmed their cells in the first Coda Lisa workshop, we noticed the need for more support in understanding the behaviour of their cells. What would greatly help here are features like the possibility for a user to *rewind* their animation and replay it slowly. When we ran the workshop, we noticed users mimicking this type of debugging by inserting `sleep` commands in their code, so this is something Coda Lisa should support. To make the replay more insightful, we could even highlight the current line of code being executed to improve understandability and make Coda Lisa more suitable for educational purposes.

4.3. A shared canvas

Finally, we would like to explore the possibility to, instead of dividing the canvas into cells, have all agents share the canvas. This increases opportunity for aesthetically pleasing compositions, as the grid structure leads to sharp edges in the final composition. However, this will also increases the complexity of the programming, both on the side of the user and on the Coda Lisa side.

References

[1] Code monster home page. http://www.crunchzilla.com/code-monster. Accessed: 2015-05-10.

[2] Gibber home page. http://gibber.mat.ucsb.edu/. Accessed: 2015-05-10.

[3] Lich home page. https://www.chromeexperiments.com/experiment/lichjs. Accessed: 2015-05-10.

[4] Pam Burnard, Nick Brown, Franziska Florack, Louis Major, Zsolt Lavicza, and Alan Blackwell. *Processing: A Programming Handbook for Visual Designers.* http://static1.squarespace.com/static/5433e132e4b0bc91614894be/t/5465e778e4b02ea3469103b0/1415964536482/research_report_dc_02.pdf, 2015.

[5] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers.* MIT Press, 2014.

Live Coding the computer as part of a free improvisation orchestra of acoustic instruments

Antonio José Homsi Goulart
Computer Science Department
Institute of Mathematics and Statistics
NuSom – Sonology Research Centre
University of São Paulo – USP
antonio.goulart@usp.br

Miguel Diaz Antar
Music Department
School of Communication and Arts
NuSom – Sonology Research Centre
University of São Paulo – USP
miguel.antar@usp.br

ABSTRACT

In this paper we present our experience of having a live coder amongst acoustic musicians in a free improvisation orchestra. The acoustic musicians in the orchestra had no previous experience with live coding. We will discuss all the difficulties we experienced and the process for overcoming them, illustrating our observations with audio excerpts from some of our recorded sessions. We will also go through a discussion about the nature of the instruments and raise the question of how deeply the musicians should understand code in order to effectively interact in the free improvisation context. Finally we propose a piece for live coder and orchestra that artificially explores the different dimensions of an orchestra session.

1. INTRODUCTION

Both authors of this paper are members of a free improvisation orchestra[1]; Antonio is the live coder and Miguel is the bass player. For 3 years we have been collecting thoughts, sessions records, performances videos, and discussions related to the art of programming audio on the fly amongst other musicians using classical acoustic instruments. In this paper we share these ideas and facts and some of our theories. We discuss some exercises we use in order to develop a good interaction channel between the musicians and we also propose a piece for live coder and orchestra in which we elaborate on a possibility to tackle the challenges in such a scenario.

Albeit sharing some concerns like participation, communication within the orchestra and with the audience, our approach is different from those of other works relating improvisation and laptop ensembles or laptops in ensembles (Burns 2012, Lee and Freeman 2014, Ogborn 2014). Although some of these groups incorporate acoustic instruments occasionally, our orchestra is not based on laptops, but only features one live coder joining instrumentalists playing alto and baritone saxophones, trombone, didgeridoo, flute + effects, double bass, prepared piano, and voice + everyday objects.

Wilson et al. (2014, 2) talk about the role of "intervention mechanisms to facilitate aspects of musical interaction that can be difficult to realize in strictly free performances". They use networked music systems for the exchange of information in a context of a laptop orchestra. But in our case we are using live coding as a stand-alone instrument, and we are concerned about how the orchestra musicians can interact better, understanding the live coder issues and possibilities. We think that by sharing what we have observed and experimented with we can help and also hear back from other groups working in a similar context.

[1]Orquestra Errante (OE) is an experimental group based on the Music Department of the School of Communication and Arts at University of São Paulo, founded in 2009 as a laboratory to the practice of free improvisation and reflections about its theories and territories. OE membership is seasonal and variant, usually composed by students and professors from the Music Department, with some sporadic students from other departments. OE is also open to occasionally hosting visitor musicians from outside the academy. In our weekly meetings we practice free improvisation sessions and also oriented exercises, with a round of discussions after each piece. We keep crude records of our improvisations so our members can access them for later reasoning and reflections for their research. Some samples of our work, which we refer to later in this paper, can be found at *soundcloud.com/orquestraerrante*

OE members' background comes from classical music, and some also studied jazz. They have been through a lot of instrument practice and conventional ear training, but only recently are also starting to explore audio effects and electronics as part of their set of tools, and getting used to electro-acoustic / electronic / computer music repertoire and possibilities. In the first year of OE with live coding a common fact that we were observing in our sessions was that most times the live coded sound acted as a set of background layers with little influence on the acoustic musicians, while the live coder could not find a way to interact either. However, by the time the group understood better the nature and range of live coding, and the live coder got better at his activity, the group started to evolve towards a better interaction and wholeness.

In the remainder of the article we will go through some thinking about free improvisation in Section 2, the differences about the instruments involved in our group and their manipulation in Section 3, and then in Section 4 we present the interaction issues we went through in our sessions. In Section 5 we describe an interesting idea for a performance we developed, before concluding the text.

2. FREE IMPROVISATION (FI)

As a musical performance FI in a group presupposes an intentional collective and collaborative activity. Involved musicians must, through sound, start a conversation which evolves in an uninhibited manner, and each performer intervention simultaneously creates, modifies and draws the way (Costa 2003, 42)[2]; as in a spontaneous conversation, the subjects and their unfoldings are diverse. This metaphor highlights one of the major characteristics of FI, that of the creator / interpreter role of each participant. Within an FI session the unraveling of a performance is dictated merely by the interactions between them, in a sound flow shaped along the *devir* of the act. Free improvisers play a musical ludic game not expecting the victory, but individually proposing ideas that may or may not be corroborated by the collective (Falleiros 2012, 59)[3]. Costa (2007, 143) argues that "There is a presupposition that everything is impermanent and the forms are provisional aspects of intermediations made feasible by unexpected and rhizomatic connections".

Another aspect of FI is the deliberate act of trying to overcome established musical idioms in search of a "non-idiomatic" improvisation (Bailey 1993, xii), and an openness to the possibility of using any sound in the discourse. In such a way free improvisers are always looking for different ways and techniques to extend their instruments, many times during the course of a performance. In such a way we can neither know nor expect anything from a FI session. There is the potential for absolutely anything regarding sonorities and interactions. It is a collaborative (when in a group) experimentation process, radically striving towards real time music creation. Usually there is no previous indication about the musical content, but FI is also open to minimal scripts with any type of instructions that can be revealed in advance or when the session is about to start.

Previous information about the other performers is not a prerequisite for 2 or more people to free improvise together. Musicians should be able to interact based only on the sound they create and hear, not needing to know anything about each other's instruments' sounds and way of playing (we address that problem later). For instance, the bass player should be able to jam with the drummer even not knowing anything about drum sticks or the relation between the different drums and their timbres. In such a way an active hearing is a crucial aspect in FI; Costa (2007, 143) argues about the necessity of a Schaefferian reduced listening, abstracting "the sound sources, the musical meanings and the inevitable idiomatic gestures present in each musician's enunciations". In other words, when free improvising we should focus our attention to the sonorous aspects of the instrumental actions.

Falleiros argues that along with this active listening, a promptness for a benevolent interaction is also a must, where "(...) improvisers are distant from manipulative or polarizing actions, (...) which drive the listening in a previous direction that presents an established way to deal with". This benevolent intention resembles a chamber music orchestra attitude, in which "the first step to learn to correctly execute chamber music is learning not to exhibit oneself, but to recede. The whole is not constituted by an auto-affirmation of the individual voices" (Adorno 2009, 190, our translation). The same goes for FI, where this chamberistic posture (courtoisie[4]) permeates through the intermediation of the sound fragments and enunciations.

[2]All translations from Costa were made by us from the original in Portuguese.
[3]All translations from Falleiros were made by us from the original in Portuguese.

3. MANIPULATING MUSICAL INSTRUMENTS

Some theories characterize the musical instrument as an extension of the body (Ihde 1979; ___ 1980 apud Magnusson 2009, 1), assigning to the musician the role of improving this condition up to the point of being able to fluently express musical intentions, or as put by Nijs (2009, 2), "(...) the musician no longer experiences a boundary between himself and the instrument". In the context of improvised music this ability is crucial, as usually there is no previous structure or composition to guide the interaction between the musicians and guarantee coherence in the music created, which depend then on the performers' sensibility and dexterity.

The body is fundamental in a music based on sounds and empirical gestures on the instruments. The FI process is radically related to body and gesture, leaving no room for separation between body and mind (Costa 2003, 94-95), so there is an inherent physicality in the musical practice which varies according to the musician and the instrument, but regardless of both, conveys a sensation of immediacy from physical stimulus to sound response. So, in other words, there is a body responsible for creating immediate sounds using an acoustic instrument as an extension of the body.

Programming the computer on-the-fly, however, the relationship between musician and instrument changes from a physical-mechanical (body-instrument) to a physical-abstract (body-code) one. Notice that although we agree with Magnusson (2009, 1) when he interprets the computer instrument as a mind extension, we still put it as an extension of the body, in part because the musician should write code, an action which takes time on stage, and also because in an orchestra context the body is still there as part of the group and as visual information to the other performers and audience.

Compared to acoustic instruments, it takes more time to realize a musical intention in an algorithm (which then outputs sound). This lag comes both from the time needed to elaborate the algorithm and the actual writing (sometimes done together), and regardless how fast a performer is in typing and how literate about the techniques (Nilson 2007, 4), unless snippets with basic blocks and specific sounds are used, the implementation of a musical intention is not immediate. However, albeit imposing immediacy restrictions on the performance, live coding enables the performer to implement gestures for the future. As pointed out by Stowell and McLean (2013, 3), "a live coder, like any improvising performer, is under pressure to do something interesting in the moment. Live coders can use abstraction and scheduling so the notion of 'in the moment' may be a little different to that for more traditional instruments improvisers".

Besides that there are the vast possibilities offered by live coding, compared to acoustic instruments or fixed graphical user interface based computer instruments (Blackwell and Collins 2005, 6). Theoretically any sound can be generated with the different synthesis techniques and audio effects, and the musical decisions are only limited by the coder ability and creativity (and also the pressure due to time lags).

Finally there is the projection issue. TOPLAP manifesto[5] asks to "show your screens", but projecting the code to the audience is not unanimously seen as a good choice (McLean et al. 2010, 1). In our case the live coder is part of an orchestra, so at first we decided not to project because we thought that it would bring most of the attention to the live coder and his code. Our usual audience is more used to acoustic instruments and as a result of this the projection would have a boosted shock factor (Collins 2011, 3) which could in turn blur their attention to the group music and performance. But then we tried the projection at one of our practice meetings, and it makes more sense to talk about that in the next section.

4. INTERACTION IN PRACTICE AND SOME OF OUR EXERCISES

The same way that the bass player and the drummer can jam without staring each other, should they be able to jam with the live coder without reading his code? Does reading his future processes contribute to the improvisation, or does it give too much information in advance?

---

[4]French word for courtesy, politeness, civility, which appears in Adorno's text two paragraphs before the excerpt we cited.

[5]http://toplap.org/wiki/ManifestoDraft

We observed that in our group the complex relations established at the FI process are made of the physical, corporal and emotional dimensions from the instrument gestures, and the musicians' attention fluctuates between the sonorous and the visual.

We understand that the reunion of a live coder with musicians playing acoustic instruments offers a rich context for the creation of very interesting improvised music, being up to everyone to understand everyone's potentials and limitations. Regarding live coding, as put by Wilson et al. (2014, 1), "On the one hand, it becomes possible to do many previously unimaginable things; on the other, it is often not possible to do them very quickly".

At first OE musicians did not seem to understand these lag issues regarding programming audio on-the-fly. The tendency in our performances was to observe conversations between the acoustic instruments, while the live coded sound remained as a background layer without much interaction, most times only influencing and being influenced by the orchestra in the long term. An example of this background layer can be observed from 00:58 up to 02:30 in the piece[6] "Estados, eventos e transformações", where the high-pitched sounds from live coding were not influencing the acoustic musicians. However, the same sound pattern explored with a faster tempo drives at 02:35 the recovery from a sort of break and triggers a beautiful movement from the piano player at 02:55.

One of the tools we use to articulate the flow is to explore moments of duos or trios, either in a spontaneous fashion or previously settling the groups and when / how they should take place. For instance, we always warm up for a session making a circle and practicing interactions in duos. Two musicians start jamming, when appropriated the next one on the circle enters in the interaction and they improvise as a trio for a while, then the one who was in the trio for most time leaves, the remaining duo improvise for a while, and so it goes until everyone jammed. Each duo would typically last for about 2 to 3 minutes. Another problem we observed was the extreme difficulty for the live coder to interact in these short moments of duos. This difficulty can be observed in the example "Transição de duos – Set2014", where duos transitions were explored. Notice that good interactions happen up to 05:00, when starts the live coder and flautist turn and they struggle to contribute and support each other. The only attempt to interact was based on imitating attacks up to 06:16 (interesting musical output, though), when the live coder recedes in order to prepare something for the wrap up of the exercise and the flute continues. Lack of time might be pointed as the main obstacle here, forcing a tendency of using snippets or coding simple percussion or melody lines with the basic waveforms. But it also seems to exist in almost everyone a kind of anxiety to make a remarkable movement in the current turn, and we think that within this scenario the discourses are unlikely to converge.

Although we eventually made good performances[7], it was clear to us that more information about the live coding process and computer music techniques were lacking for OE members. Up to that point our coder was using SuperCollider (mainly the Patterns library (Kuivila 2011) and some simple one-liners), so we talked about some synthesis techniques, how long takes in average to code them as SynthDefs, what can be done with the Patterns after the synthesis definition is coded, and how huge turnarounds in output can be obtained with small twists in code. Finally, we made our practice session projecting the code (half the orchestra have not seen live coding before), and they got a better idea about the live coder different way of playing, like, as beautifully put by McLean (2008), "In contrast to musical instruments directly plucked, struck and bowed by a human performer, the movements of a live coded performance all happen inside a computer. The live coder may be typing furiously, but it is the movement of control flow and data inside the computer that creates the resonances, dissonances and structure of music".

In the session "at MusicaNova 2014", which we considered overall successful, a duo between the live coder and the trombone player starts at 04:20. The first one struggles to make something interesting, and while only a slow tempo rhythmic tone is his actual sound, the latter makes his best with a nice sewing around the live coder poor sound, carrying the duo for a while. By the time the live coder finally fixes the code and get what he was trying to, the trombone player is immediately influenced and a nice interaction and new sonority are obtained by the duo, which then decides to continue exploring it until 07:20, when the bass

[6]All pieces that we use as example can be found at *soundcloud.com/orquestraerrante*

[7]The second half of the piece "Buscando chaves" is in our opinion an example of good interaction between all the performers (everyone active listening everyone's musical ideas and supporting each other) and interesting musical result (contrasts, nuances, resolutions of tensioning, *crescendos* and *diminuendos*).

player comes in. Another interesting moment in this piece (from the live coding point of view) happens from 25:40, with a buzz that morphs into impulses, which increases in density and drives a nice conversation between all instrumentalists. This is then followed by a sort of break with the coded clicks and acoustic textures in *piano*. At 27:35 there is a huge drop in the coded sound where only a regular click is left, and the musicians unanimously stop for a while and then come back for another *tutti*, showing perfect interaction in this long sequence of tension and release. Also in the performance "at MusicaNova 2014", there is a very intense *tutti* passage at 31:06 where the live coder does achieve interaction, influencing and being influenced by the acoustic musicians' quick movements and textures transitions.

And now we come back to the questions posted in the beginning of this section, with another question. OE members do not know programming, and even if they knew, how feasible would it be to simultaneously perform their instruments, explore extended techniques, active listen to everything, and also read code?

We adapt the free improvisation presupposition and state that it is possible to interact only through sound, given that everyone have and idea about everyone's instrument nature. People playing acoustic instruments should know that in live coding (usually) there is not a direct relationship between typing and sound output, and writing sentences (the live coder gesture) - takes time. With that, the acoustic musicians can support the live coder lots of different ways while an algorithm is being written or bugs being fixed. Instrumentalists should also keep in mind that once the initial algorithm is written, with little time and typing lots of variations and musical intentions can be achieved and explored collaboratively.

However, if the musicians (except the live coder) in the orchestra are able to read code and understand where the live coder is sonorously going, interaction with the future is enabled and the musicians can also experiment making room (or creating tension) for the coded sounds that are to come (the resolution). If only one or a few of the members can read code they can still make this bridge to the future (and if their musical movements are convincing they can lead the way in that bridge), but the pitfall in this case would be an unconscious process of paying too much attention to this one aspect, "overlistening" to the other members, ruining the lack of differentiation between the musicians in a free improvisation scenario. Maybe (?) an interesting scenario for an acoustic orchestra with a live coder would be if everyone could read code and interact with the future, also being able to, preferably via sounds, warn the live coder about tortuous ways.

Of course not every free improvisation orchestra with acoustic instruments members might want to learn programming and digital signal processing algorithms. But the same way that the live coder should know something about his fellow musicians' possibilities and limitations, like a composer when choosing specific instruments for writing pieces, it is extremely important that all orchestra members are at least exposed to the process of live coding, so they can be aware of its specificities.

5. DEVIR (FOR LIVE CODER AND GHOST ORCHESTRA)

In this piece we wanted to explore the relations between the space where the improvisation happens, the visual exchanges between the musicians, the interactions based on the immersion on the sounds and the different possibilities of a musical discourse with a specific material.

As we observed and mentioned before, the sonorous is often blurred by the visual in collective musical practices. We also discussed about the live coder's different types of gestures and peculiarities that the group should take into account when interacting. So we wanted to artificially create a context in which everyone is forced to interact with the live coder, actively listen to the live coded sound, and where the live coder actually tries to suggest how the individual aspects of the resultant orchestra music will be imagined.

In Devir, the musicians are divided in duos which are placed acoustically and visually isolated. The live coder and audience are placed in yet another room, where the code is projected for the audience and the live coder controls a mixing board into which a mono signal from each duo and the stereo signal from the computer are connected. The live coded audio is also sent to one musician from each duo through headphones.

In the performance, the live coder controls the volume of each duo and the live coded sound in the final mix, the one that is reproduced in the room with the programmer and audience. The live coder also controls the volume of the computer sound that one musician from each duo will receive in the headphones. So the live

coder is faced with the challenge of coming up with a live coded sound that at the same time is interesting for the final mix and suggests directions for one musician in each duo, which then interacts with the coded sound, guiding the interaction within their partners within the duos.

In some moments the live coder might find it interesting to use only the computer sound in the final mix, or use it in conjunction with only one duo (for instance to emulate a trio even when the full orchestra is playing) or a few of the duos sounds (to artificially leave out of the performance a part of the orchestra for a while). Another interesting possibility is using only the duos sounds in the final mix, individually suggesting directions for each duo (voiding the volume of the headphones for all but one duo at a time).

Interestingly, in this piece, the live coder role is actually to live code both his algorithm, the entire orchestra, and also the final mix and musical result of the session. Of course, depending on the technical structure of the venue some variations for what was proposed can be tested, for instance isolating all the musicians and individually communicating with them, or also to try a scenario with more than one live coder, perhaps even one for each acoustic musician, and the live coders talk to themselves like a clan playing first-person shooter video games[8] (either shouting in the same room or with headphone sets over the network).

Aside from the complexity regarding the necessary hardware, we don't suggest projecting the code for the duos because we want the musicians to know that the computer musician is live coding but we do not want them to have to be able to read the code, as in a regular setup they wont have time for it, or many would not even be able to, like we considered earlier. But occasionally, if conditions permits and some musicians can read code, another variation would be to project the code to one musician in each duo and send audio over headphones to the other one, creating another dimension within each duo. Another more extreme variation would be to isolate everyone and send audio to half of them and projection to the other half, or even only project the code to all isolated musicians and mix their individual sounds of interaction with the *solfeged* code.

6. CONCLUSIONS

In this paper we considered the collision of two extreme musical practices in an acoustic orchestra context. In free improvisation musicians' enunciations do not go through any kind of filter. This is an intentional liberation from music traditions. The only limits regarding sonorities and musical conversations are the performers' creativity and dexterity.

Above all the difficulty involved in finding a definition for live coding, there is an agreement that it "(...) involves the writing of algorithms in real time" or, concatenating another two excerpts by Magnusson (2014), is a "method of composing through performance in real time", involving "deep exploration of the medium at hand". It is striking how this last description is intimately related to broader definitions of free improvisation (Falleiros 2012, 18). And we agree with Wilson et al. (2014, 1) when they say that "live coding might be seen as an improvisational interface par excellence but lacks in agility".

Given all the difficulties we went through and the differences between acoustic instruments and code as an instrument, we were at first wondering if live coding was adequate as part of a primarily acoustic FI orchestra. We think it is, but important concepts must be worked by all the members to enable fluency in conversations within the flow of a session. It is desirable that the acoustic musicians new to live coding attend some performances or watch good videos, preferably accompanied by the orchestra coder. They should have an idea about the processes of writing, debugging, adapting and reusing code so they can tell what is going on in interaction situations when improvising with the live coder.

We also considered the necessity of coding/DSP knowledge by the acoustic musicians. Of course that it would enable them to come up with interesting anticipations while improvising, but we do not think it is a need, and we should not forget that FI presupposes interaction based solely on the sound up to the current moment.

Up to this point we still have not experimented with incorporating in the live coding possible rhythmic information that might arise in an improvisation. Although rhythm patterns are usually avoided or

[8]http://en.wikipedia.org/wiki/First-person_shooter

disguised in FI practices, it could be interesting to deal with rhythm synchronization issues between the acoustic and computer sounds.

The possibility of interacting with the future would not be a stray from Bailey's (1990, 95) "playing without memory" paradigm. But do we want them to read code? Agreeing with McLean and Wiggins (2010, 2) when they talk about the code written by live coders, "(...) Their code is not their work, but a high level description of how to make their work". In free improvisation there is a priority relationship with sound, and not respecting that would be going too far from the free improvisation paradigm.

No, we do not want them to need to read code. And programs sometimes crash. But that is not a problem if we have programmers playing acoustic instruments in Orquestra Errante. The interaction between all the performers should be based on the sounds at the current instant allied with a benevolent attitude and an openness to the future, in an expectation projected through code *solfege*, not imprisoned by its abstract interpretation, but derived from its sonorous concreteness.

Acknowledgments

We would like to thank all members of Orquestra Errante for all the practices and discussions over the last 3 years. Our work was partially funded by NuSom (Sonology Research Center at USP) and CAPES (proc. Number 8868-14-0).

REFERENCES

Adorno, Theodor. 2009. *Introdução à sociologia da música*. São Paulo: Fundação Editora da UNESP.

Bailey, Derek. 1992. *Improvisation: Its nature and practice in music*. England: Moorland Pub.

Blackwell, Alan and Nick Collins. 2005. "The programming language as a musical instrument". In: Proceedings of the PPIG05 (Psychology of Programming Interest Group).

Burns, Christopher. 2012. "Adapting electroacoustic repertoire for laptop ensemble". In: Proceedings of the 1st Symposium on Laptop Ensembles and Orchestras, 76-79.

Collins, Nick. 2011. "Live coding of consequence". Leonardo 44 (3): 207-211.

Costa, Rogério. 2003. *O músico enquanto meio e os territórios da Livre Improvisação*. PhD thesis, Pontifícia Universidade Católica, São Paulo.

_____. 2007. "Livre Improvisação e pensamento musical em ação: novas perspectivas". In: Notas Atos Gestos, edited by Silvio Ferraz, 143-178, Rio de Janeiro.

Falleiros, Manuel. 2012. *Palavras sem discurso: Estratégias criativas na Livre Improvisação*. PhD thesis, Universidade de São Paulo, São Paulo.

Kuivila, Ron. 2011. "Events and patterns". In: The SuperCollider Book, edited by Scott Wilson, David Cottle and Nick Collins, 179-206. MA: MIT Press.

Lee, Snag Wong and Jason Freeman. 2014. "Real-time music notation in mixed laptop-acoustic ensembles". Computer Music Journal, 37 (4): 24-36.

Magnusson, Thor. 2009. "Of epistemic tools: musical instruments as cognitive extensions". Organised Sound 14 (2): 168-176.

_____. 2014. "Herding cats: Observing live coding in the wild". Computer Music Jornal 38 (1): 8-16.

McLean, Alex. 2008 . "Live coding for free". In: FLOSS+Art, edited by A. Mansoux and M. de Valk, 224-231. London: Open Mute.

McLean, Alex and Geraint Wiggins. 2010. "Live coding towards computational creativity". In: Proceedings of the International Conference on Computational Creativity, 175-179.

Mclean, Alex, Dave Griffiths, Nick Collins and Geraint Wiggins. 2010. "Visualisation of live code". In: Proceedings of the EVA'10 (2010 International Conference on Electronic Visualization and the Arts), 26-30.

Nijs, Lunc, Micheline Lesaffre and Marc Leman. 2009. “The musical instrument as a natural extension of the musician”. In: Proceedings of the 5th Conference on Interdisciplinary Musicology.

Nilson, Click. 2007. “Live coding practice”. In: Proceedings of NIME'07 (7th International Conference on New Interfaces for Musical Expression), 112-117.

Ogborn, David. 2014. “Live coding in a scalable, participatory laptop orchestra”. Computer Music Journal, 38 (1): 17-30.

Stowell, Dan and Alex McLean. 2013. “Live Music-Making: A Rich Open Task Requires a Rich Open Interface”. In: Music and Human-Computer Interaction, edited by Simon Holland, Katie Wilkie, Paul Mulholland and Allan Seago, 139-152. London: Springer.

Wilson, Scott, Norah Lorway, Rosalyn Coull, Konstantinos Vasilakos, and Tim Moyers. 2014. “Free as in BEER: Some explorations into structured improvisation using networked live-coding systems”. Computer Music Journal, 38 (1): 54-64.

Embodiment of code

Marije Baalman
nescivi
marije@nescivi.nl

ABSTRACT

The code we read on the screens of a livecoding performance is an expression of our compositional concepts. In this paper I reflect on the practice of livecoding from the concept of embodiment as proposed by Varela, Thompson, and Rosch (1991): how we as coders embody the code, how machines can embody code, and how we inevitably need to deal with the physicality of the machines and our own bodies that we use in our performances, and how we adapt both our machines and bodies to enable our performances.

1. Introduction

This paper is an expansion of the author's thoughts on the topic of embodiment in the context of livecoding, following the invited talk at the *Livecoding and the Body* symposium in July 2014. As such it is a starting point for further research and reflection on the practice of livecoding and the role of our bodies in it.

Varela, Thompson, and Rosch (1991) defined *embodied* as follows (pages 172-173):

> "By using the term embodied we mean to highlight two points: first that cognition depends upon the kinds of experience that come from having a body with various sensorimotor capacities, and second, that these individual sensorimotor capacities are themselves embedded in a more encompassing biological, psychological and cultural context."

In the context of livecoding as a performance practice, this concept of embodiment surfaces in various ways - the physical interaction of our body (sensorimotor interactions of typing, viewing what the screen shows us) with the machine (its interface to receive input and giving output, but also the hardware architecture that defines the body on which code is interpreted), as well as how the grammar and structure of a programming language shapes our thinking about concepts and algorithms, and in turn the development of a programming language is determined both by the hardware on which it will run, as by our desire to express certain concepts. Similarly, Hayles (2012) writes in her book *"How we think"*:

> "Conceptualization is intimately tied in with implementation, design decisions often have theoretical consequences, algorithms embody reasoning, and navigation carries interpretive weight, so the humanities scholar, graphic designer, and programmer work best when they are in continuous and respectful communication with one another."

Where she suggests a close collaboration between humanities scholar, designer and programmer - within the livecoding scene, many performers write (and share) their own tools, i.e. they are bridging the artistic and the technological within their own practice, and actively contribute to the context within which they work; shaping their enviromenment at the same time as they are navigating through it. This refers back to both the second part of the definition of *embodied* of Varela, Thompson, and Rosch (1991).

In this paper I will first reflect on how the properties of a programming language influences our thinking and expression in code, then I will look at how our computers can embody different aspects of code, before discussing the physicality of the machine and how our code not only has its intended effects on the output media, but also on the machine itself. In the last section, I go into the interaction loop between the coder and the machine, and how we attempt to optimize this interaction loop.

2. Code embodied by the human

The practice of livecoding music has connections to both algorithmic music, and its expression in computer music, as well as to what in dance is termed *instant composition*: (Collective 2015):

> "*INSTANT COMPOSITION* combines the notion of working from the moment (INSTANTaneous creation) with the intention to build something (COMPOSING a piece with an audience present). This means that for us, improvisation principles are always concerned with both the question of FREEDOM and the question of STRUCTURE."

We express our concepts for musical structure in code - and as livecoders we do this at the moment of performance, rather than completely in advance - we change the algorithm, while it is unfolding its structure. Where in improvisational dance or improvised music, the structure of the composition is not apparent to the viewer or listener - in livecoding - when the screen is projected for the audience to see - the improvised compositional structure is made apparent, although it may be changed again by the livecoder before it is actually heard.

In our first encounters with a programming language as a coder, we try to adapt our minds to understand the language and develop our own methods and 'ways of doing things' within the language. That is, we create our own subset of the language that we use in our daily conversations. If we find we are dissatisfied with what we can express, we either look for another existing programming language that fits our needs, or we write our own (extensions of a) language to enable the expressions that we want. At the same time, the longer we spend programming in a particular language, the more our mind gets attuned to the language and shapes our thinking about concepts to express. Not only our mind gets attuned, also our body - as we type certain class names, our fingers can become attuned to particular sequences of letters that form words that have meaning in the code. As such code is an embodiment of our concepts (Varela, Thompson, and Rosch 1991), which develops itself in a dialogue between ourselves and the coding language.

(Rohrhuber, Hall, and De Campo 2011) argue that adaptations of a language (in their discussion the "systems within systems" of SuperCollider) are "less like engineering tasks than works of literature, formal science, and conceptual and performative art", and they place this practice within a social context of sharing code and such adaptations with other coders. Thus, code is not only the embodiment of concepts in our own mind, but they co-evolve with concepts of others, which feed into new adaptations of code and programming languages, which may lead to new concepts to express, or musical composition to be made.

On a social scale, in a comment made in a conversation at the SuperCollider Symposium in London in 2012, Scott Wilson commented how he enjoys the community around the programming language, in that even though the composers/musicians within the community have vastly different styles of music they create, they share a common mindset. This may well be because either the programming language attracts people with a certain mindset, or using the programming language pushes towards a certain way of thinking - or perhaps a combination of these.

3. Code embodied by the machine

In his description of the B1700 architecture, Wilner (1972) starts out describing how programmers using Von Neumann-derived machines are forced 'to lie on many procrustean beds'. He positions the motivation for the architecture of the Burroughs computers against the rigidity of the Von Neumann-derived architecture. In contrast the Burroughs machines are designed based on the requirements of the programming languages that should run on them, and the kinds of tasks that these programs need to perform. This resulted in features of the hardware that even today are still unheard of, such as variable bit-depth addressable memory and user-definable instructions in the instruction set of the B6000 machines (Mayer 1982). While in their first machines, the architecture was designed specifically for the programming languages ALGOL and COBOL, in the design for the B1700 they designed an architecture that would be able to adapt to a variety of programming languages through the use of emulators that they called *S-machines* which could interpret application programs. The hardware of the Burroughs machines was more complex than the contemporary machines of competitors, but it embodied many routines that would otherwise have needed to be handled by software routines (Mayer 1982). It made for a very close relationship between the code written in a high level language and the resulting machine code, considerably reducing the time to design and run these programs. Far ahead of their times, the Burroughs machines are an outstanding example of the embodiment of code on the side of the machine.

For specific purposes, such as graphics calculations or digital signal processing (DSP), there are specialised computation units that have been designed with these applications in mind: these DSP CPU's have an architecture that allows them

to do the same mathematical operation on a vector of data points at the same time, thus speeding up computations on large data sets.

At the other end, for the Von Neumann derived architectures, programming languages have been "stretched on procrustean beds", that is they have been optimized to run on the hardware architecture, favouring specific methods that fit the architecture (e.g. byte-oriented coding, the development from 8-bit, to 16-bit, to 32-bit to 64-bit machines) over ones that fit the application (e.g. working within the decimal system like one of the Burroughs machines (Mayer 1982)). In those parts where code needs to be optimised to run fast on specific hardware, certain programming strategies are preferred over others, and have shaped common practices of coding based on the hardware properties.

Most modern programming languages, and notably the programming languages used by livecoders, are further and further away from the hardware level; given the fast processors and the generous memory space we have today, speed and memory constraints are less of an issue in the design of programming languages, and the languages can be geared more towards particular ways of expressing algorithms. The interpreter or compiler takes care of translating these into machine code to run on the hardware, and in most cases the programmer in a high level language does not need to be aware of the underlying hardware.

4. The physicality of the machine

Having built-in the possibility of extending the instruction set, the designers of Burroughs were at the same time aware of security risks, so they built in a flag into the storage of data that would determine whether the word could change the instruction set or not; only programs that were part of the operating system were allowed to make this change (Mayer 1982). Also with the Von Neumann-derived machines, the now common (modified) Harvard architecture (Wikipedia 2015), makes livecoding — seen from the machine's point of view — impossible: we cannot change the program instructions at the very momemt they are running. We can just manipulate code that is compiled on-the-fly or interpreted into a sequence of machine instructions that the machine knows. But once it is in the pipeline towards the CPU (or in its cache), we cannot interfer anymore. We cannot change the machine, we can just change the commands that we put into the cache. Only if we interfer with the hardware of the machine itself, we can make a change. In Jonathan Reus' *"iMac Music"*, this is exactly what he is doing, while the machine is executing software, the performer hacks into the circuitry, changing the pathways of the electronics and distorting the images and sounds created by the iMacs (Reus 2012).

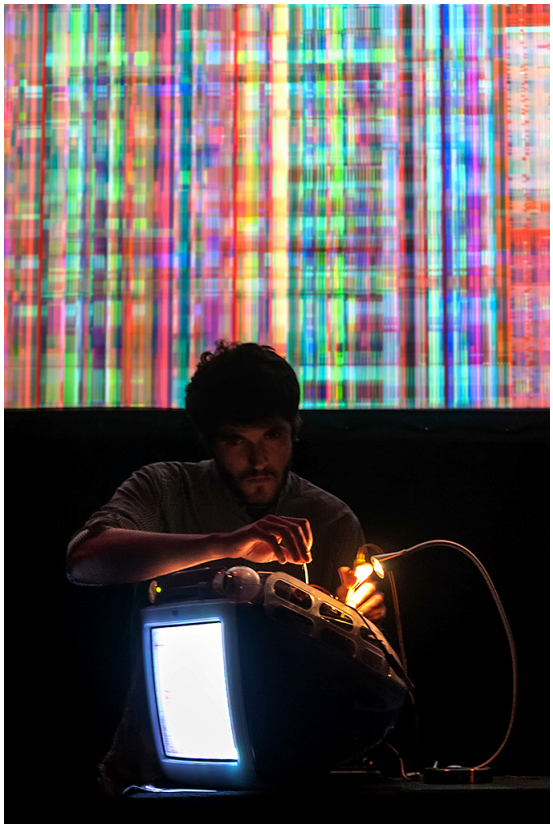


Figure 1: *Jonathan Reus performing "iMac music", photo by Ed Jansen.*

The instruction sets of computers consists very basic elements, methods of fetching data from memory, manipulating these bytes of data (negation, bit shifting, etc) and combining of data (addition, subtraction, multiplication, etc). What we call code is for the machine just data, through which we manipulate the memory map that determines how we move through this data. The compiler or interpreter translates our data to machine instructions that will determine these movements.

Griffiths (2010) with his project *"BetaBlocker"* created a virtual "8 bit processor with 256 bytes of memory" that can only play music, with a small instruction set. In his perfomances the state of the machine is visualised, as you listen to the sound that is created; in that *"BetaBlocker"* exposes the (virtual) machine completely, and the livecoding is happening close to the physicality of the machine.

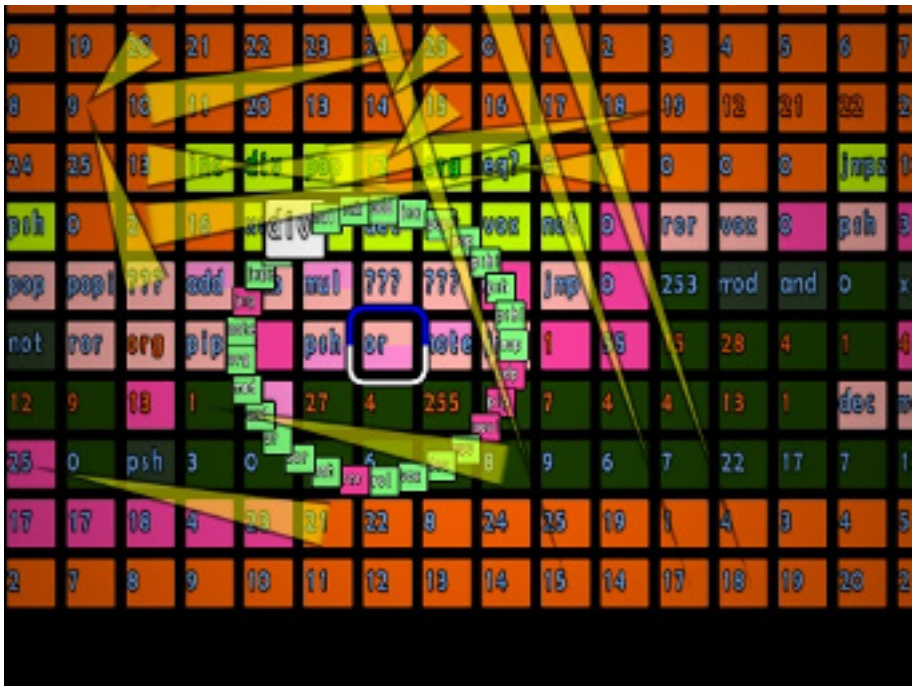


Figure 2: *Dave Griffiths' BetaBlocker (image from McLean et al. (2010)).*

The data that flows within the computer can come from and go to different sources; the data is communicated to the CPU through various buses. The processing of this data has both intended and unintended effects. If we write an algorithm for sound synthesis, the data we process is intended to go to the audio card and be transformed into sound waves. But at the same time when the CPU is very busy, it heats up, and safety mechanisms are in place to ensure that as soon as a high temperature is measured, the fan within the computer turns itself on. Sometimes the effect can be both intended and unintended, e.g. when accessing the harddisk (we intend to get data from the harddisk, but we do not necessarily intend it to go spinning), or refreshing the screen (we want to see an updated text, but in order to do so we regularly refresh the screen). Both of these things cause alternating electromagnetic fields around our computers, which we do not necessarily intend.

Following Nicholas Collins' circuit sniffing method (N. Collins 2006) using electromagnetic pickups to look for the sound inside the machine, (Reus 2011) is transforming these sounds and manipulates them with code, and thus causing new sounds to be created by the machine, in his work *"Laptop Music"*.

In both *"iMac Music"* and *"Laptop Music"*, Reus draws attention to the physicality of the machine, and how the code that runs on it can be affected by hacking into the hardware, or how the code running on the machine affects what happens inside the body of the machine. Both works remind us of the limits of the machine and the processes that happen inside it - how the machine has a life of its own. For code to come to life — to express its meaning — it needs a body to do so. Code needs to be embodied by a machine and its meaning cannot manifest without this machine.

5. The human body and the machine

When we livecode, we try to translate the concepts in our minds to code that runs on the computer, that will result in some audible or visible or otherwise experienceable outcome. Let us look at this in detail (see also figure 3): in our mind

we formulate concepts into code words that will make up the program, we then send commands through our nervous system to control our arms, hands and fingers to type in these words on the keyboard. The keyboard sends signals to a bus, the CPU gets a notification of these signals, and sends them to our editor program, which displays the code on the screen, and at the same time, puts the code words into the computers memory, so that given the right command, the editor can send the code to the interpreter, which will translate the code into machine code that is processed by the CPU to computer output data, which is then sent to output media, such as sound, light or visuals. Both the information from the display screen of the computer (our view on the editor), and the result of the code as observed from the output media, drive us to adjust our motor output (correct mistakes in our typing of the code, as we see it happen), as well as adapt our concepts or come up with the next ones, as we see the output media going into a certain direction.

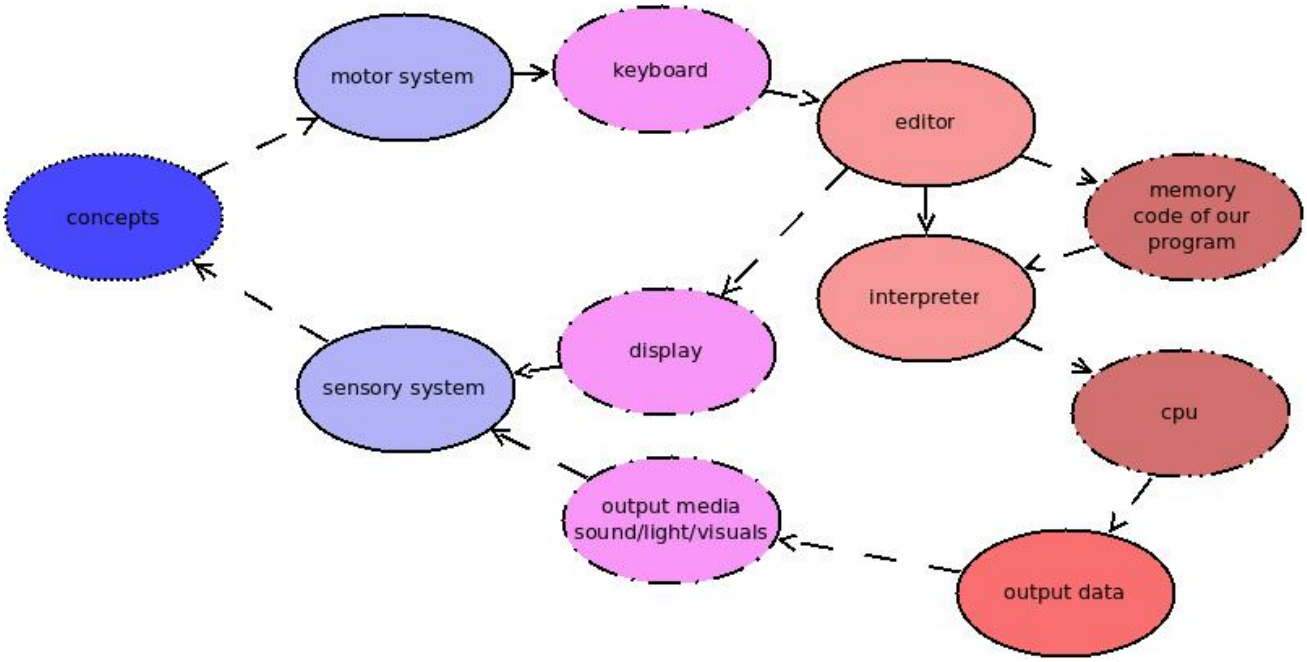


Figure 3: *Schematic view of interaction of the human body interfacing with the body of the machine to translate our concepts into code.*

As we prepare for the act of livecoding, we try to optimize the qualities of the machine - both the physical and the software environments within which we code - to make this interaction loop go faster and smoother; we may choose a specific keyboard and a screen, we choose our editors and how they display code, and we choose our output media. At the same time we train our bodies to perform the motor skills to control the machine better, such as learning how to touch type, fast switching between mouse and keyboard, using keyboard shortcuts, and so on. We train the motor skills needed for typing and correcting mistakes, so that they take less of our brain time to think about concepts (when I type, I tend to feel when I make a mistake, sometimes even before I see the letters appear on the screen; before I consciously realise I made a mistake, I already hit the backspace key to go back and correct the mistake). We have a need for speed, we want to prevent the bodies of the human and the machine from obstructing the translation of concepts to code that the machine understands. We do this by adapting our bodies; the physicality of the machine is important (the keyboard layout, the feel of the keys, the feel of the mouse or touchpad), and we even embody part of the coding languages in our body; if you always use sine oscillators (*SinOsc*) and never saw tooths (*Saw*), your fingers will almost automatically type *Si*, before you can hold back and adjust to type *Sa* in the rare occasion that you want a different sound.

Similar to how we put the code into the computer's memory, we keep in our own memory a representation of the code. When I work on code, I usually have a map in my mind of the structure of the code - when an error occurs, I first visualise internally where the origin of the error might be. I tend to object to the notion that SuperCollider (the object-oriented language I usually work with) is unlike the *visual* programming environments (such as PureData and Max/MSP), because its editor interface is text-based. In my mind, I navigate the code I work on as a visual map of interconnected objects.

In the performance *"Code LiveCode Live"* (Baalman 2009), I *"livecode the manipulation of the sound of livecoding, causing side effects, which are live coded to manipulate the sound of livecoding, causing side effects, ..."*. The concept of this livecoding performance is to only use the sound of the typing on the keyboard of the laptop as the source material for the audio in the performance. Furthermore, the sensing devices embedded in the laptop (e.g. accelerometers, touchpad, trackpoint, camera, the keyboard) are made accessible and mapped (through livecoding) to the manipulation of the sound material. Also the processes of the code itself (e.g. their memory and cpu usage) are used as sensors and used in the performance.

As such the performance of livecoding is influenced through its own side effects, transforming the code not only in the logical, programmatical manner, but as an embodied interface of the machine on our lap.

On the SuperCollider list an interesting comment was made about this performance; a viewer had watched the online documentation of the piece, and commented in a thread about the auto-completion features of the SuperCollider-Emacs interface that it seemed rather slow. I had to write back that no auto-completion features of the editor were used in that particular screencast. Apparently I succeeded in embodying the SuperCollider code to such an extend, that it seemed an enhancement of the editor, rather than the training of my body.

6. Conclusion

In this paper, I have reflected on different aspects how embodiment takes place within the livecoding practice. I emphasize on the aspects of the physicality of the act of livecoding - livecoding is not just the development of algorithms over time, our bodies play an important role in the performance practice of livecoding; likewise, the physicality of the machine plays an important role - its architecture determines in the end what the outcomes can be. We can only speculate what livecoding would have looked like today, if the Burroughs machine architectures had been predominant since their first invention — would we have had hardware that was designed to run SuperCollider so we would hardly need a compiler to run the code? Can we imagine a different interface than the keyboard and the screen to input the code into the machine, that will allow for a different way of embodying the code? What programming languages would evolve from such an interface? What kind of conversations would we have?

7. References

Baalman, Marije. 2009. "Code LiveCode Live." https://www.marijebaalman.eu/?cat=17.

Collective, Carpet. 2015. "Interdisciplinary Instant Composition - to Define, Share and Develop Concepts of Improvisation." http://www.instantcomposition.com/.

Collins, Nicolas. 2006. *Handmade Electronic Music : the Art of Hardware Hacking*. Routledge, Taylor; Francis Group.

Griffiths, Dave. 2010. "BetaBlocker." https://github.com/nebogeo/betablocker-ds.

Hayles, N.K. 2012. *How We Think: Digital Media and Contemporary Technogenesis*. University of Chicago Press. http://books.google.nl/books?id=737PygAACAAJ.

Mayer, Alastair J.W. 1982. "The Architecture of the Burroughs B5000 - 20 Years Later and Still Ahead of the Times?" *ACM SIGARCH Computer Architecture News* 10 (4): 3–10. http://www.smecc.org/The%20Architecture%20%20of%20the%20Burroughs%20B-5000.htm.

McLean, Alex, Dave Griffiths, Collins Nick, and Geraint Wiggins. 2010. "Visualisation of Live Code." In *Electronic Visualisation and the Arts (EVA), London 2010*. http://yaxu.org/visualisation-of-live-code/.

Reus, Jonathan. 2011. "Laptop Music." http://www.jonathanreus.com/index.php/project/laptop-music/.

———. 2012. "IMac Music." http://www.jonathanreus.com/index.php/project/imac-music/.

Rohrhuber, Julian, Tom Hall, and Alberto De Campo. 2011. "Dialects, Constraints, and Systems Within Systems." In *The SuperCollider Book*, edited by S. Wilson, D. Cottle, and N. Collins. Cambridge, MA: MIT Press.

Varela, Francisco J., Evan Thompson, and Eleanor Rosch. 1991. *The Embodied Mind: Cognitive Science and Human Experience*. MIT Press.

Wikipedia. 2015. "Harvard Architecture." http://en.wikipedia.org/wiki/Harvard_architecture.

Wilner, Wayne T. 1972. "B1700 Design and Implementation." http://bitsavers.trailing-edge.com/pdf/burroughs/B1700/Wilner_B1700designImp_May72.pdf.

Craft Practices of Live Coding Language Design

Alan F. Blackwell
afb21@cam.ac.uk

Sam Aaron
samaaron@gmail.com

University of Cambridge
Computer Laboratory

ABSTRACT

This paper reflects on the development process of two Live Coding languages, Blackwell's *Palimpsest* and Aaron's *Sonic Pi*, from the perspective of practice-led arts and craft research. Although informed by prior research in education, music, end-user programming and visual languages, these projects do not apply those principles through conventional software engineering processes or HCI techniques. As is often the case with practice-led research, the development process itself provides an opportunity for reflection on the nature of software as a craft – both for live-coding researchers, and for users of the live-coding systems that we create. In reflecting, we relate our own practice to recent perspectives on software as material, and on the role of craft as an element of interaction design research. The process that we have followed to support this analysis could be applied by other developers wishing to engage in similar reflection.

1. INTRODUCTION

Creating a new live coding language seems like research, but what kind of research is it? In this paper, we draw on our experience of creating two different live coding languages, in order to make the case that the creation of a programming language can be considered as a type of practice-led research, and that the practice of language creation is a craft practice (one that can be considered in contrast to either fine art or conventional software product design).

The process that we describe here, in one respect, describes the way that working programming language designers have always worked. As with many practice-led academic fields, this is as we should expect – the research should indeed be led by (i.e. it should follow) the practice. The distinct contribution made by the academic in this context is to offer an analytic rigour and critical reflexivity that is more difficult to achieve in the professional environment outside the academy – mainly because of pressures arising from commercial questions of business models and management, along with lack of opportunity for explicit conceptualisation through teaching. However, as in other practice-led areas of academic work, it is important that this presentation be informed by our own professional practice – one of us has been designing novel programming languages for over 30 years, and the other for over 15 years.

The first part of the paper presents a characterisation of the research process that we have been exploring. The second part evaluates this process in the light of our two practice-led case studies of live-coding language design. These are personal accounts, that we have tried to present in a reasonably personal manner, giving rich and reflective descriptions of our craft experiences.

2. CHARACTERISING PRACTICE-LED CRAFT RESEARCH

The characterisation of our research process starts from a perspective in programming language design that is informed by human-computer interaction (HCI) research, a field in which there is already a well-established understanding of design practice as a component of the research process. We will broaden this understanding, building on traditions of craft in design. However, in an academic context, this raises the question of how rigour should be understood, both as personal discipline and for independent evaluation. As noted by one reviewer of this paper, can practice-led craft research also exhibit scientific falsifiability, or by another reviewer, will the findings be replicable? We argue that, rather than the criteria of falsifiability and replicability applied in fields such as computer science, the key consideration in our research has been to maintain a critical technical practice in which technical activity is embedded within a recognized and acknowledged tradition, and subjected to rigorous reflection as argued in the rest of this section.

2.1. Research through Design

A programming language is both a user interface – between the programmer and the (virtual) machine – and also a mathematically-specified engineering system. As a result, programming language design combines aspects of human-computer interaction (HCI) and software engineering, alongside theoretical considerations from computer science. We argue that the most appropriate way to integrate these very different epistemological perspectives is by considering the joint human-technical experience as a craft design practice. In doing so, we extend the concept of research-through-design, where the development of a new (prototype) interactive product is regarded as an integral part of the research process (e.g. Fallman 2003, Gaver 2012). Although there are several different understandings of research-through-design in HCI, we suggest that the integration of artistic and practical craft orientation in programming language design can be understood within Fallman's "pragmatic account" of design as "a hermeneutic process of interpretation and creation of meaning" (Fallman 2003).

Although all design disciplines combine engineering with elements of art and craft, there is a tendency for software design to be considered only from the conservative perspective of engineering design, which chiefly aims to specify repeatable processes for design work. As a result, the design contribution of HCI can easily be isolated as representing a particular ideational phase within an engineering process (e.g. the sketching phase), or a particular process adjustment (e.g. iteration after usability testing of prototypes). Design commentators within HCI often focus on sketching and iteration as characteristic HCI design practices (Fallman 2003, Buxton 2007). This raises the question whether programming language design can or should be segregated into ideational and implementation phases. In addition, much academic programming language research is concerned with formal and mathematical analysis of language properties, with little consideration given either to the intended experience of the eventual programmer, or (at first) the question of how the language will be implemented.

2.2. Craft as Research

As an alternative, we note that there has always been a tension between this type of perspective, and the agenda of craft design, for example as advocated in the Bauhaus manifesto (Gropius 1919). Although considered unfashionable for many years, recent attention by scholars has returned to consideration of craft as a valuable perspective on contemporary life (Sennett 2008, Frayling 2011). Furthermore, the Crafts Council of England is pursuing this agenda, not only as an advocacy body for traditional crafts, but also as a potential contribution to contemporary industry and national wealth from professional trade skills, through their manifesto *Our Future is in the Making* (2015).

When considered as an aspect of programming language development, a craft view of design suggests the need to account for tacit or embodied knowledge of the skilled designer, rather than separating creative sketching from predictable engineering processes. It has been argued that when professional designers use digital tools, there is indeed an element of embodied knowledge that must be accommodated by the tool (McCullough 1998, Ehn 1998). Indeed, Noble and Biddle used the Bauhaus manifesto as a starting point for their provocative *Notes on Postmodern Programming* (2002), and research into programming in an arts context confirms that such programmers are comfortable with descriptions of their work as craft (Woolford et al 2010).

There is a popular interest in more practice-based approaches to software development, using terms such as 'software craftsmanship', or 'software carpentry'. A sophisticated analysis of software as craft, drawing on McCullough and Sennett among others, has recently been published by Lindell (2014). However, the description of software as craft requires an analogy between the ways that knowledge becomes embodied in the use of a physical tool, and the immaterial knowledge that is embedded in software development tools. Physical craft tools have 'evolved' to fit the practiced hand through generations of use - in fact 'co-evolved', because craft training proceeds alongside the reflective practices of making and adapting one's own tools. It might therefore be expected that the craft of software would be partly 'embodied' in programming tools that encode evolved expert practices such as prototyping, modelling and refactoring.

2.3. Software as Material

In craft traditions where there is believed to be significant embodied knowledge, creative artists and designers often advocate "letting the material take the lead" (e.g. Michalik 2011), working directly with materials and traditional tools in order to discover design outcomes through that embodied interaction

process. Programming language developers are indeed committed to their tools – in the case of our two projects, we find that craft traditions are embodied in our professional tools in ways that allow the craft designer to carry out technically competent work without conscious critical intervention that might otherwise be an obstacle to innovation. For Alan, this tool-embodied knowledge was obtained through the IntelliJ IDEA environment for Java development, and for Sam, Emacs and Ruby, both of which have evolved through reflective expert usage to support agile development practices, that also become an academic resource within a design research context.

The second point of comparison for this software-as-material analogy is that the professional designer is often described as having a "conversation with the material" (SchÖn 1983, SchÖn & Wiggins 1992), in which design concepts are refined by observing the development of a work in progress. A standard account from design ideation research relates this conversation specifically to sketches, arguing that sketches are intentionally produced in a manner that allows ambiguous readings (Goldschmidt 1999), so that the designer may perceive new arrangements of elements that would not be recognized in verbal descriptions or internal mental imagery (Chambers & Reisberg 1985). The understanding of software as material initially appears counter-intuitive, because of the fact that software is of course immaterial. However, we can draw on the understanding of materiality in interaction (Gross et al 2013) to observe that code is often a recalcitrant medium, offering resistance to manipulation by the programmer, in the same manner as the media materials of artistic practice.

2.4. Rigour in Practice-led Research

To some extent, these craft practices might simply resemble undisciplined programming – hacking, in the old terminology – as opposed to professional business software development processes of the kind that are prescribed in many textbooks (and indeed, taught to undergraduates by the first author). It is true that these projects, as with much academic software development, were carried out using methods that differ from professional commercial practice. However, as with academic research, there are also essential elements of rigour that are a prerequisite of success.

Within art and design disciplines, academic regulation has led to more precise statements of the criteria by which practice-led research should be evaluated (Rust 2007). The research process involves the creation of novel work, motivated and informed by prior theory, resulting in a product that is analysed in terms of that theory. This process is consciously reflexive – the object of analysis is not only a novel artefact (in the case studies below, our two novel programming systems), but also the process by which it was created (in this project, iterative development combined with rehearsal, performance, collaboration and creative exploration). This is significant because of the insistence in design research on interpretive and craft traditions – "designerly ways of knowing" (Cross 2001) – rather than treating research products purely as experimental apparatus for the purpose of testing hypotheses about human performance or market potential. Similarly, evaluation proceeds by reflection on precedent and process, rather than assessment (either by an audience or by potential users). The recalcitrance of the code "material" supports the same opportunities for intellectual discovery that are described by Pickering (1995) as the "mangle of practice" in which new kinds of scientific knowledge arise. This experience of discovery through the "material" resistance of code is identified by Lindell (2014) as an intuition shared by many programmers.

In summary, the process that we explore in this research responds to the requirements of programming language design by working in a way that greatly resembles the professional practice of programming language designers themselves. However, the standards of rigour to which it subscribes have been adopted from practice-led art, craft and design research, rather than conventional programming language design, in a manner that is consistent with characterisations of code as material, and of craft and materiality as a recent research orientation in HCI.

2.5. Toward a Critical Technical Practice

Following this lead, we propose four elements for craft-based research during live coding language development: i) understanding of the design canon – a body of exemplars that are recognised by our community of practice as having classic status; ii) critical insight derived from theoretical analysis and from engagement with audiences and critics via performance, experimentation, and field work; iii) diligent exploration via "material" practice in the craft of programming language implementation; and iv) reflective critical assessment of how this new work should be interpreted in relation to those prior elements.

Although these are discussed here as separate methodological elements, they were not divided into the ordered sequence of analysis, synthesis and evaluation that is characterised by Fallman (2003) as underlying the conservative account of the design process, but were integrated within a reflective craft practice as already discussed. In the broader computer science context, they suggest the opportunity for a truly practice-led approach to critical technical practice (Agre 1997), in which our standards of rigour are derived from those of practice, rather than the conventional scientific criteria of falsifiability and replicability.

2.6. Introduction to the case studies

In the remainder of this paper, we explore the potential of this research approach through two case studies of live coding language development. The two languages considered in these case studies are *Palimpsest* (developed by the first author, Alan) and *Sonic Pi* (developed by the second author, Sam). Palimpsest is a purely-visual language that supports interactive manipulation and collaging through constraint relationships between layers of an image (Blackwell 2014). Sonic Pi is a Ruby-based music live coding language that has been developed with support from the Raspberry Pi Foundation, to encourage creative educational experiences through programming (Aaron and Blackwell 2013, Burnard et al 2014). Sonic Pi has been widely deployed, on a range of platforms, and has a large user base. Sam performs regularly as a live coder using Sonic Pi, both as a solo artist and in a variety of ensembles. Palimpsest has not been released, but has been evaluated in a number of experimental studies with artists in different genres, and has occasionally been used in performance by Alan. Sam and Alan have performed together as audio-visual duo *The Humming Wires*, using Sonic Pi and Palimpsest on separate machines with networked interaction.

3. CASE STUDY 1: PALIMPSEST

The Palimpsest project emerged from an extended period of collaboration with professional artists who use digital media within conventional artistic genres (e.g. deLahunta 2004, Ferran 2006, Gernand et al 2011), studying their working practices, and creating new programming languages and programmable tools for them to use. It was also motivated by intuitions that new programming paradigms could exploit novel visual metaphors to widen audience (Blackwell 1996). In the context of design research, Alan's intention was to use his experience of collaboration with artists as a way to "jump away from" the conventional practices of software engineering, and understand alternative dimensions of the programming language design space (Gaver 2012).

3.1. Element i) Identifying design exemplars

The identification of creative experiences as a) desirable for computer users, and b) lacking in existing programmable systems, has been a starting point for major programming language innovations in the past, including Sutherland's *Sketchpad* (Sutherland 1963/2003) and Kay's *Dynabook* (Kay 1972) and *Smalltalk* user interface (Kay 1996) that inaugurated the Xerox Alto GUI (Blackwell 2006). Those visual environments were conceived as offering an alternative to computer programming, but in both cases had the significant side effect that they transformed the mainstream engineering paradigms of interacting with computers. Although programming continues to be regarded as primarily an engineering pursuit, there is no a priori reason to exclude programming from the domain of *ludic interaction* – computing for fun (Gaver 2002).

A second motivation for Palimpsest was to create a system in which the distinction between abstract notation and directly manipulated content was less strictly enforced. The spreadsheet, originating from *Visicalc*, is an example of a programmable system in which the user can directly view and manipulate the content of interest (columns of numbers), while also expressing constraints for automated processing of that content (spreadsheet formulae) and more complex data representations or algorithms (macros). Each of these can be regarded as notational "layers" beneath the surface of the cell array – the formulae and constraint relationships are normally invisible, but can be imagined as always present underneath the grid. Macros and database connections are not part of the grid at all, but a "back-end" service that is behind both the surface data and the formula layer.

This analytic metaphor of abstract notations being layered over or under other representations had been developed through previous experiments implemented by students (Blackwell & Wallach 2002), and was a critical insight motivating the Palimpsest project. In media and cultural studies, the word palimpsest has been extended from its original meaning of a text written over an erased original, to refer to the layered meanings that result when different cultural or historical readings replace each other or are superimposed

(Dillon 2007). The palimpsest thus offered a fertile metaphor for the integration of computation into the visual domain, as a way of exploring relationships between direct manipulation and abstraction layers. This conception of expressing computation purely as relations between visual layers also revives the long-standing challenge of the "purely visual" language, of which Smith's *Pygmalion* (Smith 1977), Furnas's *BitPict* (1991), and Citrin's *VIPR* (1994) have been inspiring examples.

3.2. Element ii) Critical orientation

These earlier investigations into programmable systems for creative applications had resulted in an engaging surface of direct manipulation that was more clearly associated with users' intentions (diagrams and cartoon line graphics in the case of Sketchpad, published pages of novels, music or artwork in the case of the Dynabook/Alto). But these early GUIs were not simply approximate simulations of pencils, drawing boards or musical instruments – they offered a more abstract layer of interpretation that was derived from programming language concepts, including collections, inheritance, data types and constraints. Such systems present computational expressivity via an abstract notation, while also allowing users to have direct access to representations of their work products. However many recent systems are no longer layered in this way. If the computation is not modifiable by the user, then we have a WYSIWYG system that offers direct manipulation but little abstraction (sometimes described as "What You See Is All You Get"). This was a motivating factor for Palimpsest, that had developed from the developer's theoretical and experimental investigations of abstraction as a key property of programming that is distinct from the user experience of direct manipulation (Blackwell 2002).

The separation between abstract notation and concrete manipulation has long been identified as an educational challenge, not only in teaching programming, but for mathematics education more generally. The "turtle" of Papert's *LOGO* language was motivated by Piagetian theories of educational transition from concrete to abstract thinking (Papert 1980). More recently, many systems to teach programming have integrated concrete manipulation of content into programming environments – this strategy is made explicit in systems such as *AgentSheets* (Repenning & Sumner 1995), *Alice* (Conway et al 2000) and *Scratch* (Resnick et al 2009), all of which provide direct manipulation assistance when interacting with the syntax of the abstract notation, while offering separate access to media content via preview windows and some combination of content creation and browsing tools. The objective in Palimpsest was to explore whether directly manipulated graphical content and abstract behavior specification could be integrated into the same (layered) visual surface.

Finally, this research was informed by critical attention to programming as a distinctive user experience (Blackwell & Fincher 2010). An emergent theme from previous research with practicing artists has been to explore intermediate ground between the different user experiences that are characteristic of sketching and of programming. When compared to computer programs, sketches have different notational conventions (Blackwell, Church et al 2008) and design functions (Eckert, Blackwell et al 2012). Sketching implies a different kind of intrinsic user experience (casual, exploratory, playful), but also different extrinsic user motivations (less concerned with detail, working quickly, open to unexpected outcomes). The intrinsic benefits of using digital tools are also likely to be driven by the importance of psychological flow in creative experience, rather than by task efficiency (Nash & Blackwell 2012), a factor that is directly relevant to live coding.

3.3. Element iii) Exploratory implementation

The implementation element of this project, situated within an experimental design research context, was not intended to take any of the above discussion as fixed requirements to be satisfied by a resulting product. Rather, these analytic perspectives provided initial starting points for exploration. Furthermore, serious consideration was given to the historical fact that the systems chosen as design exemplars often turned out to benefit users other than those who had inspired their development (successors to Smalltalk had benefits beyond children, successors to Visicalc had benefits beyond accountancy). For this reason, although the development process drew on understanding of different user experiences in order to emphasise sketch-like exploration, it was not assumed that this would result in a product meeting the needs of any specific user group. The objective was rather to support a full range of computational expressivity, in a manner that was accessible to a diverse user base, rather than meet an immediate end-user goal.

Preparatory work with professional artists had provided initial insight into alternative styles of user experience, and alternative approaches to computation, drawing on sketching practices. However, it also informed the development process, since the creative potential of following the "material" of software resembled an artistic process more than an engineering one. It is not necessary that a software development process emulate the work habits of the potential users. Nevertheless, adopting an artistic process offered a counter to the conventional approach to programming language design, which often results in products that meet the requirements of programming language designers (consider the respect given to languages that can be used to write their own compilers – a requirement that reflects a very small user base indeed, and distracts from the need to empathise with users who are unlike the designer).

The key consideration in the exploratory implementation process was to defer as far as possible any decisions regarding the eventual system functionality. Although preparatory collaborations and analytic concerns informed the general approach to be taken, the only starting point was the intention to create a "layer language". The first day of the project therefore started by Alan implementing a visual "layer", without yet knowing what this would imply. Over time, he conceived the relationship between these layers as a stack of superimposed treatments, then as a tree of regions that might be treated differently, then a separation between the stack and these regions, then a strict separation between content and control regions, then the introduction of a control region that provided a window onto content imported from another layer, and so on.

3.4. Element iv) Reflective assessment

A reflective diary recorded the more dramatic changes in design intention as they occurred, although these became less frequent over time. However large-scale changes in the system architecture continued to occur throughout the 10 month development period. Changes of this kind relied heavily on the refactoring capabilities of the development tools used – an example of the "craft" affordances that are now embodied in professional tools – in this case, IntelliJ Idea.

Some of these changes were prompted by early demonstrations to groups of researchers, or trials with prospective users. However, the great majority of the design changes were in response to Alan's own experience of using the prototype while it was under development, working in an intentionally isolated environment (a house in the forest, located in the Waitakere mountain range on the North West coast of New Zealand). During the most intensive development period, he used the prototype for approximately an hour each day. The intention in doing so was to provide a constant comparison between the "craft" experience of developing the Palimpsest prototypes in the recalcitrant material of the Java language, and the more exploratory sketch experiences that were intended for users of Palimpsest itself. When combined with his own previous research into end-user programming (drawing on approximately 30 years of experience), and previous experience of arts collaboration process research (extending over 10 years), this immersive practice was intended to promote reflective design, and theory development that emerged from a personal creative practice.

Rigorous reflection on work in progress need not be conducted in the manner of an engineering investigation – indeed, Alan feels that his most valuable findings were creative insights that occurred after periods of relatively unconscious reflection (while jetlagged, or on waking from sleep (Wieth, & Zacks 2011)). These findings were seldom attributable to a process of hypothesis formation and testing. Although he found it valuable to compare his own experiences with those of other users, observing a single user session generally provided sufficient external stimulus to inform a month of further development work.

4. CASE STUDY 2: SONIC PI

The development of Sonic Pi has followed three distinct phases, each oriented toward a different user population. The first of these was the development of a tool that would support school programming lessons, with music used as a motivating example. In this phase, collaboration with teacher Carrie-Anne Philbin shaped the requirements and application of the interface (Burnard, Aaron & Blackwell 2014). The second phase was the project *Defining Pi*, funded by the Arts Council of England, and in collaboration with Wysing Arts Centre and arts policy consultant Rachel Drury. In this phase, several artists were commissioned to make new work that extended Sonic Pi in ways that might be accessible to young makers (Blackwell, Aaron & Drury 2014). The third phase, named *Sonic Pi Live and Coding*, was funded by the Digital R&D for the Arts scheme, led by performing arts venue Cambridge Junction, and working in

collaboration with education researcher Pam Burnard as well as a large number of other individuals and organisations. In this phase, the focus was on support for the school music curriculum rather than simply programming, and included a summer school for 60 participants (Burnard, Florack et al 2014)

4.1. Element i) Identifying design exemplars

Sonic Pi was created for educational use, but with the intention that it should be a live-coding language from the outset. As a result, it inherited influences from *Overtone*, a system that Sam had co-designed with Jeff Rose. Sonic Pi draws heavily from Overtone's system architecture and there are many direct correlations from a system design perspective. For example, both systems sit on top of the SuperCollider synthesiser server and both are built to enable concurrent interaction with the server from multiple threads. Overtone itself was influenced directly from Sorensen's *Impromptu* (and later *Extempore*) with ideas about pitch representation essentially migrating from Impromptu to Sonic Pi via Overtone. McLean's *Tidal* was also an influence purely in terms of its operational semantics and expressivity over pattern. Finally, Magnusson's *ixi lang* was a large influence with respect to its simple uncluttered syntax and tight relationship between visible syntax and sound.

In the education domain there are three systems that were influential as design exemplars for Sonic Pi. First is *LOGO* - which demonstrated how an extremely simple system both syntactically and semantically can allow children to create sophisticated output. It also was a system that used creativity as a tool for engagement - a philosophy Sonic Pi strongly associates with. Second is *Scratch* - a graphical language which allows users to nest 'pluggable' pieces of syntax to represent semantics. This nested approach to syntactic structure is followed closely by Sonic Pi. Finally, *Shoes* by Why the Lucky Stiff demonstrated how Ruby's syntax is flexible enough to represent sophisticated concepts in a simple and readable manner.

4.2. Element ii) Critical orientation

The two primary goals for Sonic Pi were that it should be easy to teach to 10 year olds, and that it should be possible to perform with. The teaching focus was not solely a question of motivation and learnability, but to support the classroom practice of Carrie-Anne Philbin, the teacher with whom the initial curriculum was developed. This meant removing as much as possible anything that related only to the tool, rather than her teaching goals. Understanding these considerations involved direct observation of children using the language (and its various iterations), experience in classrooms, and many conversations with teachers and education experts. Imagining a teacher explaining an approach to a 10 year old in the classroom means asking how each idea or abstraction works with other similar abstractions. This idea of similarity seems to be very important, and must be combined with a very simple starting point. Any new feature or idea needs to not 'pollute' this initial starting experience. Simple things should remain simple (syntactically and semantically) whilst sophisticated things may be complex provided that complexity fits the abstract constraint reflection properties previously described.

With regard to performance, a key priority was that it should be possible to anticipate what the musical output would be, in a way that is not always true with other live coding languages. Sam's previous work with Overtone had resulted in a tool for building instruments, whereas Sonic Pi was intended to *be* an instrument. This new emphasis arose in part through the practical advantages of performing as a solo artist – Sam's band Meta-eX was structured as a duo, with an instrumentalist playing controllers whose sounds and behaviours were customized using Overtone. The change to a solo environment meant that the huge flexibility that had been designed into Overtone became more constrained in Sonic Pi, with many technical choices defined in advance. These include the use of stereo buses throughout, a fixed set of synthesisers, and effects that can only be combined in linear stereo chains. Many of these features offer a more simplified conceptual model to young performers. For example, every synth has a finite duration, notes are expressed in MIDI (although fractions are allowed for microtonal effects), synth and FX lifecycle are fully automated (similar to garbage collection in modern languages), and logical thread-local clocks allow time to be easily manipulated in an explicit manner enabling the construction of further timing semantics (Aaron et al 2014).

4.3. Element iii) Exploratory implementation

The earlier development of Overtone had been strongly embedded within the characteristic software craft perspective of exploratory development in the Lisp/Scheme language family – here the Clojure dialect. Sam's tools were the familiar customizable Emacs environment, with his personal set of extensions having

become popular among other live coders – the *Emacs Live* system. However, the concern with performance as a craft-practice of its own has led the implementation work in an interesting direction.

Sonic Pi isn't just about teaching computing skills in isolation. It uses music as a tool for framing the computational ideas and to engage the learner. A core goal has therefore been to demonstrate that the music part isn't simply 'bolted on' to enhance the experience, but that the learner sees Sonic Pi as a credible performance tool. The motivation should be "I want to be able to make music that is interesting!" In order to ensure that Sonic Pi enables interesting performance, Sam has had to become a dedicated Sonic Pi performer himself. Since Oct 2014 he has used only Sonic Pi for his live coding performances, rehearsal and personal practice/preparation.

This introduces the challenge of separating his development work (coding new ideas) from practice with using Sonic Pi as a musical instrument. Sam's experience is that these are very distinct coding activities. Given that his background and comfort zone was in traditional software development rather than music performance, it was very easy to interrupt any practice session with a coding session. This would therefore interrupt the performance flow often for the remaining duration of the practice period. In order to ensure that this did not happen, his development practices have acquired the following disciplines:

- Allocate distinct times for each activity. Typically, this would involve development coding in the daytime and performance practice in the evening.
- Performance practice was done solely on the Raspberry Pi computer. This was initially in order to ensure that performance was optimized for the Raspberry Pi, so that Sam could present it as a serious instrument (avoiding "hey kids, use the Raspberry Pi for live coding, however I use an expensive Mac").
- Meanwhile, the low budget architecture of the original Raspberry Pi meant it was conveniently impractical to continue coding on the same machine. It couldn't smoothly run Emacs with the 'Emacs Live' extensions developed for Overtone, and an Emacs session running concurrently with the Sonic Pi environment would cause CPU contention disrupting the audio synthesis.
- Practice time was also differentiated by being offline – indeed, Sam would remove all laptops and phones from the room whilst practicing, so that he didn't have an idea, start googling it, and before he realised it would be back coding on his Mac again…

4.4. Element iv) Reflective assessment

The strict separation between performance and development meant that Sam would have lots of neat ideas in the live-coding moment which were promptly forgotten. He therefore started keeping a paper diary in which he throws notes down as quickly as possible when he has an idea, then continues practicing. The diary-like note taking has subsequently become a central aspect of ongoing Sonic Pi development and formed the backbone of a reflective discipline. Often during a practice session Sam will be in a musical position and have an idea of where he'd like to go, but run into syntactic or semantic barriers in the language that prevent him getting there easily. Or, he'll find himself writing boilerplate code again and again and therefore start imagining a nicer way of automating the repeated task. Ideas that would spring into his head he'd just jot down.

Occasionally, when there is no obvious solution to a problem, he might allow himself to interrupt a practice session (if the opportunity has sufficient merit) and then spend a bit of time imagining he has already implemented the solution, so that he can see if he could sketch the solution to the idea in Sonic Pi. This sketching activity does not seem like development work, but has far more the character of reflective craft – design ideation that must be captured in the performance moment. Sam describes this as 'a feeling that my mind had been tuned specifically for the context of performance and so I'd be able to consider solutions in that light'. If he came up with a decent solution, he could then jot this down in his notes and continue performing. In actuality, implementing things during practice sessions has become rarer and rarer. Instead, Sam accumulates a list of ideas (sometimes the list would be short, other times incredibly long) and the following day he reads through the list to potentially motivate aspects of that day's development. Often ideas don't make much sense the next day, and can just be ignored.

The following evening's practice session starts with a fresh page - if the same problem arises during two consecutive practice sessions, or even repeatedly, annoyance becomes a key motivator for working on a

solution. However, both the software and Sam's experiences are constantly changing, offering new perspectives. As a result, it is occasionally useful to work through all his previous notes in a single sitting, 'mining for nice ideas'.

These private practices are supplemented by ensemble collaboration. As mentioned earlier, Sam and Alan perform together occasionally as *The Humming Wires*. Rehearsals for those performances often result in exchange of design principles that result from tension between two very different execution paradigms: the underlying constraint resolution architecture of Palimpsest, and the data flow graph and event structure of Supercollider. Sam finds similar creative tension in other collaborative performance contexts, for example in his new band *Poly Core,* in which he improvises live with a guitarist, finding that when they discuss new ideas, it is valuable to code them in the rehearsal moment. Some can be achieved fluently, whereas others are not so simple - frictions that potentially become new entries on the list of coding tasks.

5. CONCLUSIONS

In this paper, we have described our experiences of programming language development from the personal perspective of live coding practitioners. We do not think that these experiences are unique – on the contrary, we expect that many of the observations made in the above case studies will be very familiar, or at least will resonate with others who create live coding languages. However, we also believe that these kinds of experience will appear familiar to all programming language developers, whether or not they consider themselves 'live coders' as such. We think it may be very interesting to explore analogies between the performance / rehearsal / development distinction in live coding work, and the transitions between implementation and experiential perspectives in other programming languages.

Despite the formal and abstract presentation of many programming language design textbooks, we believe that most programming language designers are motivated by a need to change the user experience of programming – usually based from introspection on their own experience, or perhaps (if academics) from observation of conceptual problems faced by their students. Unfortunately, as a result, many programming languages are designed only from the perspective of people who are already programmers, and without opportunities for the kinds of creative tension and reflection in practice that we have described in this paper. Research in end-user programming has attempted to address this deficiency through the application and adaptation of user-centered HCI research methods for the programming domain – including development of user models, techniques for analytic evaluation and critique, and a mix of contextual and controlled user study methods.

These projects have explored an alternative approach, influenced by the methods of practice-led design research. They have been derived from the availability of a new generation of "craft" tools for agile software development, including Emacs, Git, IntelliJ, Ruby – and enabling design concepts to emerge from the actual construction of a prototype language. Many aspects of these designs, including significant conceptual foundations, develop through reflection on the experience of building and using the system. The research outcomes can be attributed in part to insight and serendipity, in a manner that while probably recognizable to many research scientists, is not normally specified either in prescriptions of scientific method, or of user-centred software development methodology.

These projects have stepped away from "requirements capture", back to the traditions of craft professions in which tools are made and adapted by those who actually use them. They intentionally blur the boundary of software development and creative exploration, and avoid many conventional practices in each field. We hope that the results might be of value, by highlighting the systematic and rigorous alternatives that can be taken to the craft of programming language design.

Acknowledgments

The development of Sonic Pi has been supported by generous donations from Broadcom Corporation and the Raspberry Pi Foundation. The development of Palimpsest was supported by a grant from Kodak, sabbatical leave from the University of Cambridge, and a visiting appointment at the University of Auckland, hosted by Beryl Plimmer. Other aspects of the work described have been funded by grants from the Arts Council of England, and the Digital R&D for the Arts programme.

6. REFERENCES

Aaron, S. and Blackwell, A.F. (2013). From Sonic Pi to Overtone: Creative musical experiences with domain-specific and functional languages. Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design, pp. 35-46.

Aaron, S., Orchard, D. and Blackwell, A.F. (2014). Temporal semantics for a live coding language. In Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design (FARM '14). ACM, New York, NY, USA, 37-47.

Agre, P. E. (1997). Towards a Critical Technical Practice: Lessons Learned in Trying to Reform AI. In Bowker, G. Star, S. L & Turner, W. (Eds) *Social Science, Technical Systems and Cooperative Work*, Lawrence Erlbaum, Mahwah, NJ, pp. 131-157.

Blackwell, A.F. (1996). Chasing the Intuition of an Industry: Can Pictures Really Help Us Think? In M. Ireland (Ed.), *Proceedings of the first Psychology of Programming Interest Group* `Postgraduate Student Workshop`, pp. 13-24.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 2-10.

Blackwell, A.F. (2006). The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction (TOCHI),* 13(4), 490-530.

Blackwell, A.F., Aaron, S. and Drury, R. (2014). Exploring creative learning for the internet of things era In B. du Boulay and J. Good (Eds). *Proceedings of the Psychology of Programming Interest Group Annual Conference(PPIG 2014)*, pp. 147-158.

Blackwell, A.F., Church, L., Plimmer, B. and Gray, D. (2008). Formality in sketches and visual representation: Some informal reflections. In B. Plimmer and T. Hammond (Eds). *Sketch Tools for Diagramming*, workshop at VL/HCC 2008, pp. 11-18.

Blackwell, A.F. and Fincher, S. (2010). PUX: Patterns of User Experience. *interactions* 17(2), 27-31.

Blackwell, A.F. and Wallach, H. (2002). Diagrammatic integration of abstract operations into software work contexts. In M. Hegarty, B. Meyer and N.H.Narayanan (Eds.), *Diagrammatic Representation and Inference*, Springer-Verlag, pp. 191-205.

Burnard, P., Aaron, S. and Blackwell, A.F. (2014). Researching coding collaboratively in classrooms: Developing Sonic Pi. In `Proceedings of the Sempre MET2014: Researching Music, Education, Technology: Critical Insights` Society for Education and Music Psychology Research, pp. 55-58.

Burnard, P., Florack, F., Blackwell, A.F., Aaron, S., Philbin, C.A., Stott, J. and Morris, S. (2014) Learning from live coding music performance using Sonic Pi: Perspectives from collaborations between computer scientists, teachers and young adolescent learners Paper presented at Live Coding Research Network.

Buxton, B. (2007). Sketching User Experiences: getting the design right and the right design. Morgan Kaufmann. 2007

Chambers, D. & Reisberg, D. (1985). Can mental images be ambiguous? *Journal of Experimental Psychology: Human Perception and Performance* 11:317-328.

Church, L., Rothwell, N., Downie, M., deLahunta, S. and Blackwell, A.F. (2012). Sketching by programming in the Choreographic Language Agent. In *Proceedings of the Psychology of Programming Interest Group Annual Conference. (PPIG 2012),* pp. 163-174.

Citrin, W., Doherty, M. and Zorn, B. (1994). Formal semantics of control in a completely visual programming language. In *Proc. IEEE Symposium on Visual Languages*. 1994. St. Louis, 208-215.

Conway, M., Audia, S., Burnette, T., Cosgrove, D. and Christiansen, K. (2000). Alice: lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI conference on human factors in computing systems* (CHI '00). ACM, New York, NY, USA, 486-493.

Crafts Council of England (2015). *Our Future is in the Making*. Available as print publication and from www.craftscouncil.org.uk/educationmanifesto

Cross, N. (2001). Designerly ways of knowing: design discipline versus design science. *Design Issues*, 17(3), pp. 49–55.

DeLahunta, S., McGregor, W. and Blackwell, A.F. (2004). Transactables. *Performance Research* **9**(2), 67-72.

Dillon, S. (2007). The Palimpsest: Literature, criticism, theory. Continuum.

Eckert, C., Blackwell, A.F., Stacey, M., Earl, C. and Church, L. (2012). Sketching across design domains: Roles and formalities. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 26(3), 245-266.

Ehn, P. (1998). Manifesto for a digital Bauhaus. *Digital Creativity* 9(4), 207-217.

Fallman, D. (2003). Design-oriented human-computer interaction. In *Proceedings of the SIGCHI conference on human factors in computing systems* (CHI '03). ACM, pp. 225-232.

Ferran, B. (2006). Creating a program of support for art and science collaborations. *Leonardo* 39(5), 441-445.

Frayling, C. (2011). *On Craftsmanship: Towards a new Bauhaus*. Oberon Masters.

Furnas, G.W. (1991). New graphical reasoning models for understanding graphical interfaces. *Proceedings of the SIGCHI conference on human factors in computing systems* (CHI′91), 71-78.

Gaver, W. (2002). Designing for Homo Ludens. *I3 Magazine* No. 12, June 2002.

Gaver, W. (2012). What should we expect from research through design? In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems (CHI '12).* ACM, New York, NY, USA, 937-946.

Gernand, B., Blackwell, A. and MacLeod, N. (2011). *Coded Chimera: Exploring relationships between sculptural form making and biological morphogenesis through computer modelling*. Cambridge, UK: Crucible Network.

Goldschmidt, G. (1999). The backtalk of self-generated sketches. In JS Gero & B Tversky (Eds*), Visual and Spatial Reasoning in Design*, Cambridge, MA. Sydney, Australia: Key Centre of Design Computing and Cognition, University of Sydney, pp. 163-184.

Gropius, W. (1919). *Manifesto and programme of the state Bauhaus in Weimar, April 1919*. http://bauhaus-online.de/en/atlas/das-bauhaus/idee/manifest

Gross, S. Bardzell, J. and Bardzell, S. (2014). Structures, forms, and stuff: the materiality and medium of interaction. *Personal and Ubiquitous Computing* 18(3), 637-649.

Kay, A. (1972). *A personal computer for children of all ages*. Xerox PARC Research Report.

Kay, A. (1996). The early history of Smalltalk. In *History of Programming Languages II*, T.J. Bergin, Jr. and R.G. Gibson, Jr., eds. ACM, New York. 511--598.

Lindell, R. (2014). Crafting interaction: The epistemology of modern programming. Personal and Ubiquitous Computing 18, 613-624.

McCullough, M. (1998). *Abstracting craft: The practiced digital hand*. MIT Press

Michalik, D. (2011) Cork: Letting the material take the lead. *Core 77*, entry dated 4 Oct 2011. http://www.core77.com/blog/materials/cork_letting_the_material_lead_20707.asp [last accessed 27 Aug 2012]

Nash, C. and Blackwell, A.F. (2012). Liveness and flow in notation use. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME),* pp. 76-81.

Noble, J. and Biddle, R. (2002). Notes on postmodern programming. In Richard Gabriel, editor, *Proceedings of the Onward Track at OOPSLA 02*, pages 49-71.

Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. Harvester Press, Brighton, UK.

Pickering, A. (1995). *The Mangle of Practice: Time, agency and science*. University of Chicago Press.

Repenning, A., & Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, *28*(3), 17-25.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, November 2009.

Rust, C. Mottram, J. Till, J. (2007) *Review of practice-led research in art, design & architecture.* Arts and Humanities Research Council, Bristol, UK

Schön, D.A. (1983). *The Reflective Practitioner: How professionals think in action*. New York, NY: Basic Books.

Schön, D.A. and Wiggins, G. (1992). Kinds of seeing and their function in designing, *Design Studies*, **13**:135-156.

Sennett, R. (2008). *The Craftsman*. New Haven: Yale University Press.

Smith, D.C. (1977). *PYGMALION: A Computer Program to Model and Stimulate Creative Thinking*. Birkhäuser, Basel, Switzerland.

Sutherland, I.E. (1963/2003). *Sketchpad, a man-machine graphical communication system*. PhD Thesis at Massachusetts Institute of Technology, online version and editors' introduction by A.F. Blackwell & K. Rodden. Technical Report 574. Cambridge University Computer Laboratory

Wieth, M.B. and Zacks R.T. (2011): Time of day effects on problem solving: When the non-optimal is optimal. *Thinking and Reasoning* **17**, 387-401

Woolford, K., Blackwell, A.F., Norman, S.J. & Chevalier, C. (2010). Crafting a critical technical practice. *Leonardo* 43(2), 202-203.

Hugh Davies's Electroacoustic Musical Instruments and their Relation to Present-Day Live Coding Practice: Some Historic Precedents and Similarities

James Mooney
University of Leeds
j.r.mooney@leeds.ac.uk

ABSTRACT

The purpose of this paper is to present the self-built electroacoustic musical instruments of Hugh Davies (1943-2005) to the international live coding community, and to propose points of similarity between Davies's practice and present-day live coding practice. In the first part of the paper, the context within which Davies's instrument-building practice developed, in the late 1960s, is outlined, and a number of specific instruments are described. Aspects of Davies's performance style, repertoire, and the ensembles with which he performed are discussed, as are activities such as instrument-building workshops and public exhibitions of instruments, in which he regularly participated. In the second part of the paper, four areas of connection with present-day live coding practice are suggested. Respectively, these focus upon live coding's status: (1) as part of a long historic tradition of live electronic music performance (as opposed to electronic music constructed in the studio); (2) as a practice in which the performer him or herself builds the apparatus (whether physical or code-based) through which the music is mediated; (3) as an improvised or semi-improvised art-form in which music is developed in real time, within a framework bounded by material or quasi-material constraints; and (4) as a community of practice with a distinct agenda of promoting understanding through engagement. This paper is presented as a case study in exploring live coding's historic precedents, and as a contribution toward situating live coding within a broader historical, cultural context.

1. INTRODUCTION

Hugh Davies (1943-2005) was a musician, historical musicologist, and instrument-builder, professionally active from 1964 up until his death in 2005. As well as making significant contributions to the documentation of electroacoustic music's history (Mooney 2015a), throughout his career Davies built more than 120 musical instruments and sound sculptures that 'incorporate[d] found objects and cast-off materials' (Roberts 2001) such as kitchen utensils, plastic bottles, springs, hacksaw blades, and many other materials that might normally be considered 'junk.'

The reader is encouraged to watch the following video, in which Davies plays and briefly talks about one of his self-built instruments: https://www.youtube.com/watch?v=wPT9A0IsGgs (Klapper 1991). Specifically, Davies plays the first of his self-built solo performance instruments, which is a device called the Shozyg. (Towards the end of the video he also plays a number of others.) The Shozyg was built in 1968, and consists of a collection of fretsaw blades, a ball-bearing, and a spring, the sounds of which are amplified via two contact microphones that feed a stereo output. These objects are mounted inside the cover of a book that has had its pages removed; this is an encyclopaedia volume that covers the alphabetic range of topics from SHO to ZYG, which is where the instrument gets its name from. The Shozyg is electroacoustic because the means of initial sound production are acoustic, but the vibrations—which would be too tiny to hear otherwise—are amplified electronically. The Shozyg was designed to be played with the fingers or with the aid of accessories such as 'needle files, small screwdrivers, matchsticks, combs, small electric motors, small brushes, coins, keys, etc.' (Davies 1968a). (In the video Davies appeared to be using a screwdriver.) A second model of the Shozyg was built later the same year, comprising a different set of amplified objects; both models of the Shozyg are shown in Figure 1, below.

Keith Potter—a close colleague of Davies's for many years at Goldsmiths, University of London—made the following comments in an obituary that he wrote in the *Independent* newspaper:

> [I]n the 21st century, it seems that Hugh Davies's innovatory, do-it-yourself, lo-fi approach—*which in several respects prefigured present laptop culture*—is finding favour with a younger generation to whom this remarkable and iconoclastic innovator now appears as a significant father figure. (Potter 2005, emphasis added)

Potter does not specify precisely *how* Davies's approach prefigured present laptop culture, nor indeed which specific laptop culture it prefigured; but Potter's comments suggest that there might be some connections between Davies's instrument-building practice, which began in the late 1960s, and present-day live coding practice. The purpose of this paper, then, is to begin to explore what some of those connections might be.

The author has previously suggested some speculative points of contact between Davies's instrument-building practice and live coding, based upon three recurring themes in Davies's work: materiality, economy, and community (Mooney 2015b). In the current paper, two of these themes (materiality and community) are developed further; the third theme (economy) has for the time being been dropped, in order to allow for a greater focus upon the aspects that are most directly relevant to the field of live coding, though it is still considered relevant and will be further explored in the future.

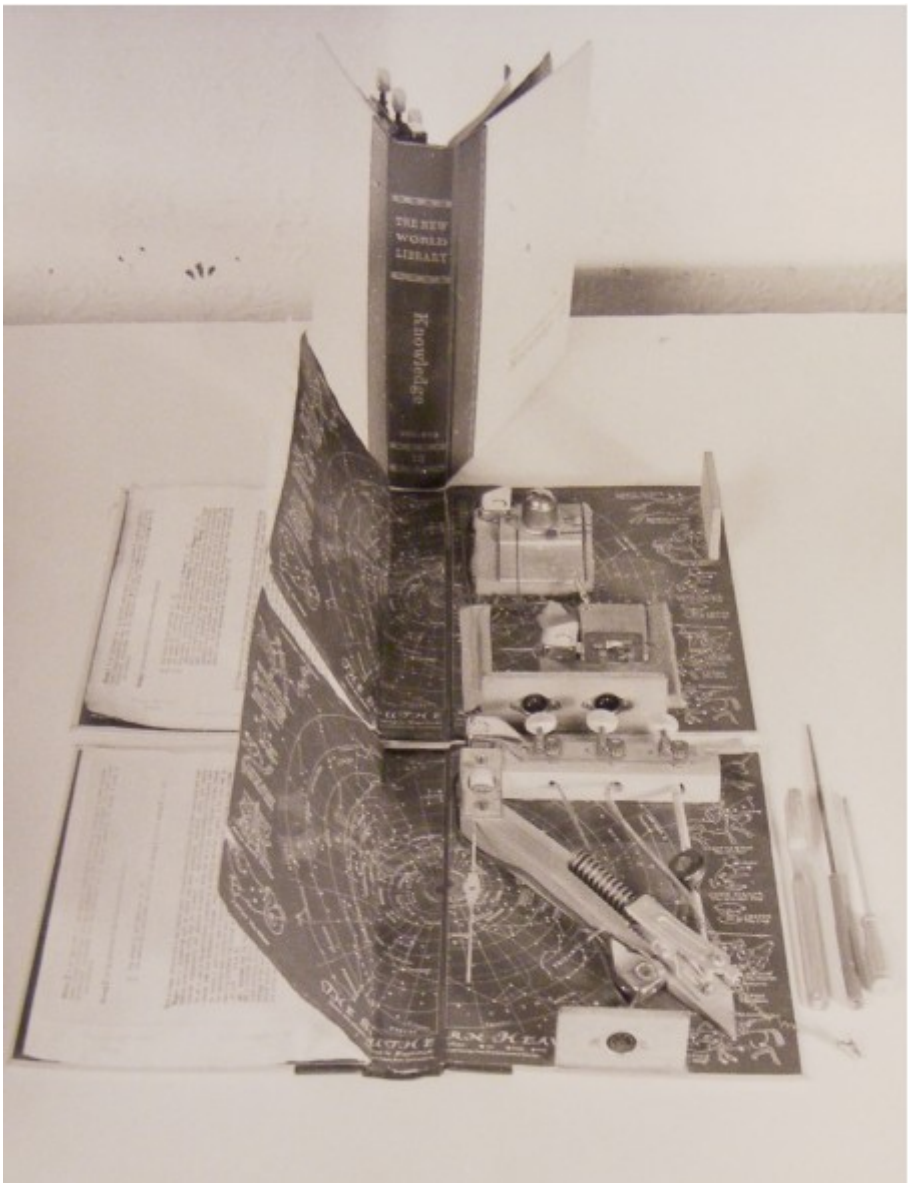


Figure 1. Shozyg I (above); Shozyg II (below). Photo © Pam Davies. Courtesy of The British Library.

2. BACKGROUND AND CONTEXT

In 1964, at the age of 21, Davies became personal assistant to the avant-garde composer Karlheinz Stockhausen. Davies lived for 2 years in Cologne, where, amongst other things, he became involved in

performances of Stockhausen's latest work, *Mikrophonie I* (1964). *Mikrophonie I* is a piece in which two performers excite the surface of a large tam-tam gong, using a range of different objects such as drinking glasses, cardboard tubes, hand-held battery-operated fans, and kitchen implements (Davies 1968b). A further two performers amplify the sounds produced by the tam-tam using hand-held microphones, so that details of the sound that would otherwise be inaudible can be heard. Two final performers affect the further transformation of the amplified sounds using electronic filters, as well as controlling the volume; thus, there are six performers in total.

With respect to Davies's nascent instrument-building practice, three things are significant about *Mikrophonie I*. First, it involved the repurposing—which is to say, 'hacking'—of every-day objects as musical instruments. Second, it involved the electronic amplification of acoustic sounds that would otherwise be too quiet to hear. Third, and above all, it was a work of 'live electronic' music, that is, it involved the use of electronic equipment to manipulate sound in a live performance context, as opposed to producing electronic music on magnetic tape in the studio. From the end of World War II up until at least the beginning of the 1960s, experimental work in electronic music was overwhelmingly dominated by magnetic tape (Davies 2001, p.98); sounds recorded and transformed in the studio using tape manipulation techniques were painstakingly assembled into compositions by cutting the tape up with a razor blade and sticking it back together with splicing tape. A finished composition could easily take months to realise. The practice of producing electronic music in real time, to a reasonable approximation, did not exist, and did not start to become common until the 1960s; and Stockhausen's *Mikrophonie I* was among the earliest pieces to systematically explore this new area.

The three characteristics just highlighted—live electronic music, amplification, and the hacking of every-day objects—went on to become defining characteristics of Davies's instrument-building practice, and Davies himself acknowledged the influence of his experience as Stockhausen's assistant in catalysing this aspect of his work (Davies 1997, p.12). Another influential early work of live electronic music, however, was John Cage's *Cartridge Music* (1960), in which the otherwise inaudible sounds of various found objects are amplified by inserting them into the apertures of gramophone cartridges.

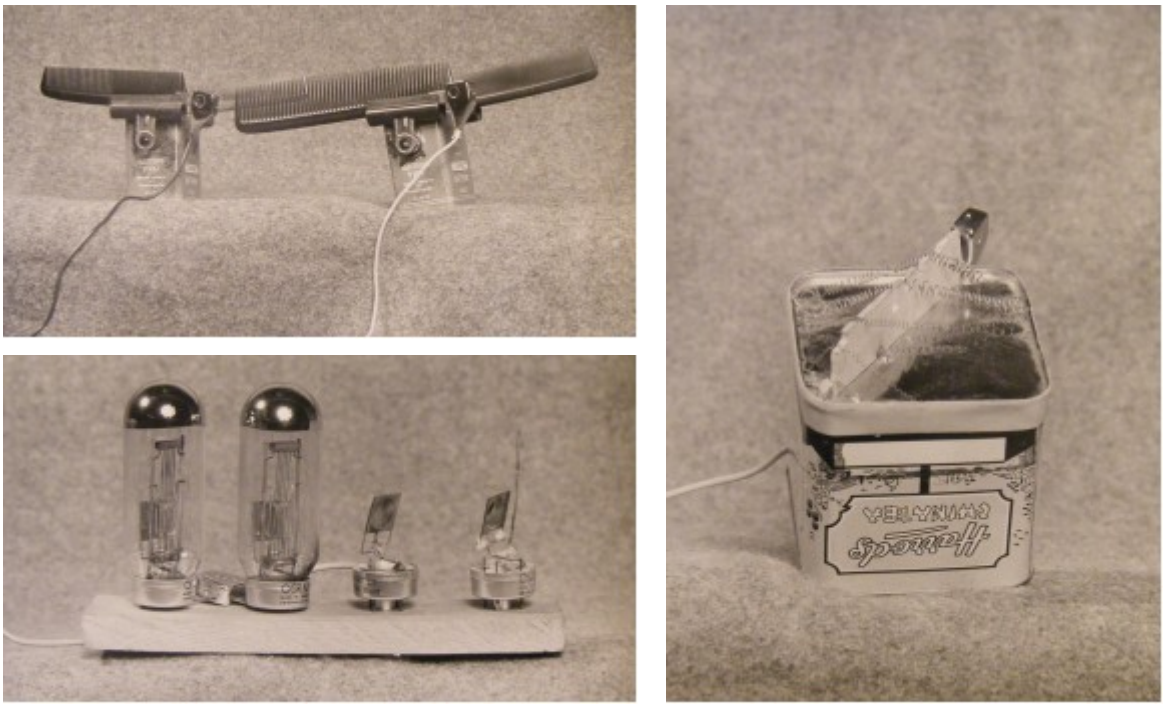


Figure 2. Some of Davies's early instruments. Photos © Pam Davies. Courtesy of The British Library.

3. EARLY INSTRUMENTS (1967-8)

On returning to England, Davies found that he no longer had access to the sophisticated equipment that had been available during his time as Stockhausen's assistant, and could not afford to buy any such equipment of his own (Davies 1979, p.1). Hence, the very earliest of Davies's self-built instruments—dating from 1967—represent Davies's efforts to imitate some of the techniques he had encountered in *Mikrophonie I* using

found or cheaply available objects, including combs, broken light-bulbs, and springs stretched across the opening of a metal tin (see Figure 2). The sounds of these objects were amplified via contact microphones. These early instruments were originally developed as ways of generating sound material for tape-music compositions (Davies 1997, p.12), but Davies soon recognised the potential for using such contraptions in a live performance context, and began to build instruments specifically with live performance in mind.

4. FIRST LIVE PERFORMANCE INSTRUMENTS (1968-72)

As mentioned previously, the first of Davies's performance instruments was the Shozyg, of which two different models were produced; these comprised a prefabricated selection of objects mounted inside the cover of a book, and a range of implements or accessories that could be selected to activate those fixed components of the instrument in performance. Beginning in 1970, Davies built a dozen so-called Springboards (Mk. III is shown in Figure 3), in which 'a number of springs (from two upwards) are mounted on a wooden board, and treated rather like strings' (Davies 1997, p.12). The springs were amplified, usually using magnetic pickups. Another one of Davies's instruments was the Concert Aeolian Harp (shown in Figure 4), first built in 1972, which consisted of a collection of 'thin fretsaw blades […] mounted in a holder […] [which were] blown on by the human breath as well as played with a variety of miniature implements such as a feather and a single hair from a violin bow' (Davies 1997, p.13).

Figure 3. Davies with Springboard Mk. III. Photo © Michael Dunn. Courtesy of The British Library.

Davies combined several of his self-built instruments in a compound instrument that he referred to as his Solo Performance Table (a.k.a. Multiple Shozyg, 1969-72). The Solo Performance Table (see Figure 4) incorporated versions of the three instruments mentioned previously—the Shozyg (Mk. II), Springboard (Mk. V), and Aeolian Harp—as well as an amplified 3D photograph, 'whose grooves [were] played by running fingernails across them at different speeds', two unstretched springs amplified via a magnetic pickup, a metal egg-slicer amplified via a further magnetic pickup on top of which it is sitting, two long springs 'with keyrings by which to vary their tension', again amplified via the aforementioned pickups, and

a guitar string amplified by being mounted inside a turntable cartridge (recalling Cage's *Cartridge Music*), 'whose tension is varied by [a] bamboo holder', and which is either plucked or bowed (Davies 1974). In all of these self-built instruments Davies's tendency to repurpose, modify, or hack ready-made or every-day objects can clearly be seen. A custom-built multi-channel mixer was used to mix the various amplified sounds together in real time during performance.

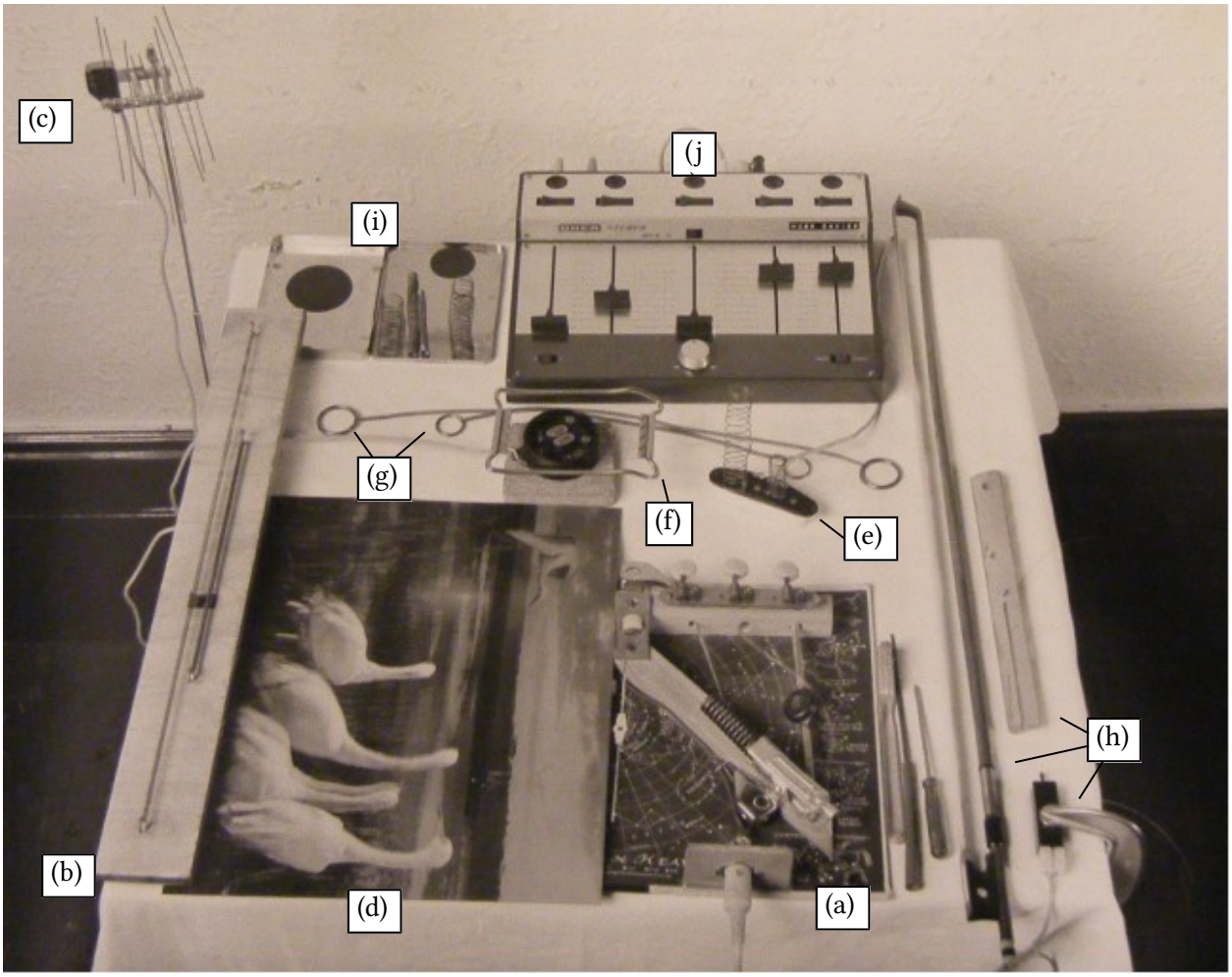


Figure 4. Solo Performance Table, incorporating: (a) Shozyg Mk. II (1968) with a range of playing implements; (b) Springboard Mk. V (1970); (c) Concert Aeolian Harp (1972); (d) 3D postcard; (e) two unstretched springs and magnetic pickup; (f) Egg-slicer and magnetic pickup; (g) Long springs with key-rings to vary their tension; (h) Guitar string mounted in gramophone cartridge, with bamboo tensioner and bow; (i) Diaphragms used in conjunction with egg-slicer, plus further springs; (j) Custom-built multi-channel mixer. Photo © Pam Davies. Courtesy of The British Library.

5. PERFORMANCE CONTEXT AND CHARACTERISTICS

Davies's self-built instruments were typically played in improvised, semi-improvised, and/or process-driven contexts. In the late 1960s and early 70s, Davies played his instruments in three different performing ensembles: Music Improvisation Company (1969-71), Naked Software (1971-3), and Gentle Fire (1968-75). Music Improvisation Company and Naked Software were both improvisation ensembles, the former somewhat jazz oriented, the latter less idiomatically-driven. (Even the name Naked Software suggests possible connections with computing.) Gentle Fire, on the other hand, specialised in performing compositions rather than improvisations *per se*, but favoured indeterminate scores that left a significant degree of interpretative freedom to the performers, or works that developed according to some kind of (as it were) 'algorithmic' process. These included works by Stockhausen, Cage, Brown, Grosskopf, Wolff, and others, as well as several of Gentle Fire's own *Group Compositions*, which were process pieces devised collectively by the members of the group (Davies 2001a; Emmerson 1991).

Gentle Fire's *Group Composition V* involved the rolling of dice to determine musical events and electronic processes, and various other physical and conceptual systems designed to constrain the action possibilities within an otherwise open musical framework. In *Group Composition IV*, it was the instrument itself that governed the process by which the music unfolded over time. This was an instrument built, not just by Davies, but by all five members of the group, comprising four large, suspended metal racks amplified via contact microphones, which all of the members of the group played simultaneously. 'The instrument is the score of what we're playing', is how one member of the ensemble described *Group Composition IV* (Michael Robinson, quoted in Plaistow 1973). It was the instrument itself that governed the actions of the performers, and hence the way the music unfolded over time; the instrument provided, as it were, the 'algorithm' driving the music's development, via an exploration of its affordances and constraints. Process in Davies and Gentle Fire's work is discussed further elsewhere (Mooney forthcoming/2012a, 2012b).

From the early 1970s onwards Davies began to perform more as a soloist, and less frequently in ensembles, but his solo performances retained the improvised, semi-improvised, or process-driven approach just described. Although the sound-world and overall structure of a performance might sometimes be planned in advance, it would never be entirely predetermined. Rather, Davies's performance practice was one of continual exploration of the musical possibilities of the instrument(s) at hand:

> What [Davies] requires of anyone who plays his instruments is that he or she should become sensitive to what the instrument is capable of doing and what is natural to it… (Roberts 1977, p.8)
>
> When he talks about his work it is noticeable that Davies constantly uses phrases like "the instrument tells me what to do", [or] "the materials show me how it should be." (Roberts 1977, p.11)

One might suppose that it was these kinds of principles that were continuously at work when Davies selected objects and playing techniques from his Solo Performance Table. One gets the impression of an exploratory framework, circumscribed by the instrumentarium's material properties and constraints, through which the music was allowed to develop as an emergent form via a continuous process of interaction between performer and instrument.

6. WORKSHOPS AND EXHIBITIONS

Davies's instrument-building practice was not undertaken solipsistically, nor purely for his own artistic gratification. On the contrary, Davies actively sought opportunities to engage a wider public, both in the playing of his existing instruments, and in the invention of new ones. Davies's frequently staged instrument-building workshops for children (Davies 2002, p.96), for example, as well as regularly exhibiting his instruments in art galleries, where members of the public would be encouraged to play them. Such activities were underpinned by a commitment to 'learning by doing', an ethos symbolised by the very first of his performance instruments, the Shozyg, which was described in the BBC's *The Listener* magazine as 'an encyclopaedia degutted to substitute direct experience for learning' (quoted in Davies 1974, p.5).

7. DAVIES'S PRACTICE AND LIVE CODING: FOUR SUGGESTIONS

For the purposes of the present discussion live coding is defined as the manipulation of computer code in a live performance to generate or influence the development of music in real time. (Live coding is, of course, practiced in other, non-musical, scenarios, but these are not directly considered here.) How might the present-day practice of live coding be related to Davies's instrument-building practice as I've just outlined? As a starting point for further discussion, four suggestions are offered.

7.1 Live Electronic Music: Historic Precedents

First and foremost, both Davies's practice and the practice of live coding are forms of live electronic music, that is, they both represent attempts to generate music by electronic means in the context of a real time performance, as opposed to producing electronic music off-stage in an electronic music studio. This might seem like stating the obvious, but in this case stating the obvious is an important thing to do, since it allows Davies's practice and live coding to be thought of as constituent parts of a broader historic trajectory. Within such a framework, one might look for precedent aspects of live coding practice, not just in Davies's work, but in other previous incarnations of live electronic music, such as those documented by Davies in his extensive article on 'Electronic Musical Instruments', published in the *New Grove Dictionary of Music and Musicians* in 2001 (Davies 2001b). For example, Davies refers to the use of 'miniaturized circuitry and

microcomputers' in the work of David Behrman, Paul de Marinis, Larry Wendt, and the California-based League of Automatic Composers (Davies 2001b, p.101), the latter of whom used KIM-1 microcomputers in live performances:

> With a debut performance in Berkeley in 1978, the League's performances consisted of each human member's KIM-1 networked together through flimsy, 8-bit parallel ports through which each device could be used to generate its own sound as well as receive and process data from other machines on the "network." (Salter 2010, p.206)

Whether or not these computers were actually *coded* during real-time performance, the simple fact of their use in a live performance setting at all represents a part of the historic precedent for today's live coding practices that is worthy of further exploration. As another possible precedent, David Tudor's *Rainforest* pieces (1968-73), which involved the use of 'instrumental loudspeakers' made of aluminium, steel, wood, and glass as 'signal processors' in various different permutations, might be thought of as a physical predecessor of MaxMSP, where multiple signal processing modules connected to one and other in an infinitely permutable combinatorial system. Might Davies's Solo Performance Table, comprising multiple self-contained instruments encapsulated within a single overall framework, be likened to a collection of object instances within an object-oriented programming framework? These examples do not share all of the features of present-day live coding practice, of course, but they do show how certain characteristic aspects—such as the use of computers in live performance, or the presence of an infinitely permutable signal-routing system—were present in forms of live electronic music that predated live coding *per se*.

7.2 Building, Making, Modifying

Second, in Davies's practice, as in live coding, it is the performer him or herself that builds (and/or modifies) the structures through which the music is mediated. Davies built his own instruments, which were then used in performance; live coders develop the algorithmic structures by which the music is mediated in real time, as the performance proceeds. On the surface of it, the idea that Davies's instruments were built before the performance, whereas in live coding the building takes place during the performance might appear to point to a fundamental distinction between Davies's practice and live coding; but does that apparent distinction really stand up to close scrutiny?

To quote Thor Magnusson in the documentary *Show Us Your Screens*, 'this requirement of starting from a clean slate [in live coding] is always an illusion' (Magnusson, quoted in McCallum and Smith, 9'44"). In live coding, there is always a part of the programming infrastructure that pre-exists the performance, whether it's the programming language itself, a higher level abstraction such as a graphical user interface, or a portfolio of functions or algorithms written in advance; whatever coding is done on-stage is simply an addendum to, or modification of, those pre-existing materials. This fact, of course, challenges the notion that live coding is concerned only with building code *during* the performance. Conversely, with Davies's instruments, it is true that parts of the instrument are built in advance of the performance, in that there is a collection of pre-existing materials that Davies brings on to the stage with him, but the ways in which those materials are combined and interacted with remains open-ended, and changes reactively as the performance proceeds, as appropriate to the musical development and the performance context. The choice of different accessories when playing the Shozyg, or the different combinations and interactions with the constituent objects of the Solo Performance Table, are examples of this. The use of screwdrivers and other such improvised addenda might even be thought of as modifications to or augmentations of the instrument itself, somewhat reminiscent of the way pre-existing code structures are modified, permutated or augmented in live coding. In other words, Davies's performance practice includes live manipulation and selection of the materials that constitute the instrument, just as much as it involves assembling those materials in advance.

In both Davies's practice and in live coding, then, there are materials that are built in advance of the performance, and modifications, selections, and augmentations of those pre-existing materials that are made in real time, as the performance proceeds. (A further sense in which Davies's practice and live coding might be thought of as similar is that they both involve the appropriation and modification of *ready-made* objects: every-day objects and household items in Davies's instruments; external function libraries and pre-written functions in live coding.)

7.3 Improvisation Bounded by Material (or Quasi-Material) Constraints

Third, both Davies's practice and live coding involve improvised, semi-improvised, and process-driven—that is, algorithmic—aspects. In live coding it is perhaps self-evident that there are algorithmic processes at work, since algorithmic processes are fundamental to all coding scenarios. Live coding also can involve an element of improvisation, though; to quote live coder Dan Stowell:

> For me it's about improvisation… How can you get towards a position where you can really improvise with the full potential of computer music, on stage? (Stowell, quoted in McCallum and Smith, 1'44")

Davies's practice, similarly, includes both improvised and 'algorithmically-driven' elements; it developed, as discussed previously, in improvisation ensembles, but also in groups that specialised in the performance of process-driven works. In some cases, as in Gentle Fire's *Group Composition IV*, the process itself was explicitly circumscribed by the material constraints of the instrument itself; a situation which might be likened to the way a live coder's actions are ultimately constrained by the characteristic or 'material' properties of the chosen programming environment.

Whether or not computer code can be considered 'material' is worth a brief diversionary discussion in the current context. According to Magnusson, code might be considered 'material' in the sense that 'it is a direct and non-ambiguous prescription of the computer's electronic gates' (Magnusson 2014, p.1); it is, in other words, material by proxy. Interactions with code, then, are in a sense material interactions by proxy. This is perhaps most clearly foregrounded in the many software packages whose graphical user interfaces use physical metaphors, such as MaxMSP, where the user connects patch-cords to objects. There are, of course, not *really* any patch-cords in the physical material sense, but the interaction is a material one functionally and metaphorically speaking, and a truly material one in the sense that it has a corresponding deterministic influence upon the computer's electronic gates. In this sense the live coder, operating within a given software environment, is bounded by the 'material' constraints of that environment.

However, irrespective of whether code can truly be thought of as material, Davies's practice and live coding are similar insofar as the music develops in real time, in an improvised or semi-improvised way, and within an overarching framework that is bounded by the constraints—material or quasi-material—of the chosen instrumentarium or programming environment.

7.4 Community Engagement and Interactive Learning

Fourth, in Davies's practice and in live coding, there is a clear desire to promote understanding through participation, which manifests itself in a distinct demonstrative, or perhaps even 'pedagogical' aspect, and in community or group-based activities with an emphasis on hands-on engagement. For example, Davies frequently staged instrument-building workshops for children, and exhibited his instruments in art galleries where members of the public were encouraged to play them. In live coding, a community-driven aspect is evident in the many open source software frameworks that are used and contributed to by live coders, or indeed developed specifically for live coding. Hands-on engagement is also evidenced in the many 'Hackspaces' and 'maker' events in which live coders sometimes participate (e.g. Leeds HackSpace 2015; Maker Faire UK 2015). All of these kinds of activities have strong agendas of learning and making, or, rather, learning *by* making.

One specific practice that both Davies and live coding have in common is the practice of 'screen sharing.' In live coded performances, it is common practice to video-project the computer screen, so that members of the audience can see how the code being typed relates to changes in the music (Sorensen and Brown 2007). To quote from TOPLAP's draft manifesto…

> Obscurantism is dangerous. Show us your screens. (TOPLAP 2011)

(*Show Us Your Screens* is, of course, the title of the documentary that has been cited twice previously.) Similarly Davies, in live performances, used to video-project images of his hands while playing his self-built instruments, 'enabling the audience to make a clearer connection between what they see and what they hear' (Davies 1997, p.13). In both Davies's practice and in live coding the video-projection is undertaken in order to facilitate audience engagement, and a better understanding of the processes by which the music unfolds.

8. CONCLUSION

The preceding discussion has focussed upon articulating areas of similarity between Hugh Davies's instrument-building and performance practice, which developed in the late 1960s and early 70s, and the present-day practice of live coding. In summary, it has been noted that both are forms of live electronic music in which the performer him or herself takes responsibility for defining the structures through which the music materialises. Both are performance practices in which the music develops in an improvised or semi-improvised way, as an iterative and reciprocal interaction between a performer and a given framework, where certain elements are fixed before the performance, and certain elements are selected, manipulated, and/or modified as the performance proceeds, but where the range of possible interactions is ultimately bounded by certain material or software-based quasi-material constraints. Finally, both are associated with hands-on, community-based activities that are designed to facilitate engagement and understanding through participation.

'Live coding is a new direction in electronic music and video', claims the TOPLAP website (TOPLAP 2011). Of course, there *is* a sense in which live coding is a new and exciting field, but there is clearly also further work to be done in situating live coding in its broader historical, cultural context. By drawing a comparison with Davies's work, it has been demonstrated that non-trivial similarities with live coding can be found in a performance practice that predated live coding as we currently know it, and involved neither computers nor computer programming *per se*. It is hoped that this paper might stimulate further thought, discussion and research as to how such an agenda might be usefully extended in future.

Acknowledgments

This research was funded by a Fellowship grant from the Arts and Humanities Research Council, in partnership with the Science Museum, London. Further details of the project of which this research forms part can be found at: http://hughdaviesproject.wordpress.com/. Thank you to Dr Tim Boon, project partner and Head of Research and Public History at the Science Museum. For their contributions to the discussion that helped shape the ideas in this paper, the author would like to thank: David Berezan, Stefan Drees, Annika Forkert, Martin Iddon, Thor Magnusson, Tenley Martin, Alex McLean, Adrian Moore, Stephen Pearse, Dale Perkins, Coryn Smethurst, and the anonymous peer reviewers of the International Conference on Live Coding 2015.

REFERENCES

Davies, Hugh. 1968a. "Shozyg." Performance instructions. Hugh Davies Collection, box "HDW 1." The British Library, Music Collections.

Davies, Hugh. 1968b. "Working with Stockhausen." *Composer* 27 (Spring 1968): 8–11.

Davies, Hugh. 1974. Biography and accompanying text for an unknown exhibition. Hugh Davies Collection, box "HDW 3." The British Library, Music Collections.

Davies, Hugh. 1979. Biography and description of work. Hugh Davies Collection, box "HDW 2", folder "Davies, Hugh. Articles on EM." The British Library, Music Collections.

Davies, Hugh. 1997. "Invented Instruments and Improvisation." *Avant: Jazz, Improvised and Contemporary Classical Music* Spring 1997: 12–15.

Davies, Hugh. 2001a. "Gentle Fire: An Early Approach to Live Electronic Music." *Leonardo Music Journal* 11: 53–60.

Davies, Hugh. 2001b. "Electronic Instruments." In *The New Grove Dictionary of Music and Musicians*, edited by Stanley Sadie and John Tyrrell. London; New York. Macmillan: 67–107.

Davies, Hugh. 2002. *Sounds Heard*. Chelmsford. Soundworld.

Emmerson, Simon. 1991. "Live Electronic Music in Britain: Three Case Studies." *Contemporary Music Review* 6/1 (January 1991): 179–95.

Klapper, Martin. 1991. "Visiting Hugh Davies." *Other Sounds*. London. TVF. TV broadcast, available online at: https://www.youtube.com/watch?v=wPT9A0IsGgs/ [accessed 9 May 2015].

Leeds HackSpace. 2015. "Leeds HackSpace: A Maker and Hacker Space in Leeds." http://leedshackspace.org.uk/ [accessed 9 May 2015].

Magnusson, Thor. 2014. “The Materiality of Code || Code as Literature.” Presented at *Musical Materialities* conference. Sussex.

Maker Faire UK. 2015. “Maker Faire UK: a two day festival of hackers, crafters, coders, DIYers and garden shed inventors.” http://www.makerfaireuk.com/about/ [accessed 9 May 2015].

McCallum, Louis, and Davy Smith. 2011. *Show Us Your Screens*. https://vimeo.com/20241649.

Mooney, James. Forthcoming/2012a. “Technology, Process and Musical Personality in the Music of Stockhausen, Hugh Davies and Gentle Fire.” In *The Musical Legacy of Karlheinz Stockhausen*, edited by Imke Misch and Morag Grant. Saarbrücken. Pfau-Verlag. Awaiting publication; currently available at http://eprints.whiterose.ac.uk/80572/.

Mooney, James. 2012b. “Process in Gentle Fire’s Group Compositions.” Presented at *Music and/as Process* conference, University of Huddersfield. http://eprints.whiterose.ac.uk/80580/.

Mooney, James. 2015a. “Hugh Davies’s Electronic Music Documentation 1961–1968.” *Organised Sound* 20/1: 111–21.

Mooney, James. 2015b. “Hugh Davies’s Self-Built Instruments and Their Relation to Present-Day Electronic and Digital Instrument-Building Practices: Towards Common Themes.” Presented at the *International Festival for Innovations in Music Production and Composition (IFIMPAC)*, Leeds College of Music, 13 March 2015. http://eprints.whiterose.ac.uk/84316/.

Plaistow, Stephen. 1973. Interview with Gentle Fire. Hugh Davies Collection, C1193/35. The British Library, Sound Archive.

Potter, Keith. 2005. “Hugh Davies: Iconoclastic Innovator in Electronic Music.” *Independent*, 7 January 2005.

Roberts, David. 1977. “Hugh Davies: Instrument Maker.” *Contact* 17(Summer 1977): 8–13.

Roberts, David. 2001. “Davies, Hugh (Seymour) (ii).” *New Grove Dictionary of Music and Musicians*. London. Macmillan: 61–2.

Salter, Chris. 2010. *Entangled: Technology and the Transformation of Performance*. MIT Press.

Sorensen, Andrew and Brown, Andrew. 2007. “aa-cell in Practice: An Approach to Musical Live Coding.” *Proceedings of the International Computer Music Conference*.

TOPLAP. 2011. “About Live Coding and TOPLAP.” *TOPLAP* website. http://toplap.org/about/ [accessed 28 January 2015].

Performative Code: Strategies for Live Coding Graphics

Shawn Lawson
Rensselaer Polytechnic Institute
lawsos2@rpi.edu

ABSTRACT

Performing real-time, live coded graphics requires a streamlined programming environment, efficient implementation techniques, improvisatory inventiveness, a tuned ear, and above all, aesthetic sensibility. The performer must not only pull together these concepts, but maintain an awareness and forethought, of the graphical results of and performance expectations of the live coding environment.

1. Introduction

Live coding is a unique and extend-able area of computer-based research and artistic practice, as evidenced by the wide variety of integrated development environments (IDEs) that currently exist for live coding.[1][2][3]

This text includes observations, thoughts, and discoveries the author has made over the past year of developing and performing with his own live coding IDE.[4] Specifically, this text addresses the challenges the live-coding graphic artist faces when collaborating with an audio performer on a set piece, including primarily the necessity for for an efficient and accurate coding environment. The IDE, which runs in Google Chrome, uses the OpenGL fragment shader language, features autocompilation and execution, has a customizable autocompletion engine, real-time FFT analysis and Open Sound Control (OSC) connectivity, and supports multi-output projection-mapping. Code writing and editing occurs in a text editor layer on top of the computed visual output.

While the autocompilation and autocompletion features contribute to the performer's speed and flexibility in a live situation, one must be aware of code artifacts, lines or blocks inadvertently uncommented or simply unintended, must be taken into account. While introducing the potentials for interesting discoveries, these artifacts may also produce undesirable effects, and by understanding these fallibilities to the degree that their implementation, expected or not, can be successfully integrated, the live coder expands his or her aesthetic toolbox beyond what is already known.

2. Language Hacks

The OpenGL shader language (GLSL), which executes directly on the graphics hardware, not sandboxed by another environment, is syntactically strict. Below are several low-level techniques that have become integral my performance practice; techniques that have improved aesthetic flexibility and coding speed, while satisfying the aforementioned syntactical requirements. These descriptions are specific to GLSL, but their concepts are applicable to other IDEs and languages.

2.1. Safety First

It is common practice to initialize a value at variable declaration, not only to prime each variable for instantaneous implementation, but to eliminate the potential for bugs and unexpected results. In the code sample below the value of color1 is unknown, whereas the value of color2 is black. In its current, undefined instantiation, adding *red* to color1, would likely return an unknown, potentially jarring result depending on the context. Color2, which is initialized to be black, can more accurately predict the result of adding red: black becomes increasingly red.

[1] "Live Coding." 2015. http://en.wikipedia.org/wiki/Live_coding
[2] "Software." 2015. http://toplap.org/category/software/
[3] "ToplapSystems." 2015. http://toplap.org/wiki/ToplapSystems
[4] Link to IDE *The_Force*: http://shawnlawson.github.io/The_Force/ Link to github repository: https://github.com/shawnlawson/The_Force

```
//vec4(red, green, blue, alpha) with 0.0 - 1.0 range.
vec4 color1;
vec4 color2 = vec4(0.0, 0.0, 0.0, 1.0);

color1 = color1 + vec4(1.0, 0.0, 0.0, 1.0);
color2 = color2 + vec4(1.0, 0.0, 0.0, 1.0);

//  unknown output color
gl_FragColor = color1;
//  output color will be black plus red
gl_FragColor = color2;
```

Furthermore, if the a performer decides to delete the line of code that adds red to color2, the outgoing fragment color would revert back to the initialized value of black, thereby reducing the potential for rogue values to modify the color if that line is implemented in some other process(es). Similarly, if the performer deletes the line of code that adds red to color1, the resultant value is unknown, and may lead to an undesirable, or at least unexpected outgoing fragment color.

2.2. Commenting

A simple method of working within the functionalities of an auto-compiling and executing IDE is to write code behind a single line comment maker (see example below).

```
//line of code written, ready to be uncommented
//vec4 color1 = vec4(1.0, 1.0, 1.0, 1.0);
```

It has proven, effective to implement a key command into the IDE to toggle comment/uncomment on the current line(s). The potential pitfall of this functionality is that if the code behind the comment marker has errors, they won't be apparent until after the uncommenting (which will be addressed in the next subsection), but as a process for hiding/revealing code from/to the compiler quickly, it is generally effective. Specific to GLSL, if lines or blocks of code are not directly needed then commenting out portions of code can have large performance improvements, specifically regarding frame rate and rendering resolution, but also legibility.

As in most programming environments, comments can be used to organize and structure. An IDE with a color syntax highlighter on a static background color will substantially reduce the performer's visual search for specific code locations. When the code lengthens to the degree that it scrolls off the screen, a higher level of search can be implemented using single line comments. In the comment below, the performer can do a quick visual scan of the reverse indented, color-coded comments, then narrow the search to the specific code desired, as opposed to the performer hunting through the syntax-colored code on what may be a similarly-colored, animated background. In the code example the comment color is light blue whereas in the IDE it is grey, causing it to stand out more, not dissimilar to a written text's section headings that may be in bold and larger font. The offset and color difference simply makes the comment easy to find and mentally parse.

```
    vec4 color1 = vec4(1.0, 1.0, 1.0, 1.0);
    vec4 color2 = vec4(0.0, 0.0, 0.0, 1.0);

//H Bars
    color1 = color1 * step(.4, fract(gl_FragCoord.x * .1));
    color2 = color2 * step(.6, fract(gl_FragCoord.x * .2));
```

2.3. Conditional Help

As previously mentioned, there are potential issues with the single line commenting method. Below is a workaround to those potential pitfalls that subtly exploit the auto-compiler.

```
vec4 color1 = vec4(0.0, 0.0, 0.0, 1.0);
```

```
if (false) {
    color1 = color1 + vec4(1.0, 0.0, 0.0, 1.0);
}
```

Using a conditional statement that always evaluates false allows the embedded code to be auto-compiled, but prevents it from executing. This encapsulation within a conditional statement provides the performer with a safety net for building complex or detailed code sections without worry for inadvertent or premature compilation. When successfully compiled, the performer can utilize the comment method to hide this code from the compiler, and safely remove the conditional wrapper (this is expanded in the snippets section).

2.4. Decimal Cheats

During a performance, the performer may want to create a transition or continuously change a value. Slowly increasing a value from 0.1 to 0.9 is quite easy; whereas, changing from 0.9 to 1.0 is not. To change the leading 0 to 1 would result in an intermediate value of 1.9 before being able to change the 9 to 0. A similar problem exists in the reverse direction. To change the trailing 9 to 0 would result in an intermediate value of 0.0 before being able to change the leading 0 to 1. Both 1.9 and 0.0 are quite far out of sequence from our value of 0.9.

The cheat is to intentionally commit an error, then rely on the autocompiler's memory. When an error occurs, the autocompiler retains the most recent successfully compiled shader. The performer simply inserts a second decimal point, in this example, next to the existing decimal, resulting in 0..9. Changing numbers on either side of the decimal can now happen without consequence. Once we have our desired value of 1..0, the extra decimal point is removed resulting in a final value of 1.0, which the autocompiler will accept (an additional method towards continuous change is addressed in the external input section).

3. Linguistics

```
vec4 white = vec4(1.0, 1.0, 1.0, 1.0);

//each line below has one error
vec4 color1 = whie;
vec4 color2 = vec4(1.0, 1.0, 1.0 1.0);
```

It is much easier to see the letter "t" missing from the word "white" in the declaration of color1, than to find the comma missing after the third parameter in color2. When performing, typing and text scanning are virtually simultaneous, and without considerable legibility error resolution becomes increasingly difficult. Anxiety and stress mounts when while tracking down errors, because a set performance piece continues; gradually, one falls behind and misses cues.

Debugging color2, under some time pressure, we would mentally check type (vec4); the operation (=); the parameters (count of 4 matches vec4), values (1.0 1.0 1.0 1.0), type (float float float float), separated (yes yes no yes) - finally we have the problem. And, it is a very detail oriented problem, which is not an ideal situation.

In the case of color1, we would mentally check type (vec4); the operation (=); the value (whie) - spelled incorrectly. We're more easily able to locate the spelling mistake than to dig down into the nitty-gritty.

To conclude this sub-section, one is less error-prone when using "white" than "vec4(1.0, 1.0, 1.0, 1.0)", so long as "white" is correctly defined. Plus readability and comprehension are much faster. Predefining commonly utilized variables like colors and functions, as we will discuss in the next subsection, is one way to achieve this.

3.1. Helper Functions

The development and utilization of helper functions aid with the overall performance and help to maintain a graphical pacing and aesthetic interest. In the case of GLSL, a random function does not exist in many Graphics Processing Unit (GPU) hardware implementations. To write a random function for the performance may certainly have some appeal if it were integral to the primary concept of the performance; however, when needed solely to have some random data to work with, the writing out of helper functions can consume valuable time. A resolution to the issue was to include a select set of often used helper functions into the IDE for random numbers, shapes, and sprite sheet animation.

4. Debugging

While the integration of a step through debugger may be an interesting aspect of one's performance, should the debugging process drag-on too long, the flow of the visual experience may be compromised. In order to avoid a protracted debugging session, a graceful, scale-able debugger can be an excellent addition. One method of subtle error indication is to make a slight change to the line's background color (see Figure 1).

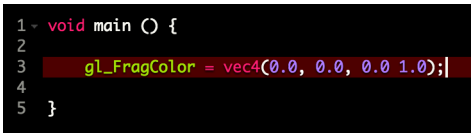


Figure 1: *Example of a debugger highlighting a line of code that has an error. Note, the missing comma again.*

Should more assistance or information about the error be needed to resolve the issue, a slightly more intrusive technique can be used. In figure 2, error markers are placed in the line number column. Additionally, when the cursor hovers over one of these markers a debug message is displayed. There is some efficiency lost with this method; because, the performer must pause typing, move the cursor, read, move the cursor to hide the message, and finally start to type again.

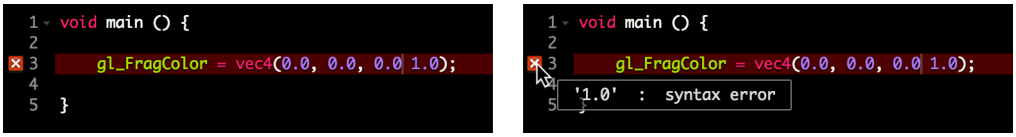


Figure 2: *Left: Example of an error marker. Right: Revealed debug message.*

An even more visually intrusive method would be an in-line debug message information system (see Figure 3). While this example is visually disruptive, especially when it shifts down lines of code when inserted, it is significantly faster for the performer to see the debugging information and make a correction.

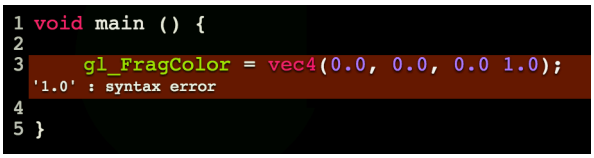


Figure 3: *Example of a debugger providing additional information, in-line to the code. Note, again, the missing comma.*

All of these approaches detract less from the overall show experience by avoiding a separate panel with a GDB style breakpoint debugger. And, as with most things, with experience and practice, these assistive debugging indicators can be scaled back as the performer gains confidence and proficiency.

5. Autocompletion and Snippets

Many autocompletion engines exist in many forms, but it is important that they are *configure-able* for one's specific needs: accuracy and speed.

5.1. Reducing and Appending

The OpenGL specification for WebGL contains many built-in OpenGL functions. By default, many autocompletors will contain all the OpenGL functions, storage classes, built-in math functions, reserved words, etc. To reduce clutter from the autocompletor, anything relating to the vertex shader is removed. Furthermore, anything that is not commonly used is removed. Predefined variables and functions are included in the autocompletor list, for example: colors, shapes, helper functions, and uniform names. This custom reduction and addition creates a highly optimized list, meaning that in most cases our desired autocompetion match is reduced to a single solution faster.

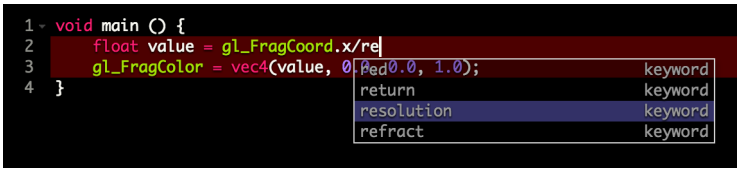


Figure 4: *Example of autocompletor showing possible known completions.*

5.2. Defining Snippets

Snippets are generally small code blocks that are entered to save time or reduce errors. The snippets are written by the user into a configuration file. The autocompletor's engine builds snippets from the file as seen in the code example below. The first line is a comment that appears when the autocompletor's snippet pop-up appears. The second line starts with the *snippet* declaration, followed by the snippet's *key*. The third line is the snippet's *value*. This particular snippet enters the circle helper function. To breakdown further, *${}* indicates the location of a parameter, and where the next tab-jump point is. The number inside the curly braces gives the tab-order of the parameters. The text following the colon will be the default value for those parameters. In this case, the word *float* is a hint as to the type of value that needs to be entered. Note the fifth tab-point was added as a quick way to jump outside the parenthesis.

```
# circle
snippet cir
    circle(${1:float}, ${2:float}, ${3:float}, ${4:float})${5}
```

In figure 5, the circle fuction has been autocompleted. The first value is highlighted, meaning it will be overwritten when typing begins. Pressing "tab" will jump to the next "float" parameter and highlight it for overwriting.

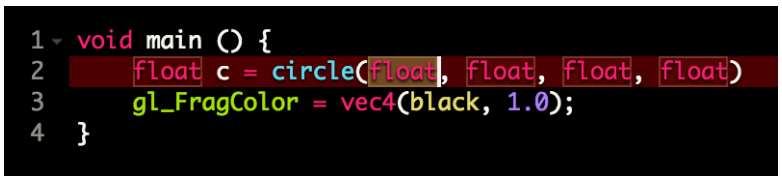


Figure 5: *Example of autocompletor completing the circle function snippet.*

5.3. Common Snippets

Some testing has led to the implementation of a variety of snippets; the following are a few of the key:value pairs integrated into the IDE. The first example is a for loop with conditional check against an integer (a float version also exists with the *forf* snippet key). The second example, *iff*, we recognize from the conditional help section above. Last is a selection of commonly used data types.

```
//snippet fori
for (int i = 0; i < int; i++) {

}

//snippet iff
if (false) {

}

//snippet vv
vec2
//snippet vvc
vec3
//snippet ft
float
```

6. External Input

Many audio visual performers use hardware input devices to launch processes, activate functions, adjust values, or simply inject a sense of physicality into the live performance. In a live coding performance, additional hardware devices can similarly help to improve efficiency, facilitate real-time and continuous access to parameter values, and inject a more perceptible layer of physicality. Additionally, performers frequently configure their software to communicate with each other. This inter-performer communication adds a rich cohesiveness.

For this IDE, the Open Sound Control (OSC) protocol was implemented. This was chosen due to a familiarity with OSC, a need to receive OSC messages from both hardware (iPad with Lemur[5]) and other performers, and in preparation for the future possibilities that may include a variety of data types and ranges.

How might this be used? Let's say that an OSC device produces data that ranges from 0.0-1.0 and is coming into an GLSL shader as a uniform labeled *fader*, see the code example below.

```
uniform float fader;

void main() {
    vec3 black = vec3(0.0, 0.0, 0.0);
    vec3 white = vec3(1.0, 1.0, 1.0);
    color = mix(black, white, fader);

    gl_FragColor = vec4(color, 1.0);
}
```

With an external OSC controller, a performer could blend or mix between the black and white colors. This method could be applied to back buffers for adding motion blur, mixing effects, cross-fading between effects, or provide more accurate cuing with audio transitions. While seemingly trivial, in the above example, this OSC technique provides a stream of continuous numbers whereas the deliberate typing of numbers into the autocompiler may have very jarring or unexpected/undesired outcomes. This input method resolves the 0.9 to 1.0 discontinuity problem as mentioned in the decimal cheats section.

7. Conclusion

Live coding introduces a particular set of technical, performative, and aesthetic challenges. While this is certainly not intended to be a comprehensive survey of these issues and their potential solutions, it is meant to contribute to the already emerging dialogue, framed primarily by the author's work. To this end, the aforementioned strategies for live coding graphics are applicable to other areas of live coding.

8. Acknowledgments

Thank you to my audio collaborator, Ryan Ross Smith.[6] And, thank you to CultureHub for providing us the time and space to continue building the *Sarlacc* set.[7][8]

[5] App interface building and performance IDE. https://liine.net/en/products/lemur/
[6] http://ryanrosssmith.com
[7] http://www.culturehub.org
[8] Video link to *Sarlacc*: https://vimeo.com/121493283

Social Imagination

Carolina Di Prospero
(IDAES- UNSAM)
diprosper@gmail.com

ABSTRACT

This is a short paper that proposes an approach to the activity of live coding as an artistic configuration constituted in a creative practice from improvisation, openness and constant exploration. I just want to share some thoughts about sociability in live coding, in terms of "imagined community" (Anderson 1991) to address this collective aspect.

The approach is anthropological, through ethnographic field work from which the method seeks to explore some combination between a scope, actors and activities and a cut of reality that encompasses practices, values and formal rules. The aim of the ethnography is to address the distinction: "between the real and the ideal culture, between what people do and what people say they do, and hence between the field of practices, values and rules" (Guber 2001).

This work seeks to provide some characterization of a collective artistic expression in constant process, which mediates and constitutes sociability and subjectivities in a sociotechnical context.

1. INTRODUCTION: COLLECTIVELY IMAGINING

In live coding activity, there is a core intention to explore the capabilities to skillfully improvise with code in a challenging way, as in the following field testimony:

> I like the idea of being able to make music using lines of code, and am fascinated about the kind of dexterity and variety of sounds which can come from the simplest bits of code. (testimony A)

But, there is also an intention of further developing a kind of participatory community, in which everyone can participate without being required programming mastery, as Alex McLean explains to Dazed and Confused magazine[1]:

> Many live coders make and adapt their own programming environments: that takes some experience. But proficiency at coding dance music is different to making financial systems or whatever. I've run workshops where I've got non-programmers making acid house together in a couple of hours. I think there's real possibility to make producing algorave music more like drumming circles, where beginners can just join in and learn through doing.

Live coding activity arises from the start as a collective activity, both keeping that daily interaction with each other in the mailing list, through the publication / socialization of its programming languages or during the performance by opening a connection with the audience, as a live coder told me:

The idea is to show and share your compositional thoughts with the audience (testimony B)

In order to ascertain the participatory intention, I would like to refer to sociability in terms of "imagined community" (Anderson 1991). According with Benedict Anderson, all communities larger than primordial villages of contact are imagined. Communities do not be distinguished by their falsehood or legitimacy, but by the style in which are imagined (Anderson 1991) then, in this sense, there would be an idea and a collective construction, from that idea.

Often, in shared conversations, interviews and small talks with live coders, many of them refer to the idea of a cultural growth, in different ways:

[1]Full article in: http://www.dazeddigital.com/artsandculture/article/16150/1/what-on-earth-is-livecoding.

> A participatory community, contribute to a culture that creates a positive environment for everyone who wants to participate, regardless of country of origin and gender. It is about being open and generate closeness and inclusion (testimony C)
>
> ////// In the UK there are only a few pockets of coders scattered around. This makes it hard for any kind of live coding scene to gather momentum. When I play a gig, I'm the oddball using a computer and typing to make music on the stage in between 2 rock bands. When playing the Algoraves I met the guys from Mexico City and they talked about how they had their own scene out there and it was like a family, putting gigs on and creating a community together. If these little pockets happen more often, then the future of live coding can only get stronger. (testimony D)
>
> I think the development of languages that become more spatial, collaborative, social, expressed in a wider range of ways, and generally more like a human language. I think the growth of localized live coding communities, are really nice to see, and will make things really real. (testimony E)

For anthropologist Georgina Born, however music has no material essence, it has a plural and distributed materiality. Its multiple simultaneous forms of existence - as sonic trace, discursive exegesis, notated score, and technological prosthesis, social and embodied performance – indicate the necessity of conceiving of the musical object as a constellation of mediations:

> Music requires and stimulates associations between a diverse range of subjects and objects – between musician and instrument, composer and score, listener and sound system, music programmer and digital code. Compared with the visual and literary arts, which we associate with a specific object, text or representation, music may therefore appear to be an extraordinarily diffuse kind of cultural object: an aggregation of sonic, social, corporeal, discursive, visual, technological and temporal mediations – a musical assemblage, where this is understood as a characteristic constellation of such heterogeneous mediations (Born 2011).

These mediations take place in four levels of social mediation: (1) music produce its own and diverse sociabilities; (2) music has the capacity to animate imagined communities, aggregating its adherents into virtual collectivities and publics based on musical and other identifications; (3) music refracts wider social relations and (4) music is bound up in the social and institutional forms that provide the grounds for its production, reproduction and transformation (Born 2011). For Hennion, the work of art as a mediation means to review the work:

> every detail of gestures, bodies, customs, materials, space, languages, and institutions that it inhabit. Styles, grammar, taste systems, programs, concert halls, schools, business… Without all these accumulated mediations, beautiful work of art does not exist (Hennion 2003).

As the author emphasizes, the work of mediation involves to stop attributing everything to a single creator and realize that creation is much more widely distributed, and is performed in all the interstices between these successive mediations (Hennion, 2003: 89).

Live coding has been constituted as a collective artistic expression that mediates and builds on sociabilities and subjectivities in a socio-technical context, then, Born´s four levels would be given by socio-technical mediations in the case of the live coding scene, because when technology is not only appropriate but, is being experienced by the people, is in turn built and the own experience with technological devices is what makes sense (Bijker 1987).

2. NEW MEANINGS IN PRACTICE

If for many live coders to reach dialogue with computers through live coding or develop new programming languages has new senses, like a political one, for example, from the proposal to achieve a dialogue with software, or software shaped as software art, or to bring programming to diverse audiences, showing the screens during the performance:

I like to think that the Toplap statment of "show us your screens" helps the laptop musician demistify the processess of what he/she is doing. It also gives the viewer a way in to what the coder is doing (testimony F).

To create a participatory community also has new meanings. For Pierre Bourdieu, “the categories of perception of the social world are essentially the product of the incorporation of the objective structures of the social space” (Bourdieu 2002), now, what happens when people try (from theory and practice) a profound change in these objective structures? Bourdieu points in this sense that “knowledge of the social world and the categories which make it possible, are the stakes of the political struggle, a struggle which is inseparably theoretical and practical, over the power of preserving or transforming the social world by preserving or transforming the categories of perception of that world” (Bourdieu 2002). Here inclusion and openness, so valued and promoted by the live coders, and the desire to create a "participatory community" can lead us to an analysis that takes into account the liminality of an “imagined community”.

A search beyond a cultural and ideological field of production, is for Victor Turner a “communitas” (Turner 1969). The concept of communitas can be helpful at this point, Victor Turner has used it to talk about "open society, which differs from the structure or closed society" (Turner 1969). The author chooses to speak of communitas instead of the community because:

> For both, individuals and for groups, social life is a kind of dialectical process involving successive experiences of ups and downs, communitas and structure, homogeneity and differentiation, equality and inequality. (...) In this process the opposites of each other and are mutually indispensable (...) In other words, each individual life experience has alternate exposure to structure and communitas, as states and transitions (Turner 1969).

Transitions where usually appear figures, representations, as signs of moral values of communitas, are opposed to coercive power of the supreme political rules, explains Turner, because from the structure all manifestations of communitas appears as dangerous and anarchic (Turner 1969). The process of liminality is negative for primary social structure, and a statement of another order of things and relationships as well. Communitas, he explains, is the product of individual human faculties including rationality, will and memory, which break through the interstices of the structure, in liminality, in experiences of unpredictable potential:

> Liminality, marginality, and structural inferiority are conditions in which are frequently generated myths, rituals, symbols, philosophical systems and works of art. These cultural forms provide men with a set of templates or models which are, at one level, periodical reclassifications of reality and man´s relationship to society, nature and culture. But, they are more than classifications, since they incite to action as well as to thought. Each of these productions has a multivocal character, having many meanings, and each is capable of moving people (Turner 1969).

Liminality is present in some ways in live coding: new projects and proposals, the search to demystify the relationship with technology, making the code a craft or artistic product, but, more than anything, in the construction of its "participatory community", a collectively imagined community. Liminality of space to express themselves and build various proposals raises transformations not only in the artistic or cultural field but also institutional, the live coding scene involves building an entire world, an art world in terms of Becker (Becker 1982). According to the author, who cooperates in producing a work of art do not do it from nothing but rest on past agreements or custom / conventions, which usually cover the decisions to be taken, and this makes things simpler (Becker 2002). However, Becker explains that people can always do things differently, if they are prepared to pay the price:

> In general, breaking with existing conventions and their manifestations in social structure and material artefacts increases artists´ trouble and decreases the circulation of their work, but at the same time increases their freedom to choose unconventional alternatives and to depart substantially from customary practice. If that is true, we can understand any work as the product of a choice between conventional ease and success and unconventional trouble and lack of recognition (Becker 2002).

The increasing of such problems Becker mentioned, a kind of output that live coders found to this difficulty in building their "art world" was to place their art in the process more than in a finished product. Musical improvisation helps to build another perspective from both, the programmer and programming languages. The emphasis on process, in which materials, digital and analog, are more important than materiality, or a final product allowed live coders to advance in the construction of their activity, and their world, always changing, always exploring the role of technology in art and art in their technological forms. It is there, in

the construction of those environments in process, where live coders feel creative and create from improvising, in the space of active materials.

"The materials are active components of a world-in-training. Wherever life is happening, they are constantly in motion - flowing, scraping, mixing and mutating" (Ingold 2013) in the case of live coding, the materials flow in direct relationship with the artist in the act of experience.

Ingold and Hallam say the difference between improvisation and innovation is that the first characterizes creativity by way of their processes (movements), while the second does so by way of their products (results) (Ingold & Hallam 2007), in this sense we can say live coding is expression and movement, which, in any case raises an end in the process itself.

3. CONCLUSIONS

Regarding the social aspect or social settings from artistic practices: "art worlds" (Becker 1982) if, as Howard Becker explains, conventions make it easier and less costly to bouil an art wolrd, but more expensive and difficult to make deep changes (Becker 1974), then, the case of live coding broadly contributes to the acceptance of change as a constant, within a framework in which artistic expression is a process rather than finished product (Di Próspero 2015). The live coding scene makes the change, openness and constant exploration practices that constitute a creative activity.

REFERENCES

Anderson, Benedict. 1991. *Comunidades Imaginadas. Reflexiones sobre el origen y la difusión del nacionalismo*. Mexico. Fondo de Cultura Económica.

Becker, Howard S. 1974. "Art as a Collective Action. Philadelphia." *American Sociogical Review*. 39: 767-776.

Becker, Howard S. 1982. *Art Worlds*. Berkeley. University of California Press.

Becker, Howard S. 2002. "*Art Worlds"* Cultural Sociology. 178-183. Oxford. Blackwell.

Bijker, Wiebe. E., Hughes, Thomas. P. y Pinch, Trevor.1987. *The Social Construction of Technological Systems. New Directions in the Sociology and History of Technology*. Cambridge, MA. MIT Press.

Born, Georgina. 2011. "Music and the materialization of identities". *Journal of Material Culture*. 1-13. UK. SAGE.

Bourdieu, Pierre. 2002. "Cultural Power". Cultural Sociology. 69-76. Oxford. Blackwell.

Di Próspero, Carolina. 2015 "Live Coding. Computer art in Process". Revista Contenido. Arte, Cultura y Ciencias Sociales. N. 5, 44-62. http://www.revistacontenido.com/live-coding-arte-computacional-en-proceso/

Guber, Rosana. 2001. *La Etnografía: método, campo y reflexividad*. Buenos Aires. Norma.

Hennion, Antoine. 2003. "Music and mediation: Towards a new sociology of music". The Cultural Study of Music. A Critical Introduction, 80-91. London. Rutledge.

Ingold, Tim, Hallam, Elizabeth. 2007. *Creativity and Cultural Improvisation*. New York. Berg.

Ingold Tim. 2013. “Los Materiales contra la materialidad”. *Papeles de Trabajo*. 11 (7): 19-39.

Turner, Victor 1969. “Liminality and Communitas”. *The Ritual Process: Structure and Anti-Structure*. Chicago. Aldine Publishing.

Live Writing: Asynchronous Playback of Live Coding and Writing

Sang Won Lee
University of Michigan, Ann Arbor
snaglee@umich.edu

Georg Essl
University of Michigan, Ann Arbor
gessl@umich.edu

ABSTRACT

We introduce Live Writing, asynchronous playback of a live coding performance or, more generally, writing. The concept of Live Writing is realized in a web-based application which logs every keystroke that a writer makes and let the writing later playable by the audience or the readers in real-time. The goal of Live Writing is twofold. One, it aims to support collaboration between musicians by reproducing a live coding performance based on keystroke data logged in the platform. This offers a new way for a live coder to archive live coding music and to communicate with others in asynchronous fashion. Two, it aims to transform written communication into a real-time experience so that a writer can display a writing to readers in a real-time manner as if it is being typed in front of the readers. We explain the design and the implementation of the system and demonstrate two different use cases of the system: live coding and writing.

1. Introduction

Live coding music is a type of written communication between human and machine expressed in programming language. At the same time, the algorithmic representation of music is a window into a composer's thoughts. Therefore, in the case of collaborative live coding, code sharing is an important part of communication, understanding each other's thoughts, and exchanging ideas. Most of the times, the collaboration in this live coding practice occurs in real time, due to its live nature and asynchronous communication being limited. However, being able to archive and replay live performances can further expand and enable important aspects of live coding performance culture. Benefits include the ability to (1) collaborate asynchronously, (2) incorporate last live performances into reinterpretation, reappropriation, and the creation of a persistent performance corpus. Static code is an core outcome of expressing compositional ideas that arise throughout the preparation process of a live coding performance. Nonetheless, program code itself does not represent the entire progress of the music performance. Static code does not contains temporal information of code execution nor captures a performer's virtuosity, such as skills and techniques to utilize existing code quickly enough to make changes in the music in a timely manner. In addition, some code written on the fly may not even exist in the final code that remains after the performance. While the screen recording of a live coder's play may capture these kinds of real-time gestures, it is fixed media in a non-symbolic format. Hence, it is impossible to get textual data from the recording so as to copy and transform, for example, someone else's code. This calls for representations that that capture the dynamic and evolving character of live code production in a format that is readily modifiable. To this end, we developed the Live Writing platform. The Live Writing platform logs all keystrokes made by a live coder with timestamp information in order to reproduce the musician's play later. The system can be used to archive any live coding performance ranging from individual practice to actual performances in public. The term "asynchronous" collaboration in "live" coding sounds contradictory. However, it seems potentially exciting to envision a live coding system such as one that will facilitate asynchronous collaboration. In other words, how do we utilize technologies to help a musician communicate with other musicians without needing to be present at the same time? As an extreme example, how could a live coder reproduce a piece that has been written by a live coder who has since passed away. Naturally, this poses two questions: how to notate a live coding piece and how to archive a live coder's playing.

Our second goal with Live Writing is to transform, as the name suggests, general written communication into a real time experience. Writing is rich form of communication and we live in an age when we produce large volumes of writing through digital platforms such as the World Wide Web, and mobile devices. The Live Writing platform lets writers (not necessarily just live coders and artists but anyone) present their writing to readers in the same way that it was typed at the time of creation. The real-time rendering of writing gives much more expressive power to the writer and gives readers the intimacy of peeking at someone's computer screen in private. The core idea of Live Writing is to reveal the incomplete stages of writing and of the textual dynamic as part of the written expression. For a quick demo of Live Writing, visit http://www.echobin.com/?aid=aboutechobin. A screenshot of Live Writing platform is shown in Figure 1.

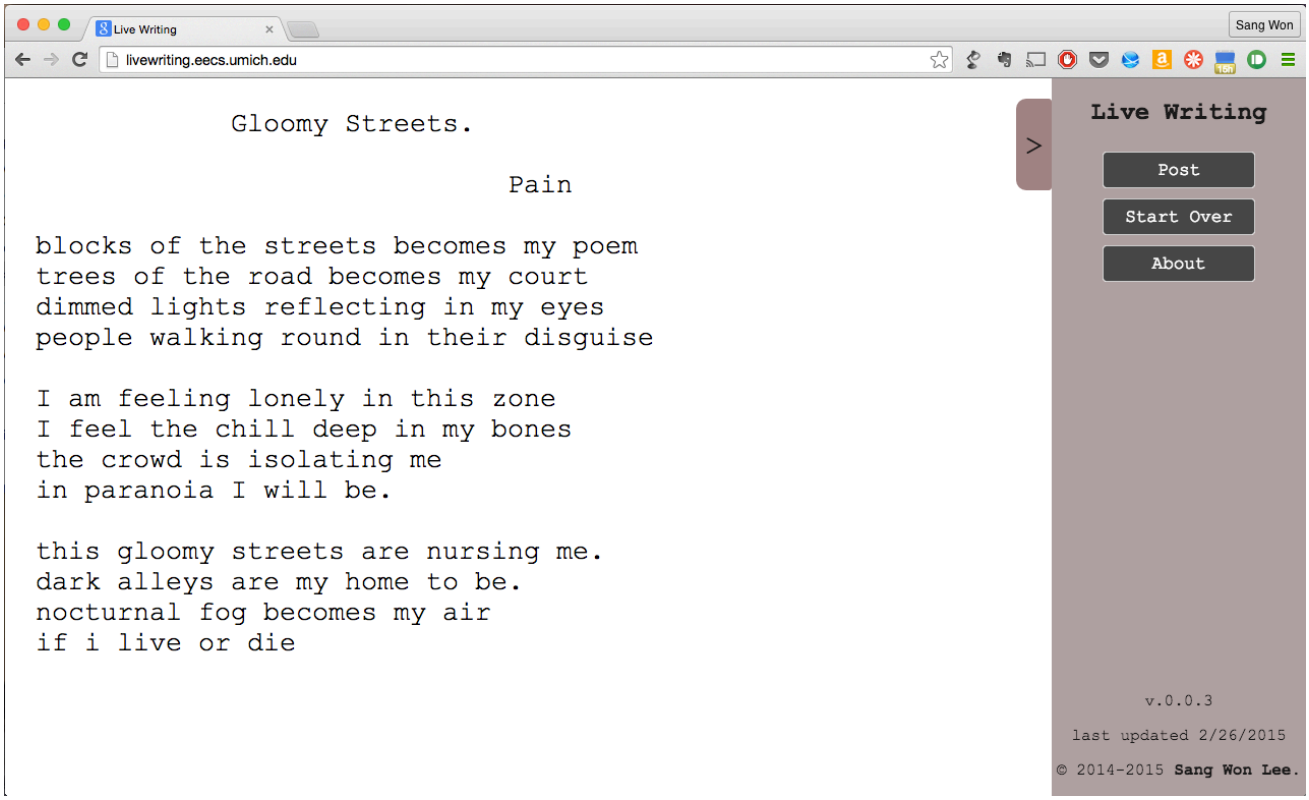


Figure 1: *a screenshot of Live Writing example. The poem written by Pain.*

In this paper, we introduce the background upon which the idea is built. We further motivate the core ideas, and describe the current implementation. Demonstrated on the web browser are examples in the context of live coding as well as writing. Finally, we conclude with future research opportunities offered by the system.

2. Asynchronous Collaboration in Live Coding

2.1. Live Coding as Network Music

Networked communication and data sharing can facilitate collaboration among live coders. The potential of networked collaboration in live coding has been present from the inception of live coding (Collins et al. 2003). *Power-Books_UnPlugged* exemplifies an interconnected network ensemble by sharing code over a wireless network among live coders and enabling sound dislocated from the code (Rohrhuber et al. 2007). Similarly, *aa-cell* added network capability to Impromptu to facilitate collaboration over the local network for sharing and executing code remotely (Brown and Sorensen 2007). The scale of collaboration in live coding has reached that of an ensemble (Wilson et al. 2014) and a laptop orchestra (Ogborn 2014c). Recently, the authors made an extension to *urMus* (Essl 2010), a programming environment for mobile music, to support multiple live coders, addressing issues that emerged for collaborative live coding in general (Lee and Essl 2014a).

In our previous work, we applied existing frameworks coming from the tradition of network music to the previous works of networked collaboration in live coding (Lee and Essl 2014b). For example, we used Barbosa's framework to classify computer-supported collaborative music based on synchronism (synchronous / asynchronous) and tele-presence (local / remote), as shown in the Figure 2 (Barbosa 2003). This is used to learn opportunities in live coding and to identify relatively underdeveloped areas in the classification. Most networked live coding performances fall into the local/synchronous category, where live coders are co-located and play at the same time. Recently, researchers explored remote/synchronous collaboration in live coding music: networked live coding at Daghstuhl Seminar (Swift, Gardner, and Sorensen 2014), *extramous*, language-neutral shared-buffer networked live coding system (Ogborn 2014d), and *Gibber*, a live coding environment on a web browser, which supports remote code edit and run (similar to Google Doc) (Roberts and Kuchera-Morin 2012). However, most previous live coding ensembles have focused on collaboration in synchronous fashion (the lower half of the chart in Figure 2).

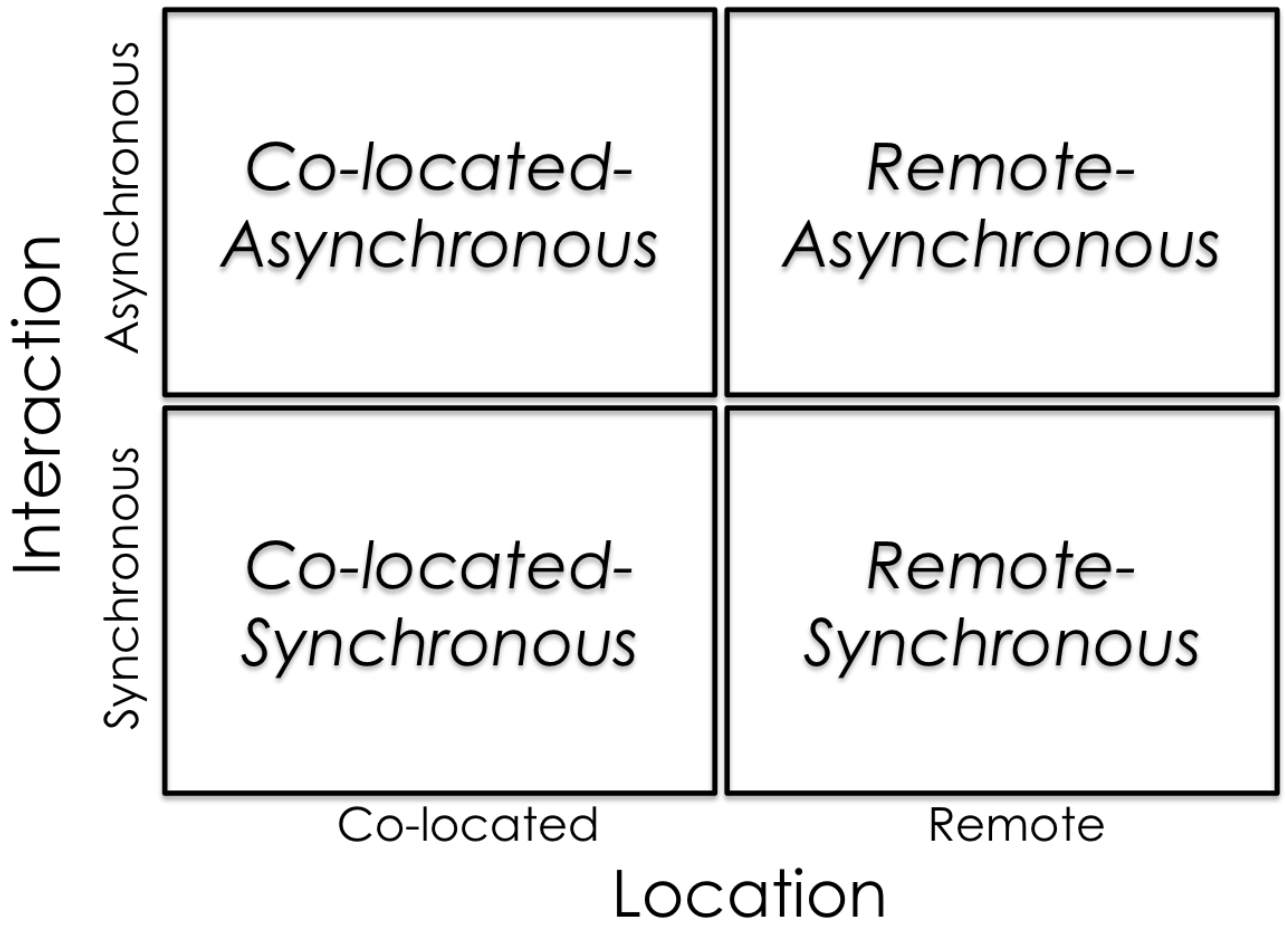


Figure 2: *Classification Space for Network Music. Adapted after (Barbosa 2003).*

2.2. Music Notation in Live Coding.

In the traditional sense of music notation, where the objective is for a composer to communicate with performers in the future, the music notation for live coding is not well-established for discussion. Traditional music notation is certainly not well fitted for notating live-coded music. This is because live coding is a highly improvisational and interactive form of art where composition occurs at the time of performance; in live coding there is no clear distinction between composer and performer. Rather, a live coder is "composer-performer."

Indeed, the discussion of notation in live coding has been focused on its real-time usage at the performance. In this context of real-time composition/performance, the code (either textual or graphical) serves as a music score and diverse scoring approaches as code representation have been attempted (Magnusson 2011). Graphical visualization of the code is effective for better audience communication (McLean et al. 2010). In addition, the real-time music notation can be a medium that connects live coding musicians and instrumental performers (Lee and Freeman 2013; Hall 2014).

2.3. Archiving a live coding performance.

It seems that, for the time being, the music notation for live coding music with the traditional goal of documenting a piece may not need to reach a single conclusion. Instead, live coders should be able to archive live coding performances for the following benefits: i) an archive of a live coding performance can be used to exchange ideas and data with collaborators, ii) the collection of live coding pieces can be fruitful, from a long-term perspective, for the live coding community as a way to transfer knowledge, providing inspiration and developing the field for posterity.

One immediate solution to archive a live coding piece is to save, as electronic files, the program code that was used in the performance. However, the code alone is not enough for a live coder to document a music piece in a way that someone in the future (including oneself) would be able to repeat the performance. Static code itself cannot reproduce a live coding piece because it does not contain the performance aspects of algorithms execution. Furthermore, a certain amount of virtuosity in live coding is involved with real-time manipulation of code, which is not well exposed in the final text of the code. Practitioners use static code in combination with other tools such as audio recording, videotaping, screen recording, or open form score (e.g., textual or graphical notes) as part of the documentation in rehearsal, composition, and preparation steps of a performance.

Archiving a live coder's performance will have more potential in the scenario of a live coding ensemble to support asynchronous communication among the members. At the moment, the most obvious collaboration strategy that all live coding ensembles take is a combination of rehearsals and individual practice. However, it will be practically challenging to find the time (and the place) that every member of the ensemble can participate. The problem will worsen in scale, such as in the case of a live coding orchestra (Ogborn 2014c). Alternatively, each musician can do at-home practice to create new ideas for the ensemble and the individual. However, communication is delayed until the next gathering. To that end, a live coding ensemble will benefit from developing a new format of archiving a music performance, which will eventually support non real-time collaboration among members.

Lastly, archiving a live coding performance will enhance the composition process for a solo live coder. A live coder can log each composition session to document the progress; the archive will help the composer capture new ideas, which appear of a sudden and go away during the exploration and the improvisation.

2.4. Live Writing: Documenting and Replaying Textual Dynamics of Live Coding

The live coding piece performed on Live Writing platform will be logged in the keystroke levels with timestamp information so that the generative music can be reproduced by the typing simulation. Given it is sequenced data rather than fixed media, this differs from audio/video recording, and it is different from the music notation, as it can be used to capture a particular performance. It provides symbolic information, which a musician can easily copy, duplicate, and make changes to. The log file generated from Live Writing platform is analogous to a MIDI sequence file, which describes musical dynamics as an intermediate often input-centric representation. Live Writing serves a similar function in terms of capturing input-centric dynamics of text entry. Figure 3 shows the analogy between traditional music and live coding in terms of ways to archive a music performance.

The highlight of Live Writing platform is that it provides temporal information of the live coding piece that static code cannot. Replaying the code of a piece can inform the order of code written, the timing of execution, and the virtuosic moves of a live coder, elements that would have been impossible to know in the final code. The playback feature could help transfer practical knowledge of carrying out the piece and help people practice along with collaborator's simulated performance. The idea of Live Writing draws upon existing systems that enable asynchronous displays of code. In

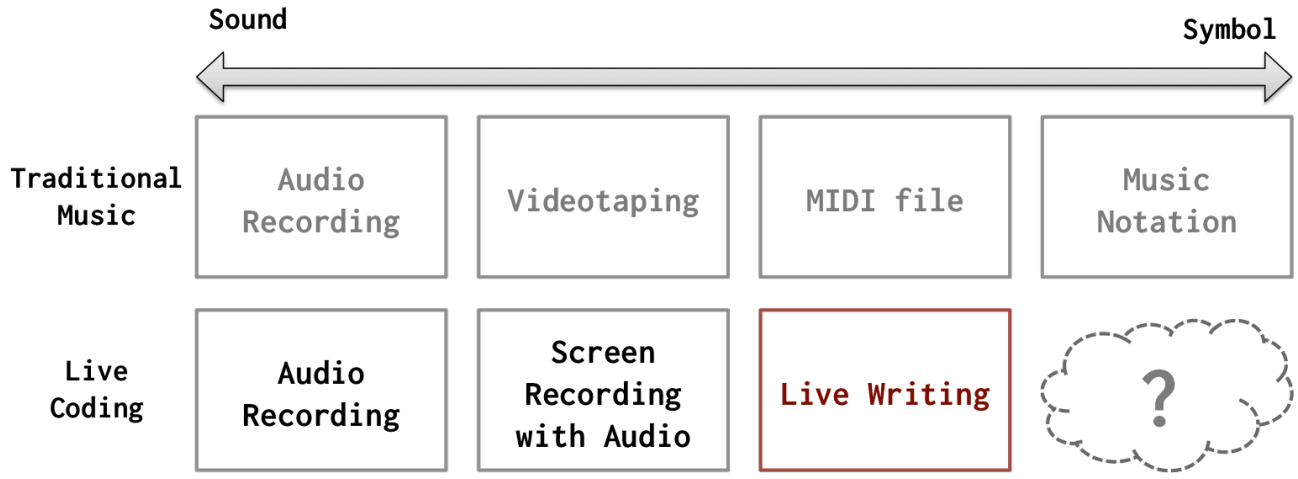


Figure 3: *The methods to archive music performances in traditional music and live coding.*

particular, Live Writing platform is directly inspired by Ogborn (2014e)'s performance using *Daem0n.sc* system, which types the text of a file automatically—in sync with the default metronome in Supercollider. *Gibber* provides a playground in which people can publish and browse *Gibber* code so that the hub supports asynchronous collaboration in a broad sense within the community (Roberts et al. 2014). In *Threnoscope*, Magnusson (2014) introduced code snippets in a piano-roll-like notation where it is used as a secondary composition/performance interface, potentially a great archive format to document a performance on the system. In non-artistic cases, Google Wave, which is no longer available, contained the playback feature at the email transaction level so that one can understand the temporal order of email messages especially for those who got included in the conversation later. Lastly, thecodeplayer.com implements keystroke level playback of writing code but it is not currently accessible to the public to create contents("Thecodeplayer.Com").

2.5. Beyond Archive: New Media for Collaborative Live Coding

The symbolic representation of the archived performance offers novel opportunities for asynchronous collaboration among musicians. A live coder can access the code generated on the fly easily while a screencast or the final static code cannot offer such data, i.e., copy the code in the middle of the playback simulation and paste it on one's editor to use existing code to develop ideas, and to create collaborative musical patterns. It can be used throughout the rehearsal process for a live coding ensemble to sketch ideas and exchange code over emails, and to rehearse along with the ensemble members asynchronously.

Keystroke log is a valuable asset for monitoring/analyzing a live coder's performance. It is not only great for self-reflection on one's rehearsal/performance but also a useful material to analyze a live coder's performance. The performance log can be used to analyze live coding musicians' play and learn complex behaviors that are not translated easily from audio, static code and the artist's statement, for example, the virtuosity of a performer in programming and musical style. Such research has been done to create a performer's fingerprint (Swift et al. 2014) and would have been expedited and enriched with the corpus of logged data accumulated in the system, otherwise, such data should be manually annotated from the screencast.

Lastly, supplementing visualization/navigation techniques to be implemented in the future (the navigation bar, keystroke density visualization, temporal typography) will offer an interactivity in the middle of playback of live coding. For example, a reader can pause, modify, fork or remix a performance and create a unique live coding collaboration that can occur over a long time. Forthcoming features of the Live Writing system will be discussed in detail at the end of this paper.

3. Written Communication into a Real Time Performance

In this section, we switch the context of discussion from live coding music to writing in general. First, imagine a performer going on stage and starting to write a poem (whether it is typed on a screen or written on a piece of paper, shared with the audience). The live coding community would easily call this a live writing performance and the performer would need nothing more than sheets of paper and a pen or a text editor on a laptop projected on a screen. Live Writing platform

allows users to dynamically present a writing; readers can read the article as if it is being typed right in front of them. It does not necessarily need to be a synchronous performance in public but anything written in private (for example, poem, essay, instant messages, email, or tweets) can be presented as if readers were watching the writer typing in real time. The same platform that was used to archive live coding performances is repurposed for this new expressive art.

3.1. Keystroke Logging and Playback of Writing

The realization of Live Writing platform is to utilize keystroke logging on a computer. For a long time now, the idea of logging user input from a keyboard has been a simple but powerful research method in many different contexts. *Keystroke Level Model* (KLM) was an important aspect of research into modeling human performance in many human-computer interaction tasks (Card, Moran, and Newell 1980). It was also used to analyze email organization strategy (Bälter 2000). Keystroke dynamic has also been used for verifying identities. Habitual temporal patterns of key entries provide a behavioral biometric useful for identifying individuals (Joyce and Gupta 1990; Monrose and Rubin 2000).

In helping analyze real-time writing behavior, keystroke logging has become a common approach in the field of writing research. It provides a non-intrusive and inexpensive technique to monitor user inputs. Writing researchers have developed a number of keystroke logging applications—*Inputlog*(Leijten and Van Waes 2006) and *ScriptLog*(Strömqvist et al. 2006) are two examples of such programs. Most keystroke logging applications include real-time playback recorded keystrokes and it is an effective approach to help subjects account for their writing in retrospect, which is less intrusive than having them think aloud while writing (Latif 2008). These applications are mainly for the research purpose of real-time writing analysis rather than for writing in general public.

3.2. Live Writing : Writing as Real-Time Performance

The direct inspiration of the notion of live writing is live coding. Writing is an expressive process guided and adapted by thoughts that evolve over time. By revealing the real-time writing process to readers, Live Writing lets a writer show the thought process and the intermediate stages of the writing as it is typed, including typos, corrections, and deletions. This principle is well captured in the following statement of the *TOPLAP* manifesto: "Obscurantism is dangerous. Show us your screens." We expand the same idea to writing in general. Showing the entire process of writing as it emerges sheds light on the trajectory of thoughts and provides more expressive power than when reading final text. It is analogous to revealing the thought process of a composer as a form of algorithm in live coding music.

The real-time rendering of writing in the same way that it was typed is certainly more expressive than static writing. There are various kinds of writer's emotional states (such as contemplation, hesitation, confidence, or agitation) that can emerge during the process of typing based on temporal patterns, for instance, pause, bursts, or corrective steps. The fact that every single entry is shown in the final outcome transforms writing in general into a creative experience in real-time, similar to musical improvisation. Indeed, such an improvisational nature is prominent on the World Wide Web, for example, vlogging, podcasting, live game streaming, and video tutorials with screen recording and voice over.

It may be argued that calling this "Live" Writing is somewhat misleading as the Live Writing platform enables asynchronous (that is apparently not "live") uses. However, we hold that the very process of writing is the live act in this case, and this liveness of the writing process is captured, hence justifying the use of "live" in this context. Moreover, by explicitly capturing the the temporal dynamic of the writing process it suggests a changed mindset in the writer with respect to the meaning and performative function of the writing process itself. A writer can now ponder the arrangement of text over time and leverage the articulation of the temporal dynamics (like a composer organizing sound to make music). This can be premeditated as in compositions, or improvised. Further it suggests a deliberation of the delivery process even if the material is fixed, leading to an expressive layer of interpretation that transcends yet potentially interrelates with the text. One can see this analogous to the notion of a "live" recording. The production process is live, yet the reproduction method is not. This does by no means preclude the consideration of live writing in a fully real-time synchronous performance setting. In fact we are currently exploring synchronous live writing piece as a form of audiovisual performing arts, of which a visualization technique on typography is discussed elsewhere (Lee and Essl 2015).

4. Design and Implementation

Live Writing is implemented as a web application in order to be easily accessible from a variety of platforms and devices. The web browser is chosen to make the platform language neutral (as long as the live coding language is textual). In addition, the web browser is one of the most popular writing platforms today. Many live coding language environments

allows the editing part to be separated from the local machine and to reside on the web browser or web-based editors (such as Atom, Sublime Text, or Brackets I/O). This would work, of course, for live coding languages built into the web browser. We realize this system cannot support many live coders who use other popular editors (e.g. emacs). However, we chose the web browser given it is a good platform-independent application and the use of web browser is on the increase in live coding. Eventually, modern web-based editing APIs such as CodeMirror and ACE have been and will be evolving to support many functions like shortcuts and macro to replace such editors or the Live Writing feature can be easily implemented in across multiple editing environments if needed. It should be addressed that the system will be limited to textual live coding languages but the idea can be extended to graphical languages.

The API is designed to extend existing an html object <textarea> or code editing APIs (e.g. codemirror, ace) so that it can easily extend existing systems on the web browsers. In the following section, the implementation of the application for the writing purpose is described first. Demonstrated later in this section is an archiving and playback example built on top of *Gibber*.

4.1. Live Writing App for Written Communication

For the writing purpose of Live Writing, the application is publically available for anyone to use http://www.echobin.com/. The current implementation is crafted with minimal design; it has only one clean slate, an initial dialog with brief description, screen-sized <textarea> objects, and a few buttons hidden on the side (see Figure 1). Once a user presses the start button, all keystrokes and mouse cursor clicks made inside the <textarea> will be recorded on the local machine. And then, by pressing the post button on the right side, the user can post the piece of writing. Once the data is posted, the user is given the link that accesses the real-time playback of the writing. The link contains the article ID and anyone with the link can view the playback of the writing. Therefore, access control of the writing is up to the writer and whom he or she chooses to share the link with, an action that can be easily done via email, a facebook post, or instant messages.

The web application is realized in javascript/jQuery/AJAX and node.js. We released the code as an open-source API so that any html <textarea> can be extended to feature keystroke logging and playback by including one javascript file. The server runs node.js script which handles the static files and store/retrieve recorded keystrokes log in json format. All the other functions of keystroke logging and playback is implemented and run in the client machine which the writer uses. The logged data is not stored in the server until the user chooses to post the writing. Alternatively, a user can choose to download the log file instead of posting to the server, if the user wants to keep the logged data locally. Anyone who has the log file will be able, by uploading the file, to replay the writing. Providing raw data in the file will be useful for extracting high-level information such as keystroke dynamics or behavioral pattern when processed. We hope that this supports researchers wanting to use this as a keystroke-logging application.

To implement playback, the keystrokes logged are simulated by specifically reproducing what the keystroke would change in the content of <textarea>. In other words, a web browser does not allow, for security reasons, javascript to generate the exact same keystroke event. Instead, for example, to reproduce a user's pressing of the "s" key, it has to append the "s" character to where the cursor is in the <textarea> at the moment. Note that the <textarea> used in keystroke logging is again used for playback. Improvements to the website are in progress so that a user can customize the writing not only visually (font type, size etc.) but also through its playback setting (e.g.,, playback speed, navigation across time, etc.).

4.2. Playback of live coding : *Gibber* code on *codemirror*

To demonstrate the archiving and replaying features of Live Writing, this study has chosen *Gibber* (Roberts and Kuchera-Morin 2012), a live coding environment on a web browser. It could have been possible to create the same features with other live coding languages (e.g., Supercollider) that can be edited on a web browser and communicated with the live coding engine by sending textual data to the localhost. Because *Gibber* is built into the web browser, it makes the demo accessible to public without any configuration and installation. Lastly, for the code-editing environment, the study uses *Codemirror* (Haverbeke 2011). Codemirror comes with a set of features readily available for programming (e.g., syntax highlighting, keymap binding, autocomplete).

The user experience in the *Gibber* demo is similar to that of the writing web application introduced above. In terms of implementation, however, a few differences from the writing scenario should be noted. This uses high-level events (onchange, cursor activity) supported from codemirror instead of low-level events (e.g., keyUp, keyDown, keyPress) in <textarea>. In addition, the pressing of shortcut keys related to code execution (Ctrl-Enter, Ctrl-.) is separately stored as a message so that simulated typing will trigger a code run to get audiovisual outcome while playback. The working demo is available at http://www.echobin.com/gibber.html. For a simple playback demo without entering Gibber code, visit http://www.echobin.com/gibber.html?aid=OKDMWHgkDCdAmA which shows the demo captured in Figure 4.

Figure 4: *Screen captures of Gibber Demo Audiovisual outcome generated automatically by playback of Gibber code over time (from left to right). This demo is available at http://www.echobin.com/gibber.html?aid=OKDMWHgkDCdAmA*

5. Conclusion

In this paper, we have introduced Live Writing, a web-based application which enables keystroke logging for real-time playback of the writing. The motivation for this idea is to facilitate asynchronous collaboration among live coders and to archive a live coding performance in an informative way. Additionally, the Live Writing platform is repurposed for general written communication and has potentially turned a writing activity into a live coding like performance.

There are a number of directions that we can take for future research. We plan to integrate Live Writing editor with existing live coding environments and evaluate the usage of features in practice. It would be desirable to investigate the system usage in a collaboration scenario like live coding ensemble. On the other hand, with the features of playback, we would be able to build a live coding gallery in which users publish their piece and people can enjoy the live coding piece remotely and asynchronously. In addition, we want to make the playback feature not interfere with key entry so that a spectator can write text while keystrokes are replayed. This will create unique opportunities in a crowd-sourced music piece that grows over time by people's participation in a system that is similar to github.

Another ongoing effort is to explore the idea of live writing and to deliver a live-writing performance and a participatory audiovisual art on a web browser. Currently, temporal typography based on a web browser is in development to augment the text editor with animated fonts (Lee and Essl 2015). It will afford novel ways of visualizing text in combination with music and algorithms, associated with the content of writing. To explore the idea of live-writing poetry, an audiovisual piece is in preparation for a public concert. In addition, visualization of the text will be used to visualize the program state as well as music in the context of live coding.

Lastly, the Live Writing application will be further developed for general programming and evaluated in terms of human-computer interaction. We believe that the playback feature of the editor will attract programmers to use the editor for numerous reasons (self-evaluation, fine-grained version control, online tutorial, collaboration, programming challenge/interviews). Improvement of the editor in progress includes the navigation bar, typing density visualization, and automated summary. The collection of source code published to the server can be used to mine valuable information (i.e., temporal progress of good/bad coding style) in the context of software engineering and pedagogy in computer science

References

Bälter, Olle. 2000. "Keystroke Level Analysis of Email Message Organization." In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 105–112. ACM.

Barbosa, Álvaro. 2003. "Displaced Soundscapes: a Survey of Network Systems for Music and Sonic Art Creation." *Leonardo Music Journal* 13: 53–59.

Brown, Andrew R, and Andrew C Sorensen. 2007. "Aa-Cell in Practice: an Approach to Musical Live Coding." In *Proceedings of the International Computer Music Conference*, 292–299. International Computer Music Association.

Card, Stuart K, Thomas P Moran, and Allen Newell. 1980. "The Keystroke-Level Model for User Performance Time with Interactive Systems." *Communications of the ACM* 23 (7): 396–410.

Collins, Nick, Alex. McLean, Julian. Rohrhuber, and Adrian. Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (03): 321–330.

Essl, G. 2010. "UrMus – An Environment for Mobile Instrument Design and Performance." In *Proceedings of the International Computer Music Conference (ICMC)*. Stony Brooks/New York.

Hall, Tom. 2014. “Live Digital Notations for Collaborative Music Performance.” In *Proceedings of the Live Coding and Collaboration Symposium 2014*. Birmingham, United Kingdom.

Haverbeke, M. 2011. “Codemirror.” http://codemirror.net/.

Joyce, Rick, and Gopal Gupta. 1990. “Identity Authentication Based on Keystroke Latencies.” *Communications of the ACM* 33 (2): 168–176.

Latif, Muhammad M Abdel. 2008. “A State-of-the-Art Review of the Real-Time Computer-Aided Study of the Writing Process.” *IJES, International Journal of English Studies* 8 (1): 29–50.

Lee, Sang Won, and Georg Essl. 2014a. “Communication, Control, and State Sharing in Collaborative Live Coding.” In *Proceedings of New Interfaces for Musical Expression (NIME)*. London, United Kingdom.

———. 2014b. “Models and Opportunities for Networked Live Coding.” In *Proceedings of the Live Coding and Collaboration Symposium 2014*. Birmingham, United Kingdom.

———. 2015. “Web-Based Temporal Typography for Musical Expression and Performance.” In *Proceedings of New Interfaces for Musical Expression (NIME)*. Baton Rouge, United States.

Lee, Sang Won, and Jason Freeman. 2013. “Real-Time Music Notation in Mixed Laptop–Acoustic Ensembles.” *Computer Music Journal* 37 (4): 24–36.

Leijten, Mariëlle, and Luuk Van Waes. 2006. “Inputlog: New Perspectives on the Logging of on-Line Writing Processes in a Windows Environment.” *Studies in Writing* 18: 73.

Magnusson, Thor. 2011. “Algorithms as Scores: Coding Live Music.” *Leonardo Music Journal* 21: 19–23.

———. 2014. “Improvising with the Threnoscope: Integrating Code, Hardware, GUI, Network, and Graphic Scores.” In NIME.

McLean, A., D. Griffiths, N. Collins, and G. Wiggins. 2010. “Visualisation of Live Code.” *Proceedings of Electronic Visualisation and the Arts 2010*.

Monrose, Fabian, and Aviel D Rubin. 2000. “Keystroke Dynamics as a Biometric for Authentication.” *Future Generation Computer Systems* 16 (4): 351–359.

Ogborn, David. 2014c. “Live Coding in a Scalable, Participatory Laptop Orchestra.” *Computer Music Journal* 38 (1): 17–30.

———. 2014d. “Extramuros.” https://github.com/d0kt0r0/extramuros.

———. 2014e. “Daem0n.” https://github.com/d0kt0r0/Daem0n.sc.

Roberts, C., and J.A. Kuchera-Morin. 2012. “Gibber: Live Coding Audio in the Browser.” In *Proceedings of the International Computer Music Conference (ICMC)*. Ljubljana, Slovenia.

Roberts, Charles, Matthew Wright, J Kuchera-Morin, and Tobias Höllerer. 2014. “Rapid Creation and Publication of Digital Musical Instruments.” In *Proceedings of New Interfaces for Musical Expression*.

Rohrhuber, Julian, Alberto de Campo, Renate Wieser, Jan-Kees van Kampen, Echo Ho, and Hannes Hölzl. 2007. “Purloined Letters and Distributed Persons.” In *Music in the Global Village Conference (Budapest)*.

Strömqvist, Sven, Kenneth Holmqvist, Victoria Johansson, Henrik Karlsson, and Åsa Wengelin. 2006. “What Keystroke-Logging Can Reveal About Writing.” *Computer Key-Stroke Logging and Writing: Methods and Applications (Studies in Writing)* 18: 45–72.

Swift, Ben, Henry Gardner, and Andrew Sorensen. 2014. “Networked Livecoding at VL/HCC 2013.” In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 221–222. IEEE.

Swift, Ben, Andrew Sorensen, Michael Martin, and Henry Gardner. 2014. “Coding Livecoding.” In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, 1021–1024. ACM.

“Thecodeplayer.Com.” http://thecodeplayer.com.

Wilson, Scott, Norah Lorway, Rosalyn Coull, Konstantinos Vasilakos, and Tim Moyers. 2014. “Free as in BEER: Some Explorations into Structured Improvisation Using Networked Live-Coding Systems.” *Computer Music Journal* 38 (1): 54–64.

Live Coding Through Rule-Based Modelling of High-Level Structures: exploring output spaces of algorithmic composition systems

Ivan Paz
Computer Sciences Department, Barcelona Tech
ivanpaz@cs.upc.edu; ivnpaz@gmail.com

ABSTRACT

Live coding commonly takes pieces of code from algorithmic composition systems. However, sometimes algorithmic generators either do not consistently show high-level properties, like dramatic transition among parts, or simply, our aesthetic criterion prefers some particular cases among all the possible. In such cases it is useful to have tools for exploring the output space of generative systems, in order to identify and categorize outputs with specific properties. This paper presents an approach to creating linguistic rules out of human-evaluated patterns and their potential uses in live coding to create high-level structures. The methodology starts with a set of sampled examples from an algorithmic system that are evaluated by the user through qualitative linguistic variables. Then, the examples along with the user's evaluation are analysed through an inductive and rule extraction methodology. For a particular example case, these rules are extracted and evaluated. Its application then as information used for writing code on the fly, as well as its implementation in the form of filters or generators is presented.

1. INTRODUCTION

Live coding is closely related with algorithmic composition (Collins 2003a). Many times structures are taken from one domain to the other to supply new material. However, it is common that the outputs of the algorithmic systems either do not consistently show high-level properties, like real dramatic transition among parts (Collins 2009), or simply, our aesthetic criterion prefers some particular cases, among all the possible, for specific parts. In addition, some systems have a large amount of input parameters (see for example Povel, 2010) that make them difficult to set or control, specially in the context of live performance. Also, the output space of the systems is normally huge and difficult to explore by hand. Furthermore, in manual explorations, it is usually hard to establish a relationship between the variables or parameters of the system and the desired outputs. To overcome this problem within a live coding context, instead of starting from a blind exploration, it is helpful to have tools to explore the sample space of the algorithmic systems, in order to identify and categorize the outputs that we want to use. An interesting example, is presented in Dahlsted (2001), who proposed a system, based in interactive evolution, that finds preferred spots in the output space of sound objects created by synthesis and pattern generation algorithms. This work undertakes the exploration of the algorithmic system's output spaces by looking for subspaces with high-level structures. By high-level structures we mean musical objects that are derived from the combination of lower level musical objects, in this case from the combination of the input variables. This work presents an approach to creating linguistic rules out of human-evaluated patterns, characterizing those subspaces, and discusses their potential uses in live coding to create high-level structures. The rest of the paper is structured as follows: Section 2 describes the rule extraction methodology through a specific example, and section 3 discusses the implementation of the rules in the live coding context and outline some ideas for possible further work.

2. AN INDUCTIVE APPROACH FOR HIGH-LEVEL FEATURES RULE GENERATION

This section describes the rule extraction process through a specific example. For further details about the rule extraction methodology, the reader is referred to Mugica et al, (2015), Nebot and Mugica (2012), and Castro et al, (2011). The system's design is shown in Figure 1.

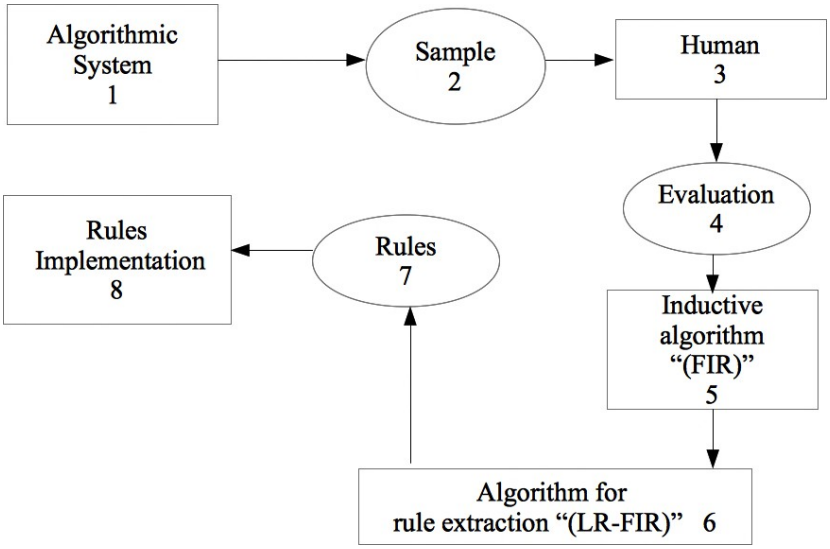


Figure 1. General chart flow of the rule extraction process.

The first module (1) is the algorithmic system. For this example, a simplified algorithm (shown in Table 1) based on (Collins, 2003b) was used. It consists of a set of probability templates for 1 bar, divided in semiquavers for kick, snare and hihat, that produces beat patterns in the style of UK garage. Each number is the probability (normalized up to 1) for an instrument to play at that point in time.

kick	[0.7, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3]
snare	[0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.5, 0.0, 0.2, 0.0, 0.0, 1.0, 0.0, 0.0, 0.3]
hihat	[0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.7, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.7]

Table 1. Templates for UK garage based on Collins, (2003b).

From these templates a training set of 100 patterns, structured in A + B form where A and B are instances of the algorithm and A ≠ B, was created. The training set was evaluated by the user by means of linguistic variables. In this example, for each pattern in the training set, the aesthetic quality of the structure was labeled as low, medium or high. The data was represented in arrays of length 33. The first 32 values are the semiquavers of the pattern A + B, with a 4 denoting silence, 1 kick, 2 snare, and 3 hihat. The final value at index 33 was the user's evaluation: 1 = low 2 = medium and 3 = high. The examples were evaluated and the data was used as input for the Fuzzy Inductive Reasoning methodology (FIR, module 5). FIR is a qualitative non-parametric, shallow model based on fuzzy logic (Nebot and Mugica 2012). It obtains a set of pattern rules, that describe the qualitative behaviour of the system, by means of searching algorithms, like trees and genetic algorithms. This rule base can be very large and difficult to understand. The Linguistic Rules in FIR (LR-FIR, module 6) contain the algorithm for linguistic rule extraction starting from the pattern rules set generated by FIR, and deriving a reduced and manageable set of linguistic rules. The algorithm performs an iterative process that compacts (or compresses) the pattern relationships obtained by FIR in order to create interpretable rules. This is done by following several steps that include algorithms for rule compaction and optimization processes (Castro et al 2011). An example of the resulting rules is shown in Table 2. Rules are expressed in terms of the values in its variables (V) representing the 33 indexes of the representation discussed above. For example RULE 1 should be read as: If location 16 has a snare or a hihat and location 24 has a kick and location 32 has a kick, then the pattern will be evaluated as high. For each rule "Classified pattern" denotes the amount of patterns (taken in A + B form) that satisfy that rule among all others in the sample space of the algorithm.

RULE	EVALUATION
RULE 1: V16 in 2-3 AND V24 in 1 AND V32 in 1 THEN V33 in 3 Classified patterns 512 **out of** 16384	TP = 16, FP = 4, PRECISION = 0.8
RULE 2: V24 in 1 AND V26 in 2 AND V32 in 1 THEN V33 in 3 Classified patterns 512 **out of** 16384	TP = 16, FP = 4, PRECISION = 0.8
RULE 3: V10 in 4 AND V24 in1 AND V32 in 1 THEN V33 in 3 Classified patterns 512 **out of** 16384	TP = 15, FP = 5, PRECISION = 0.75

Table 2. If-then rules obtained by the system and its evaluation in terms of precision. The three rules describe aesthetic structures user-evaluated as "high" (V33 in 3).

The resulting rules were evaluated using the standard confusion matrix. The Precision (True positive (TP) / True positive + False positive (FP)) was used for the evaluation of each rule. In this case, for each rule a sample of 20 pattern satisfying the rule was created and evaluated. TP patterns were patterns perceived as belonging to the class established by the rule. FP patterns were patterns that satisfy the rule but that are not perceived as belonging to the class of the rule.

3. RULE IMPLEMENTATION FOR LIVE CODING

In this section two approaches for the implementation of these rules in live coding are discussed. Both have been used during live performance with each having their own pros and cons. The standard way of implementing the rules, is to analyse them and use the resulting information as a guide to write our code on the fly. However, as the number of variables in the rules increases, this strategy becomes impractical. In addition, depending on the venue, sometimes we cannot take the risk of have awkward transition moments during the performance. In that cases, the safety of having a pre-programmed structure is preferred. Next, these two strategies are discussed.

As examples of information that can directly be inferred from the rules, let's consider two cases. First, in Table 2 it can be seen that kicks are always found in variables 24 and 32. Then, a safe bet would be to start by setting kicks in those variables. Another insight that can be directly inferred from inspection of the rules is: If we compare the amount of silences (variables in 4) contained in rules describing the high class against silences in rules describing the low and medium classes shown in Mugica et al. (2015), we find only 1 in the high class, while we find 15 and 9 in the medium class and low class rules respectively. From this, we can infer that to produce the desired structure as obtained from the listener's evaluation, it is better to avoid silences, i.e. patterns with higher content of sound information were preferred.

When instead of starting from the white screen we choose to write a patch, or prepare some code from the rules, this can be done in two main forms (without excluding possible combinations): 1. Implement the rules as a filter, to filter out patterns that satisfy rules that do not belong to the chosen class. 2. Implement the rules as a module or class for pattern generation.

3.1 Filter of rules v.s. generative implementation

Implementing the rules as a filter can be thought of as follows. Create a pattern, and if the created pattern does not satisfy any of the rules in the filter, then play the pattern. This implementation produces patterns that show more variety compared with the generative implementation (discussed next). However, we have to be careful, given that the rules do not cover the whole output space. The current methodology produces rules covering around 60% of the space and the precision is approximately 72% (Mugica et al, 2015). Thus, we may be exposed to surprises that have to be managed as we play.

The generative implementation considers the selected rules, and modifies the original algorithm to produce only patterns satisfying one of them. This implementation is perceived as more stable, but at the same time, it produces less variability because of the number of fixed variables. To achieve more variability, implementations can vary ranging from the random selection of the rules to be satisfied, to the use of pondered schemes or probability distributions, assigned according to a

ranking of the preferred rules. We can also listen to a sample of the patterns created by each rule, and select the rules for its application in different sections of the performance by aesthetic criteria.

Finally, a more conservative approach could be to pre-select some structures and launch them in turn. This has also been tried by using the rules of the prior example. Patterns in A + B form were selected and then structured in the form n times (A+B) + n times (A' + B ') + etc., to handguide the development of the piece. In this context the cells found could be used as a raw material to fill sections, guiding the evolution of the piece by hand.

Acknowledgments

My sincere thanks to Diego Garcia-Olano for the careful revision of this text, and to the anonymous reviewers that made suggestions to improve the trend of thought of the document.

REFERENCES

Collins, Nick. 2003a "Generative Music and Laptop Performance" Contemporary Music Review. Vol 22, No. 4, 67-79.

Collins, N. 2003b "Algorithmic Composition Methods for Breakbeat Science" ARiADA No.3 May.

Collins, Nick. 2009 "Musical Form and Algorithmic Composition" Contemporary Music Review Vol. 28, No. 1, February 2009, pp. 103–114.

Dahlstedt, P. 2001 "Creating and exploring huge parameter spaces: interactive evolution as a tool for sound generation". *Proceedings of the International Computer Music Conference,* Habana, Cuba.

Nebot, À. Mugica, F. 2012. "Fuzzy Inductive Reasoning: a consolidated approach to data-driven construction of complex dynamical systems." International Journal of General Systems. Volume: 41(7) Pages. 645-665.

Mugica, M. Paz, I. Nebot, À. Romero, E. 2015. "A Fuzzy Inductive Approach for Rule-Based Modelling of High Level Structures in Algorithmic Composition Systems". In proceedings of IEEE international conference on fuzzy systems.

Povel, D. 2010 "Melody Generator: A Device for Algorithmic Music Construction" J. Software Engineering & Applications, 2010, 3, 683-695.

Cognition and Improvisation: Some Implications for Live Coding

Dr Tim Sayer
University of St Mark & St John Plymouth
tsayer@marjon.ac.uk

ABSTRACT

This paper explores the perception that live coding is a "real-time" improvisatory activity. It posits the notion that because live coding requires less complex motor skills than instrumental improvisation it may be less susceptible to the influence of mechanical modes of musical expression. This hypothesis will explore the concept of goal states, models of memory and the function of reflexes and reactions as a means of mapping this territory and will provide a framework to understand the various perceptual domains with which a coder engages during a live extemporised performance. This exploration will engage in a comparative discourse relating live coding to instrumental improvisation, as a point of departure for the understanding of cognitive functioning in this rapidly developing performance paradigm.

1. GOAL STATES

In a healthy human being, perception works in such a way as to present a cohesive and flowing conscious experience. The resources associated with cognitive processing are managed by a plethora of neural processes in a manner that is optimised, for the most part, to keep us safe and healthy. This allows us to ebb and flow between mental states which might be processing sensory data in radically different ways but which are presented to us as a single, fluid experience. One of the key mechanisms in managing the various activities we synchronously engage in is the monitoring and maintaining of various goal states being held at any one time. Although many contemporary theories suggest goal pursuit is the result of conscious deliberation and desire (Gollwitzer & Moskowitz, 1996 ,Baumeister & Vohs, 2004), Ruud Custers posits a convincing argument that goal pursuit can be enacted outside of conscious awareness (Custers, 2009). This theory has been successfully tested using priming techniques and has led to the development of such concepts as nonconscious will (Bargh et al., 2001), implicit intention (Wood, Quinn, & Kashy, 2002), or implicit volition (Moskowitz, Li, and Kirk 2004).

Put simply, a goal state that is not afforded the spotlight of conscious awareness but which begins to rise in significance, will cause lower priority undertakings to have their monitoring diminished or maybe even terminated. Consider the metaphor of a driver listening to a play on the radio during a long motorway journey. For long periods of the journey the cognitive processes involved in the act of driving the car can be successfully filtered in order to pull focus on the narrative as it unfolds on the radio. The cognitive goal at this stage is to follow the narrative of the play and so conscious awareness is given over to this undertaking. As the driver passes a sign post, the 'radio play - perception' process is interrupted by the need to devote attention to the road sign as the goal of the 'journey process' has priority over the goal of the 'radio play process'. If another car suddenly brakes then the goal of the 'survival process' will be triggered by the brake lights of the preceding car and take precedence over the 'journey process' causing an action compliant with that goal state.

Many studies have concluded that affective priming can have a huge influence on goal motivation (Winkielman, Berridge, and Willbarger, 2005), (Aarts, Custers & Holland, 2007), It is quite possible that many of the goal states that comprise an improviser's practice are in fact instigated by non-musical and subliminal stimuli. After all, improvising performers are not shielded from their environment or their history for the duration of their performance and the conceptual and perceptual processes that drive their musical behaviour are fueled as much by implicitly held concepts as real-time sensory data. The conceptual and perceptual processes that determine the specifically musical goal states in improvised instrumental music and live coding are therefore likely to be influenced by factors outside of their awareness, not least of

which is the short-latency dopamine signal, believed by some to play a role in the discovery of novel actions, an area of enquiry which could be very illuminated in this context but which is outside the remit of this paper (Redgrave and Gurney, 2006).

Sperber states that perceptual processes have, as input, information provided by sensory receptors and, as output, a conceptual representation categorising the object perceived (Sperber 1996). The notion that perceptual processes have as their input the continuous data stream of sensory information and that the outputs from these processes become an input to conceptual processes, can be used to formulate a model of improvised behaviour, where the parameter (sensorial) space in which the behaviour exists is split between the internal and the external. The function of memory acquisition and retrieval therefore has an important role in pursuing goal states (Avery and Smillie, 2013).

2. MEMORY

For improvised musical behaviour of the type exhibited by live coders, the notion of perceptual and conceptual processes governing their behaviour should be put in the context of memory, particularly as there are significant differences between the role of memory in live coding and improvisation using traditional instruments. Motor skill execution is of particular importance to instrumental performers but there is a consistency across all physical motor skills in the manner of acquisition (Newell, 1991). The process begins with the repetition of small units of activity, which may be undertaken and evaluated using multiple senses. In terms of a musical instrumentalist this obviously utilises audible feedback but also uses a visual and tactile mode of feedback too, such as observing and feeling finger movement on a fret board. The process continues by constructing larger units from the primitive units of activity and so placing them in a range of contexts. At this stage the parameters and variants in the activities and their triggers can be developed. The repetitive act is in effect embedding a neural program into semantic memory that can then be triggered and passed a range of parameters. The effect of this is to significantly lighten the cognitive load, as once the activity has been triggered the detail in the activity can be undertaken with minimal conscious control, responding only to changes in its parameter space. A pianist could practice a scale in this way and when the motor units had been fully committed to memory the activity could be executed without conscious control over each of the component parts. In fact conscious attention is regarded by some to be detrimental to the accurate execution of highly complex motor skills (Wulf, 2007). The motor program may have a speed parameter that the pianist can control but is perhaps unlikely to have a parameter to change the intervals between the notes. In this way a whole catalogue of functional units can be triggered and left to run, leaving enough cognitive capacity to be able to undertake other tasks.

There is a distinction that needs to be made between the acquisition and retrieval of the activity units and the mode and manner in which they are assembled into larger conceptual units. Pressing makes this distinction by describing ‘object memory’ and ‘process memory’ as the place where the two exist. Crudely speaking ‘object memory’ for an instrumental musician would hold the activity units associated with playing the instrument, often identified by the blanket term ‘technique’. The information required to make use of the instrumental technique resides in ‘process memory’. This is where the methods for what Pressing describes as ‘compositional problem-solving’ are held. Methods of sequencing and systems for manipulating the parameter space of an activity unit are stored in ‘process memory’ and retrieved in accordance with the particular mode of performance. It is perhaps worth returning to the notion of cognitive capacity in the light of the processing that ‘process memory’ feeds. Motor activities such as tying up a shoelace or having a shave are reasonably cheap on effort and attentiveness, because the pathway through the activity units is comparatively static and can be easily predicted. However, the computational overhead will increase with human activities that require a higher level of monitoring and real-time manipulation. In the former case it’s possible for a person to experience a sensation of ‘switching off’ or diverting attention elsewhere. However, in the case of creative activities, routines executed from ‘process memory’ are likely to be more dynamic, requiring monitoring and parameter adjustment thus making greater cognitive demands. In this case the brain has a complex resource management problem to address in order for the activity to be undertaken effectively. Activities such as game playing and musical improvisation fall into this category but, as with all other activities they exist on a continuum somewhere between the extremes of static and dynamic. Their position on the continuum moment-by-moment, is determined by the resource management of the brain but over a period of time can be influenced by the level of pre-programming, or practice that is undertaken. The instinct to efficiently resource cognitive activity can be a double-edged sword when engaging in a creative

process such as improvisation. After years of encoding object and process memory it becomes possible for the whole 'creative' process to be undertaken by traversing chains of previously stored units of activity. Pressing summarises opinion on this matter when he stated that Fitts (1965) labelled this the 'autonomic phase', while Schmidt (1968) referred to it as 'automatization'. The result for the performer is a sensation of automaticity, an uncanny feeling of being a spectator to one's own actions, since increasing amounts of cognitive processing and muscular subroutines are no longer consciously monitored (Welford, 1976) (Pressing 1984). The significance of object memory for a live coder is therefore much less than for a traditional instrumentalist, although the sensation of being an observer is perhaps more vividly conscious, rather than reflexive automaticity.

The role of feedback in the perceptual loop, which gives rise to this type of experience, is important to consider. In this context, a close loop system describes the self-referential nature of activity units drawn from 'object memory' and subsequent evaluation for the purpose of error correction. Pressing describes one such feedback model, known as 'closed-loop negative feedback' (CLNF) (Bernstien 1967), in which an evaluation of intended and actual output is fed back to an earlier stage in the control chain. It is perhaps this process of evaluation that problematises the existence of closed and open loop feedback systems in improvisation. The submission by Pressing (Pressing 1988) that a consensus exists which suggests that both systems coexist seems reasonable, if one acknowledges that material drawn from process memory is also subjected to an evaluatory process, the result of which bears influence on the subsequent use of activity units. In this scenario, evaluation is undertaken at two levels simultaneously, but the nature of the evaluation is necessarily very different. It seems appropriate to apply the principles of closed loop theory to the evaluation of the primitive motor units, which suggest that a 'perceptual trace' of an activity unit is built up from its previous executions to form a template against which to gauge the success of its current execution. This theory (Adams 1976), is applicable to motor actions undertaken by improvising instrumentalists rather than live coders, because of the autonomous manner of their execution but is not sufficiently flexible to cope with this situation at the 'process memory' level. Pressing recognises why this should be the case when he asserts that in an improvising instrumentalist "the ability to construct new, meaningful pathways in an abstract cognitive space must be cultivated. Each such improvised pathway between action units will sometimes follow existing hierarchical connections, and at other times break off in search of new connections" (Pressing 1984).

The function of object memory and its relationship to motor skills in instrumental improvisation and live coding differs significantly due to the interfaces they employ to deliver the results of their creative practice using process memory. The overhead of encoding motor skills for live coding is less burdensome and consequently the cognitive resource gains of 'automaticity' are reduced compared with instrumentalists. It is true that the act of typing is a motor skill but there is less opportunity to build complex amalgamations of primitive units that facilitate direct control over the sound elements in a rich parameter space. That's not to say that fine control over sonic material is not available to live coders but the means of accessing it is more heavily reliant on process memory than pre-encoded units of behaviour from object memory. Although William James said in 1890, 'habit diminishes the conscious attention with which our acts are performed' (Pressing 1984) I am suggesting that live coders may be less susceptible to the influence of habit and mechanical modes of musical expression because the motor skills required are less complex than instrumentalists and consequently less of their behaviour is activated outside of their perceived conscious control or awareness. There is the possibility that habitual behaviour can develop via conceptual processes fed by process memory but in this instance there is a much greater opportunity for conscious intervention. Berkowitz surveys the various metaphors used by instrumentalists across a range of cultures to describe the ebb and flow between consciously controlled and automatic musical behavior during improvisation (Berkowitz, 2010). These are often characterised by a sensation of 'letting go' of conscious control and yielding to subconscious processes, described by Stephen Slawek as "an ecstatic state of effortless" (Slawek, 1998). The instrumentalist has at their disposal a vast resource of pre-encoded units of behavior which will sustain their music making once they have relinquished conscious control, which I would suggest is not available to the live coder. There is no doubt that live coders develop a repertoire of syntax and algorithmic schema they can call upon at speed and that the act of typing may become very fluid and intuitive but this is someway from Slawek's ecstatic effortlessness. Perhaps the automatic writing of the surrealists is the closest textual expression has come to engendering this state but the rigors of coding syntax would not permit such an endeavor in this context.

3. REFLEXES AND REACTIONS

I am suggesting that the live coding paradigm differs significantly from instrumental improvisation in its relationship to cognitive load. In instrumental improvisation the need to reduce the cognitive burden of conscious monitoring at high speeds is clear, but philosophically this can be a problematic notion for many musicians. Pressing identifies that speeds of approximately 10 actions per second and higher involve virtually exclusively pre-programmed actions (Pressing 1984). An informal analysis of jazz solos over a variety of tempos supports this ballpark estimate of the time limits for improvisational novelty. (Pressing 1988). It is probably fair to say that for some musicians there is a principle at stake in calling a musical activity improvisation that overtly draws on 'learnt' material. The saxophonist Steve Lacy, in conversation with Derek Bailey expresses his frustration accordingly when he says "why should I want to learn all those trite patterns? You know, when Bud Powell made them, fifteen years earlier, they weren't patterns. But when somebody analysed them and put them into a system it became a school and many players joined it. But by the time I came to it, I saw through it – the thrill was gone. Jazz got so that it wasn't improvised any more". (Bailey, 1992)

Although the causal relationship between the expression of ideas in code and the execution of those ideas by the computer is more asynchronous and less rigidly fixed to a timeframe than instrumental improvisation, live coding, like all other human activities, still has to exist within the confines of an available cognitive resource. It is important to emphasize that information being processed consciously at any one time is only a small subset of the overall cognitive workload that is being undertaken. Consciousness is a system that is fed information and sits beside and competes with a multitude of other biological systems for resources. That is not to say that if all other competing systems relented, consciousness's performance could be enhanced ad infinitum. Perhaps the most enduring observation relating to the limitations of consciousness is from 1965, when Miller suggested the magic number 7 +/- 2" (Miller 1956). This 'magic number' represents the amount of information threads or 'chunks' that could be retained in working memory at any one time. The theatre metaphor proposed by Baars suggests a spotlight of selective attention representing consciousness, into which actors move in and out, surrounded by the darkness of the unconscious, where all the other roles which facilitate the production reside (Baars, 1997). In Wiggin's adaption of Baars global workspace theory to a musical context, he asserts that his concern is not with what consciousness is, but what is presented to it. He presents a model of creative cognition in which the non-conscious mind comprises a large number of expert generators, performing parallel computations and competing to have their outputs accepted into the global workspace (Wiggins, 2012). What is not clear from this configuration is the passive/active nature of consciousness being stimulated, is this active conscious engagement or just passive awareness, this distinction is an important one in the context of live coding and instrumental improvisation because Berliner's 'creator – witness' mode of improvised behavior rather than the conscious agency needed for live coding (Berliner, 1994).

Another limitation that profoundly affects the performance of a conscious act relates to its speed of execution. In exploring this it is important to consider the differences between reflexes and reactions. Janet Chen Daniel of James Madison University explains some important distinctions, stating that only the simplest types of responses are generally regarded as 'reflexes', those that are always identical and do not allow conscious intervention, not to be confused by reactions, which are voluntary responses to a sensory stimulus. There is a long tradition of trying to quantify human performance in terms of speed of execution. In the nineteenth century one of the pioneers of experimental psychology, the German Wilhelm Wundt, developed a range of mechanical devices to measure reaction times. Probably the most well known was a device that became known as 'Wundt's complexity clock'. In 1965 German neurophysiologists Kornhuber and Deecke, discovered a phenomenon they called 'readiness potential' which suggested that the brain became active anything up to 1.5 seconds before an act was undertaken, in preparation (readiness) for the act to be performed. In the 1970s Benjamin Libet, professor of neurophysiology at the University of California Medical Center in San Francisco, took this research a stage further. Libet raised the question: if this time lag exists between preparation and the act, when do we become conscious of the act? It seemed obvious to him that there wasn't a 'one second gap' between deciding to do something and actually doing it. In 1979 he set up an experiment using Wilhelm Wundt's complexity clock to measure the time between the onset of brain activity, conscious initiation of an act and the actual act. Libet discovered that readiness potential started 550ms before the act, while the subject became conscious 20ms before the act. Libet concluded that 'cerebral initiation even of a spontaneous voluntary act of the kind studied here can and usually does begin unconsciously' (Libet 1985). From Libet's findings it became evident that the brain can

process a response to an externally initiated act more efficiently than one that is internally initiated. Our reactions are therefore quicker than our actions. In both instrumental and live coding performance contexts a performer's reflexive and reactive behaviour will impact their musical output, but the nature of its influence may differ in each circumstance. The majority of live coders tend to perform as solo artists, unlike improvising instrumentalists and so for live coders perceptual processes monitoring the performance environment are more likely to include factors external to the music, such as room acoustic, lighting, audience reaction etc., alongside anything which may directly affect the substance of the performance. The time frame in which they codify their musical expression is not necessarily synchronous with musical time and so the imperative to conceptualise, encode and transmit musical material is relieved of the '10 actions per second' restriction that Pressing identifies as the limit of improvisational novelty. Instrumentalists are responsive to subsidiary effects in the performance environment too but also to the behaviour of others actively taking part in the performance, resulting in behaviour that is likely to be stimulated by perceptual processes and object memory. In many instances an improvising musician could happily engage in verbal dialogue with someone while continuing to play their instrument but a live coder would probably struggle to talk and code at the same time. It therefore seems plausible that live coders require at their disposal, more cognitive resource to engage conceptual processes and process memory, which will tend to produce behaviour that is reactive rather than reflexive.

4. CONCLUSION

Both instrumental and live coded improvisation exists in a context where there is an expectation that novel musical material will be produced in a temporal domain that is often referred to as 'real-time' or 'in the moment'. In order to achieve this, instrumental performers have to conceptualise, encode, transmit and evaluate musical material in a manner that is perceived as a single fluid action. This action comprises cognitive processes that are fed by sensory information, implicitly held memories such as motor skills and more executive conceptual functioning. The seamless action is a fragile parameter space that is susceptible to reflexive responses and conscious and unconscious responses to internal and external stimuli. The performance being undertaken always exists within the context of available cognitive resources and could be terminated in an instant should the need arise to divert resources to a higher priority undertaking. The relationship between the temporal aspect of this undertaking and the allocation of cognitive resource differs significantly for instrumental and live coded improvisation. In both instances there is a need to access 'technique' stored as object memory but the encoding process in instrumentalists is much longer and more systematic and the resulting resource plays a much more significant part of the activity, providing elemental units of pre-programmed musical behaviour. An instrumental improviser will dynamically move up and down a continuum that stretches between perceived conscious control of individual elemental actions and executive control of pre-programmed units of material. Moving between these modes of engagement is, for the most part, outside of the performer's perception. For the live coder the need to maintain the sense of a single fluid action doesn't exist. The elemental motor skills required do not have the same relationship with the production of musical gestures and consequently the evaluation and feedback cycle is not bound to the same temporal frame. As soon as the 'real-time' shackles are loosened the freed up cognitive capacity can be utilised in musical behaviour that is guided more by compositional considerations derived from process memory and conceptual processes, with a reduced sense of automaticity, rather than being driven by more mechanical modes of musical expression.

REFERENCES

Adams, Jack A. 1976. 'Issues for a Closed-Loop Theory of Motor Learning.' *Motor Control*, 87–107.

Avery, Rachel E., and Luke D. Smillie. 2013. 'The Impact of Achievement Goal States on Working Memory.' *Motivation and Emotion* 37 (1). Springer: 39–49. doi:10.1007/s11031-012-9287-4.

Baars, Bernard J. 1997. 'In the Theater of Consciousness.' doi:10.1093/acprof:oso/9780195102659.001.1.

Bailey, Derek. 1993. *Improvisation: Its Nature and Practice in Music*. New York: Da Capo Press.

Bargh, John A, Peter M. Gollwitzer, Annette Lee-Chai, Kimberly Barndollar, and Roman Trötschel. 2001. 'The Automated Will: Nonconscious Activation and Pursuit of Behavioral Goals.' *Journal of Personality and Social Psychology* 81 (6): 1014–27. doi:10.1037/0022-3514.81.6.1014.

Baumeister, Roy F. 2004. *Handbook of Self-Regulation: Research, Theory, and Applications*. Edited by Kathleen D. Vohs and Roy F. Baumeister. New York: Guilford Publications.

Berkowitz, Aaron. 2010. *The Improvising Mind: Cognition and Creativity in the Musical Moment*. New York: Oxford University Press.

Berliner, Paul. 1994. *Thinking in Jazz: The Infinite Art of Improvisation*. 1st ed. Chicago, IL: University of Chicago Press.

Bernstien, N. 1967. *The Coordination and Regulation of Movements*. London: Pergamon.

Custers, Ruud. 2009. 'How Does Our Unconscious Know What We Want? The Role of Affect in Goal Representations.' In *The Psychology of Goals*, by Gordon B. Moskowitz and Heidi Grant, edited by Gordon B. Moskowitz and Heidi Grant. New York: Guilford Publications.

Gollwitzer, P.M., and G.B. Moskowitz. 1996. 'Goal Effects on Action and Cognition.' In *Social Psychology: Handbook of Basic Principles*, by Tory E. Higgins and Arie W. Kruglanski, edited by Tory E. Higgins and Arie W. Kruglanski. New York: Guilford Publications.

James, William. 1957. *The Principles of Psychology, Vol. 1*. New York: General Publishing Company.

Libet, Benjamin. 1985. 'Unconscious Cerebral Initiative and the Role of Conscious Will in Voluntary Action.' *Behavioral and Brain Sciences* 8 (04). doi:10.1017/s0140525x00044903.

Mendonça, David, and William Wallace. 2004. 'Cognition in Jazz Improvisation: An Exploratory Study.' In .

Miller, George A. 1956. 'Information and Memory.' *Scientific American*195 (2): 42–46. doi:10.1038/scientificamerican0856-42.

Moskowitz, Gordon B., Peizhong Li, and Elizabeth R. Kirk. 2004. 'The Implicit Volition Model: On the Preconscious Regulation of Temporarily Adopted Goals.' *Advances in Experimental Social Psychology*, 317–413. doi:10.1016/s0065-2601(04)36006-5.

Newell, K. 1991. 'Motor Skill Acquisition.' *Annual Review of Psychology*42 (1): 213–37. doi:10.1146/annurev.psych.42.1.213.

Pressing, J. 1984. 'Cognitive Processes in Improvisation.' In *Cognitive Processes in the Perception of Art*, by W. Ray Crozier and Antony J. Chapman, edited by W.Ray Crozier and Antony J. Chapman. New York, NY: Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co.

Pressing, J. 1988. 'Cognitive Processes in Improvisation.' In*Generative Processes in Music: The Psychology of Performance, Improvisation and Composition*, by John A. Sloboda, edited by John A. Sloboda. New York: Oxford University Press.

Redgrave, Peter, and Kevin Gurney. 2006. 'The Short-Latency Dopamine Signal: A Role in Discovering Novel Actions?' *Nature Reviews Neuroscience* 7 (12): 967–75. doi:10.1038/nrn2022.

Slawek, Stephen. 1998. 'Keeping It Going: Terms, Practices and Processes of Improvisation in Hindustani Music.' In *In the Course of Performance: Studies in the World of Musical Improvisation*, by ed. by Bruno Nettl with Melinda Russell., edited by Bruno Nettl and Melinda Russell. Chicago, IL: University of Chicago Press.

Sperber, Dan. 1996. *Explaining Culture: A Naturalistic Approach*. 1st ed. Cambridge, MA: Blackwell Publishers.

Wiggins, Geraint A. 2012. 'The Mind's Chorus: Creativity Before Consciousness.' *Cognitive Computation* 4 (3). Springer: 306–19. doi:10.1007/s12559-012-9151-6.

Wood, Wendy, Jeffrey M. Quinn, and Deborah A. Kashy. 2002. 'Habits in Everyday Life: Thought, Emotion, and Action.' *Journal of Personality and Social Psychology* 83 (6): 1281–97.

Wulf, Gabriele, Ph.D. 2007. *Attention and Motor Skill Learning*. 1st ed. Champaign, IL: Human Kinetics Publishers.

What Does Live Coding Know?

Geoff Cox
Aarhus University
gcox@dac.au.dk

ABSTRACT

Live coding can be seen to reflect contemporary conditions in which our lives seem to be increasingly determined by various scripts and computational processes. It demonstrates the possibility for more expansive ideas that emerge in the uncertainties of improvised performance. This paper further situates the practice of live coding in the context of artistic research and the notion of 'onto-epistemology'. One of the challenges, it is argued, is to bring code back into the frame of 'material-discursive' practice so that code can be understood for what it is, how it is produced and what it might become. This is arguably the critical task of live coding: to expose the *condition of possibility* in this way; in other words, to remain attentive to the contradictions of what constitutes knowledge in contested fields of practice, and to demonstrate modes of uncertainty in what would otherwise seem to be determinate computational processes.

INTRODUCTION

Live coding has emerged over the past decade as a dynamic creative force that has gained critical attention across cultural and technical fields (Blackwell et al, 2013), but there is little in the way of discussion about its critical potential and ability to generate new forms of knowledge. In this short paper, I want to open up this line of thinking and situate live coding in the context of artistic research and the inter-relation of epistemological and ontological domains.

By invoking the notion of *onto-epistemology*, I refer to the work of Karen Barad (2007), who combines epistemology (the theory of knowing) and ontology (the theory of being) into an ethical framework that she calls *apparatus*. Further developing the concept from the writings of Michel Foucault (Foucault 1980), in her conceptualization of apparatus, apparatuses and subject/objects mutually create and define each other. Apparatuses are not to be taken to be passive articulations of power/knowledge but instead active and "productive of (and part of) phenomena" (Barad 2007: 98). In this sense an argument around live coding can be developed that departs from the anthropomorphic tendency to situate the programmer centrally as the one who introduces uncertainty (as, for instance, through artistic intention or human error, Cox & McLean 2013: 63). Instead a position can be developed that takes into account the wider apparatuses in which human/nonhuman subjects/objects co-create and together establish uncertain outcomes. In other words, the making, doing and becoming of live code, coding and coders are materialized through the inter-action of elements.

Of course to try to discuss such complex inter-actions is an impossible task within only a few pages of a short conference paper but at least I hope to offer some starting points for further work (not least, to my own contribution to the early stages of a multi-authored book project about live coding). Part of the intention of this paper is simply to demonstrate the potential of live coding to unsettle some of the various power/knowledge regimes through which it circulates as a distinct practice, performance, experimental computer music, computational culture, aesthetic expression, and so on. The potential to undermine expectations and introduce uncertainty is particularly important, as I hope is obvious, as we are increasingly bound to closed and opaque coded systems that do not provide access to any understanding of their inner operations, let alone encourage the manipulation of them at the level of writing or reading code, or through real-time inter-actions and critical experimentation that examines power/knowledge formations.

One of the challenges then, it can be argued, is to identify code as an integral part of coding practice so that it can be understood for what it is, how it is produced and what it might become once it is materialized. This is arguably the critical task of live coding: to expose the *conditions of possibility*; in other words, to remain

attentive to the uncertainty of what constitutes knowledge in contested fields of practice, and to demonstrate these modes of uncertainty in what otherwise would seem to be determinate computational processes.

EMERGENT KNOWLEDGE

In unsettling some of the assumptions of the institutions of art and the academy, *artistic research* (or research through artistic practice) might also be useful to draw attention to the way that live coding refuses to be fixed or foreclosed by a particular knowledge domain (such as computer science, electronic music or computer music, even artistic practice). Instead it offers a challenge to some of the assumptions of what constitutes knowledge and how it might be articulated in ways where the outcomes are less certain or prescribed.

The discussion around artistic research has become fairly well established in general with the proliferation of critical writing that addresses non-propositional forms through which knowledge is disseminated in order to accommodate artistic modes of thinking and argumentation (Schwab and Borgdorff 2014), but there is little as yet that extends to artists working with computational forms and how this extends the discussion. The suggestion therefore is that non-traditional methods, such as the live inter-actions of programmer, program code and the practice of coding, further expand the range of possibilities for knowledge production and question how "various communicative and thinking 'styles' guide the flow of knowledge" (Holert 2009: 8). Decisions are made, enacted and further adapted in real-time and flow transversally.

Artistic research practice can also be seen part of the more general critique of epistemological paradigms and the possibilities for alternative forms to reshape what we know and how we know it, and to redefine some of the limits of knowledge and what escapes its confines (in the realm of alternative or non-knowledge). It becomes clear that formal epistemologies are inherently paradoxical and alternative paradigms such as live coding understood as artistic research might help to reshape how and what we know about how knowledge is produced through reflexive operations and recursive feedback loops. For example, Alex McLean's *feedback.pl* (2004), a self-modifying live code editor written in Perl, has been oft-cited as an example of this potential by establishing how both the programmer and the program are able to edit code while it runs in real-time (Yuill 2008; Cox 2011; Cox & McLean 2013). It is the inter-relation of apparatuses and subject/objects that create and define each other, and thereby the example demonstrates how neither the programmer nor program is able to make finite choices or decisions as such.

This is somewhat like *action research*, a reflective process of problem-solving, in which the act of writing or coding is made active to question what is generated in terms of research outcomes. More than this, live coding presents a further challenge to the conventions of research practices in its embrace of uncertainty and indeterminacy, including those methods that attempt to incorporate practice as a mode of research to destabilize expectations or goal-oriented approaches to research design. This sense of indeterminacy even has a mathematical foundation in the recognition of the *decision-problem* that helps to define the limits of computation. According to Alan Turing's 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem (Decision Problem)," there are some things that are incapable of computation, including problems that are well-defined and understood. Computation contains its own inner paradoxes and there is always an undecidable effect in every Turing Complete program however decidable it might appear, and it is not logically possible to write a computer program that can reliably distinguish between programs that halt and those that loop forever. The decision-problem unsettles certainties about what computers are capable of, what they can and cannot do.

Live coding might be considered to be a form of decision-making in this way in which the uncertainty of outcomes are made evident through the introduction of further 'live' elements and improvised actions outside the computer. All decisions are revealed to be contingent and subject to internal and external forces that render the performance itself undecidable and the broader apparatus an active part of the performance. It becomes clear that research practices, methodologies and infrastructures of all kinds are part of wider material-discursive apparatuses through which emergent forms of knowledge is produced and circulated. For example, as with all coding practices, the distinction between the writing and the tool with which the writing is produced is revealed to be misleading (Cramer 2007), and neither academic writing nor program code can be detached from their performance or the way they generate knowledge as part of larger

apparatuses that create and attempt to define their actions (even when they fail). With the example of *feedback.pl*, the subject/objects relations become confused and it becomes evident that these processes are open to further change and modification at all levels of the operation.

Furthermore, live coding, like all coding, is founded on collective knowledge, or what Paolo Virno (after Marx) would identity as *general intellect* as the "know-how on which social productivity relies," as an "attribute of living labour" (2004: 64-5). This know-how refers to the combination of technological expertise with social or general knowledge, and recognizes the importance of technology in social organization. This is something I have written about previously, to describe how the contemporary linguistic and performative character of labour and coding practices combine to produce an uncertain relation between code objects and code subjects — between code and coders — the "means-end of code" (Cox 2011). The politics of this extends to various communities of socio-technical practice such as the way that source code is distributed and shared through the social movements around software (Free Software) and various legal instruments (such as Copyleft software licenses). It also extends to the use of code repositories like Git and GitHub that offer distributed revision control and source code management, including collaborative working tools such as wikis, task management and bug tracking (https://github.com/). Such examples exist as part of a political spectrum of 'just-in-time production' where performative and improvisational techniques have been put to work.

There is an excellent essay by Simon Yuill, "All Problems of Notation will be Solved by the Masses" (using Cornelius Cardew's phrase), that develops similar connections to free software development and collaborative practices, including comparison with the improvisation techniques of the Scratch Orchestra in the late 1960s: "Livecoding works from within a particular relation between notation and contingency. The specificity of code is opened towards the indeterminism of improvisation." (Yuill 2008) Indeterminacy extends to the way that knowledge is produced also through improvisatory techniques, and the discussion of artistic practice can be partly characterised in this way as emergent knowledge production in real-time.

If further reference is made to Foucault's *Archaeology of Knowledge* (1972), this contingent aspect is already well established perhaps in a general way, and thereby what constitutes knowledge is open to question, and various strategies have been developed to uncover otherwise undiscovered, in counter-histories, emergent knowledge or non-knowledge. Yet, additionally, it becomes necessary to also conceptualize knowledge that is less human-centred to include ways of understanding and acting in the world that exceed what is normally seeable, audible, readable and knowable. This requires *media* and not just humans becoming "active archaeologists of knowledge," as Wolfgang Ernst puts it (2011: 239). Thus what is already understood as *archaeology of knowledge* is extended to *media archaeology* to include what can be made knowable beyond the human sensory apparatus in the nondiscursive realm of technical infrastructures and apparatuses. Live coding might similarly expand the range of possibilities for alternative epistemologies in allowing for discursive and *nondiscursive* knowledge forms.

The example Ernst gives is *Fourier analysis* as able to isolate individual components of a compound waveform, concentrating them for easier detection or removal. He considers the "operative machine" as able to perform a better cultural analysis than the human is capable of (Ernst 2011: 241). Furthermore, with reference to Ernst and the concept of *microtemporality*, Shintaro Miyazaki introduces the concept algo*rhythm* to emphasize how the very processes of computation contain epistemic qualities. He explains:

"An algo*rhythmic* study cultivates a close reading of, or to be more accurate, a close *listening to* the technical workings of computers and their networks on precisely this deep level of physical signals. Analysed on different scales from the microprogramming on the level of the CPU, or the physical layer of communication protocols between server and clients, to interface applications on all levels where human interaction is needed, to longer time spans such as the daily backing up of large amounts of data, distributed and networked algo*rhythmic* processes can be found on many levels of our current informational networks and computational cultures." (Miyazaki 2012)

The algo*rhythmic* approach offers an epistemic model of a machine that makes time itself logically controllable and, while operating, produces measurable effects and rhythms. The practice of live coding –

and more overtly *Algoraves* – resonates with these descriptions in drawing together dynamic processes, signals, rhythms, temporalities and human bodies coding and dancing.

Part of this line of thinking involves the necessity of developing a fuller understanding of onto-epistemological operations. Without this know-how, we would simply miss the detail on how various effects are produced, how human/nonhuman relations and rhythms operate together, and how codes are embedded in wider systems of control and command. In *Philosophy of Software*, David M. Berry suggests that we need to pay more attention to the "computationality of code," to understand code as "an ontological condition for our comportment towards the world" (Berry 2011: 10, 13). If code becomes the object of research in this way, its ontological dimension becomes crucially important for the way that code is understood as a condition of possibility (Ibid: 28). An onto-epistemology of live coding would therefore involve inter-actions of discursive and nondiscursive forms – to challenge how we conceptualize code-being and code-becoming. This essay aims to stress this onto-epistemological condition: that code both performs and is performed through the practice of coding in real-time. Both mutually create and define each other, inter-acting in indeterminate and uncertain ways.

UNCERTAIN ENDING

It is with his condition of uncertainty that I would like to end this essay echoing Ernst's reference to stochastic mathematics, related to the theory of probability, to stress that there is an indeterminism between human and nonhuman knowledge that comes close to the *uncertainty principle* (2011). The uncertainty principle asserts that no thing has a definite position, a definite trajectory, or a definite momentum, and that the more an attempt is made to define an object's precise position, the less precisely can one say what its momentum is (and vice versa). Things are known and unknown at the same time.

Foucault's notion of apparatus is useful in this way as it allows for an analysis of power/knowledge through understanding things to be part of larger relational systems. However Barad extends the conceptualization by insisting that apparatuses are "themselves material-discursive phenomena, materializing in intra-action with other material-discursive apparatuses" (Barad 2007: 102). To further explain:

"Although Foucault insists that objects (subjects) of knowledge do not preexist but emerge only within discursive practices, he does not explicitly analyze the inseparability of apparatuses and the objects (subjects). In other words, Foucault does not propose an analogue to the notion of phenomenon or analyze its important (ontological as well as epistemological) consequences." (Barad 2007: 201)

An onto-epistemological reading insists on broader notions of subjects and objects as well as their inter-actions. This is made clear with respect to how technologies allow for discursive and nondiscursive knowledge forms. Again quoting Barad in detail makes this apparent:

"Knowing is not about seeing from above or outside or even seeing from a prosthetically enhanced human body. Knowing is a matter of inter-acting. Knowing entails specific practices through which the world is differentially articulated and accounted for. In some instances, 'nonhumans' (even being without brains) emerge as partaking in the world's active engagement in practices of knowing. [...] Knowing entails differential responsiveness and accountability as part of a network of performances. Knowing is not a bounded or closed practice but an ongoing performance of the world." (Barad 2007: 149)

Drawing on the new materialism of Barad, and the media archaeology of Ernst and Miyazaki in this essay, allows me to assert that it is simply not possible to generate knowledge outside of the material and ontological substrate through which it is mediated. Thus the inter-relation of epistemology and the ontological dimension of the materials, technologies, things, code and bodies is established as active and constitutive parts of meaning-making. Live coding offers a useful paradigm in which to establish how the know-how of code is exposed in order to more fully understand the experience of the power-knowledge systems we are part of, and to offer a broader understanding of the cultural dynamics of computational culture as an apparatus, or set of apparatuses, with ontological as well as epistemological consequences.

Live coding can be seen to reflect contemporary conditions in which our work and lives seem to be increasingly determined by computational processes. More expansive conceptions and forms that emerge in

the uncertainties of performance become part of the critical task for live coding: to expose the conditions of possibility, to remain attentive to the contradictions of what constitutes knowledge and meaning. An onto-epistemology of live coding thus offers the chance to engage with the inner and outer dynamics of computational culture that resonates in the interplay of human and nonhuman forces. It helps to establish uncertainty against the apparent determinism of computation and how we think we know what we know.

REFERENCES

Barad, K. 2007. *Meeting the Universe Halfway*. Durham & London: Duke University Press.

Berry, D. M. 2011. The *Philosophy of Software: Code and Mediation in the Digital Age*. Basingstoke: Palgrave Macmillan.

Blackwell, A., McLean, A. Noble, J. and Rohrhuber, J. eds. 2013. *Dagstuhl Report: Collaboration and learning through live coding*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=13382

Cox, G. 2011. "Mean-End of Software." *Interface Criticism: Aesthetic Beyond Buttons*. Pold, S. and Andersen, C. U. eds. Arhus: Aarhus University Press, 145-161.

Cox, G. and McLean, A. 2013. *Speaking Code: Coding as Aesthetic and Political Expression*. Cambridge, Mass.: MIT Press.

Cramer, F. 2007. "(echo echo) echo (echo): Command Line Poetics." http://reader.lgru.net/texts/echo-echo-echo-echo-command-line-poetics/

Ernst, W. 2011. "Media Archaeography: Method and Machine versus History and Narrative of Media." *Media Archaeology: Approaches, Applications and Implications*, Hutamo, E. and Parikka, J. eds. Berkeley: University of California Press.

Foucault, M. 1972. *The Archaeology of Knowledge*, London: Routledge.

Foucault, M. 1980. *Power/Knowledge Selected Interviews and Other Writings*, Colin Gordon, London: Routledge.

Holert, T. 2009. "Art in the Knowledge-based Polis." *e-flux journal* 3. http://www.e-flux.com/journal/art-in-the-knowledge-based-polis/

Miyazaki, S. 2012. "Algorhythmics: Understanding Micro-Temporality in Computational Cultures." *Computational Culture* 2. http://computationalculture.net/article/algorhythmics-understanding-micro-temporality-in-computational-cultures

Schwab, M. and Borgdorff, H. 2014. "Introduction." *The Exposition of Artistic Research: Publishing Art in Academia*. Leiden University Press, 9-20.

Turing, A. 1936. "On Computable Numbers, with an Application to the Entscheidungsproblem (Decision Problem)." https://www.wolframscience.com/prizes/tm23/images/Turing.pdf

Virno, P. 2004. *A Grammar of the Multitude: For an Analysis of Contemporary Forms of Life*. New York: Semiotext(e).

Yuill, S. 2008. "All Problems of Notation Will be Solved by the Masses." *MUTE*.
http://www.metamute.org/editorial/articles/all-problems-notation-will-be-solved-masses

Approximate Programming: Coding Through Gesture and Numerical Processes

Chris Kiefer
Department of Informatics, University of Sussex, UK.
c.kiefer@sussex.ac.uk

ABSTRACT

Approximate programming is a novel approach to live coding that augments traditional programming methods with methods of generating and editing code through realtime numerical processes, using an underlying system that employs representations and transformations from gene expression programming. It aims to provide a hybrid environment where code can be created and modified expressively with multiparametric controllers, and well as with conventional text editing tools. It does this while aiming to keep the code as the central point of representation in the system. Two case studies are presented where the system has been used in live performance for musical improvisation and then for generative audiovisualisation. Initial trials of the system highlight its strengths as an embodied method for control of complex code structures, and as a novel method for combining low-level conceptual structures into higher-level forms. The case studies show two key limitations of the system, with challenges in comprehension of the final code output in text form, and difficulties arising from the highly nonlinear nature of the input-output mappings. Initials solutions are presented in the form of a GUI system for interacting with code in tree representation form.

1. Introduction

Approximate programming is a novel approach to interacting with code in creative software systems. It merges techniques from live coding, gene expression programming (Ferreira 2001), multiparametric control (Kiefer 2012) and parameter space exploration (Dahlstedt 2009). Code is generated in realtime through a transformation of an array of numbers analogous to a gene in evolutionary programming systems. This gene can be created by a musical controller, allowing the player to interact with code though gesture and motion capture. The gene can also be drawn from other data sources, for example from realtime music information retrieval systems, allowing code to be generated from any computational process.

The system attempts to bring expressive bodily gesture into the process of interacting with code, as an augmentation to the conventional coding editing interfaces typically used for live coding. It aims to do this while preserving code as the primary focus of the system, to harness the flexibility of code, which increasing the involvement of the body in its creation. The system is named *approximate* programming because the results for a particular gesture or gene are, at least initially, unknown, although the broad domain may be predictable based on knowledge of the gene-to-code mappings. The system is primarily exploratory, although exploration transitions towards repeatable performance as the search space is mapped out by the player.

The development of this technique was motivated by a desire to find ways to express code responsively in 'musical' time. Further details of the background to this project are given in (Kiefer 2014). The author has used the system in performance in two different contexts: for musical performance, and as an autonomous engine for audiovisualisation. Both of these contexts are presented here as case studies of the development of approximate programming techniques. Following this, the strengths and pitfalls of approximate programming are discussed, along with the development of new extensions to address the key limitations of the system.

2. Context

This work is rooted in the field genetic programming (GP) (Poli, Langdon, and McPhee 2008), and more closely to a variant of GP, gene expression programming (Ferreira 2001), taking the idea of expressing code as a transformation of a vector of numbers and using this vector to explore a large non-linear space of possibilities through a mapping of the gene to a

tree representation of the code. The similarities however end with the use of this representation, as evolutionary search techniques are not used in approximate programming. Instead, the space of possibilities is explored in realtime, as the player navigates through the parameter space. The search space can be re-modelled, expanded or restricted by changing the set of component functions on which the transformation draws. This process places the technique as a hybrid of a parametric space exploration system (Dahlstedt 2009, ???) and interactive evolution (Dorin 2001), where a human is at the centre of the search process. Within the field of live coding, approximate programming overlaps with several other projects and pieces. *Ixi Lang* (Magnusson 2011) has a self-coding feature, where the system runs an agent that automatically writes its own code. D0kt0r0's Daemon.sc piece (Ogbourn 2014) uses a system that generates and edits code to accompany a guitar improvisation. [Sisesta Pealkiri] (Knotts and Allik 2013) use a hybrid of gene expression programming and live coding for sound synthesis and visualisation. McLean et al. (2010) explore the visualisation of code; this is a theme that has emerged as an important issue in approximate programming, where code changes quickly, and robust visualisation techniques are required to aid comprehension by the programmer.

3. Approximate Programming

Approximate programming takes two inputs: a numeric vector (the gene) and an array of component functions. It outputs the code for a function built from the component functions, according to the content of the gene. It is embedded within a system that compiles and runs each newly generated function, and seamlessly swaps it with the previously running function as part of a realtime performance or composition system. It should be noted that the use of the term gene here does not imply that an evolutionary search process is used to evolve code, but this term is still used because of the relevance of other genetic program concepts. The component functions are typically low level procedures, including basic mathematical operations, but may also include larger procedures that represent more complex higher level concepts. The algorithm works as follows:

```
create an empty list of nodes that contain numerical constants, P
choose a component function, C, based on the first element in the gene
find out how many arguments C has
while there are enough gene values left to encode C and its arguments
    create a new tree node to represent this function
        use the next values in the gene as parameters for the function
        add the parameters as leaf nodes of the new tree
    add the parameter nodes to the list P
    if the tree is empty
        add this new tree at the root
    else
        use the next gene value as an index to choose a node containing a function parameter from P
        replace this parameter node with the new tree
        remove the parameter node from P
    choose a new component function, C, using the next gene value as an index
    find out how many arguments C has
```

Broadly, the algorithm builds a larger function from the component functions by taking constant parameters in the tree and replacing them with parameterised component functions. A SuperCollider implementation of this algorithm is available at https://github.com/chriskiefer/ApproximateProgramming_SC.

4. Case Studies

4.1. Case Study 1: Musical Improvisation with SuperCollider and a Gestural Controller

The first use of this system was for a live musical performance. The system was built in SuperCollider. During the performance, the component functions were live coded, while the gene was manipulated using a multiparametric gestural controller. This controller output a vector of 48 floating point numbers, at around 25Hz, and these vectors were used as genes to generate code. This is an example of some component functions:

```
~nodes[\add] = {|a,b| (a+b).wrap(-1,1)}
```

```
~nodes[\mul] = {|a,b| (a*b).wrap(-1,1)}
~nodes[\sin] = {|x,y| SinOsc.ar(x.linexp(-1,1,20,20000), y)}
~nodes[\pulse] = {|x,y,z| Pulse.ar(x.linexp(-1,1,20,20000),y, z.linlin(-1,1,0.2,1))}
~nodes[\pwarp] = {|a| Warp1.ar(1,~buf2, Lag.ar(a,10), 1, 0.05, -1, 2, 0, 4)}
~nodes[\imp] = {|a| Impulse.ar(a.linlin(-1,1,0.2,10))}
~nodes[\imp2] = {|a| Impulse.ar(a.linlin(-1,1,3,50))}
```

This example shows a range of functions from low level numerical operations to higher level functions with more complex unit generators. These functions could be added and removed from a list of the current functions that the system was using to generate code. The system generated code that was compiled into a SuperCollider SynthDef. Each new Synth was run on the server and crossfaded over the previous Synth. An example of the generated code is as follows:

```
~nodes[\add].(
    ~nodes[\pulse].(
        ~dc.(0.39973097527027 ),
        ~nodes[\pulse].(
            ~nodes[\pwarp].(
                ~nodes[\pwarp].(
                    ~dc.(0.6711485221982 ))),
            ~nodes[\pwarp].(
                ~nodes[\mul].(
                    ~dc.(0.6784074794054 ),
                    ~nodes[\saw].(
                        ~dc.(0.999 ),
                        ~dc.(0.999 )))),
            ~dc.(0.5935110193491 )),
        ~dc.(0.39973097527027 )),
    ~dc.(0.38517681777477 ))
```

4.2. Case Study 2: Generative Audiovisualisations from Realtime MIR Data

A new version of the approximate coding system was developed to create openGL shaders, using GLSL shading language. This system worked in a very similar way to the SuperCollider system; a set of components functions was present in the source of each shader, and a larger scale function was generated from a gene, consisting of calls to these component functions. Visuall, the system took a functional rendering approach; it created pixel shaders, which took a screen position as input and output the colour value for that position. The intention of this system was to create audiovisualisations, and audio was used as input to the approximate coding system in two ways. Firstly, the gene was generated from sections of audio; every bar, an analysis was carried out to generate a new gene from the previous bar of audio. The intention was that similar bars of audio should generate similar visual output, and various combinations of audio features were experimented with for this purpose. Secondly, an MFCC analyser processed the realtime audio stream and streamed its feature vector to the fragment shader; component functions in the approximate coding system drew on this data, so the resulting pixel shader reacted to the textures in the realtime audio stream. The system was fine tuned with a preset library of component functions, and then used for a live performance where it autonomously generated the audiovisualisation. The system was built using OpenFrameworks.

4.3. Discussion

These two case studies complement each other well as they explore the system in two very different use cases. This demonstrates the flexibility of approximate programming, and also highlights limitations of the system that need to be addressed.

To begin with, I will explore the benefits of using this system. The principle gain from this system is the ability to explore relatively complex algorithms in a very intuitive, bodily way, in stark contrast to the cerebral approach that would be required to generate the same code manually. Furthermore, the process of search and exploration places the programmer/players sensory experience at the forefront of the process, and externalises rather than internalises the creative process. Live coding the component functions provides a mid-point between these two poles, linking programming and sensory exploration. The component functions are interesting as they represent low-level concepts in the search

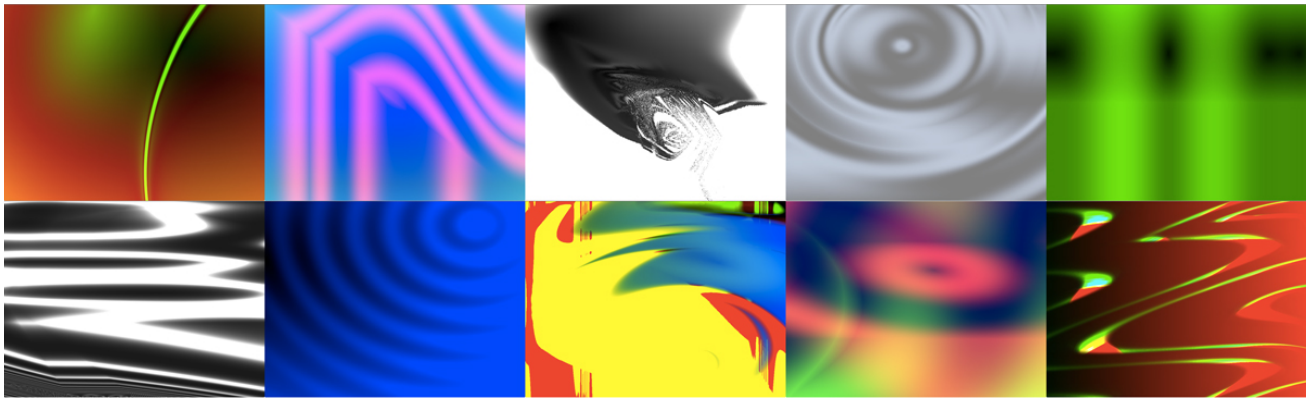


Figure 1: *Screenshots of the Audiovisualiser*

space that are transformed and combined into a higher-level concepts in the output. For example, in the sound synthesis version of the system, if sine wave generators, adders and multipliers are provided as components functions, an FM synthesis type output will be obtained. If impulse generators are added, then rhythmic outputs start to appear. In the graphical version, there is a clear mapping between low-level numerical operations and visual features that appear in the outputs, for example sine generators create circular forms, and euclidean distance functions create colour gradients. The dynamics of programming components and exploring them creates an interesting potential for content creation and performative exploration of complex multi-dimensional search spaces. To further this, the code paradigm is preserved throughout the process, and the performer can examine the resultant code output as part of the exploration. This makes the system transparent to the user, and very malleable as the results are reusable and recyclable.

There are two key limitations of the system that need to be addressed, concerning nonlinearity of the search space and interaction with the code. In case study 1, the resultant code from the approximate programming system was projected to the audience, and was displayed on the computer used for the performance to be used as part of the navigation process. The code was laid out as in the example above, but without colour coding. It was the intention to use this code as part of the reference to exploring the search space, in keeping with the design of the system which places the code form at the forefront of the system. The code however moved very fast (in this case this was partially due to the instability of the controller being used), and without visual augmentations it was difficult to comprehend at the speed it was being presented. It would be ideal if this code could be used as a significant part of the approximate programming process, and so visual augmentations to the output code are a key priority to address. The second issue was most apparent in the audiovisual system, and concerns the nonlinearity of the search space. The intention of the audiovisual system was to generate code based on analysis of sections of music in such a way that similar sections would generate similar audiovisualisations. This worked sometimes, but because of the nonlinearity of the transformation between gene and phenotype, often similar bars would create very different audiovisualisations, implying that the search landscape is far from smooth. Exploring methods for smoothing out this landscape is the second key issue to address with this system, and the most challenging one. Attempts towards addressing the first challenge of increasing engagement with the system output are now presented.

5. Engaging with the Outputs of Approximate Programming

Two new augmentations to the approximate programming system will now be described, that aim to increase potential for engagement with the code output both visually and interactively.

5.1. Visual Augmentations

The original presentation of code output by the system was difficult to comprehend - although it had some basic formatting, there were few visual aids, and fast changes were challenging to follow. Showing this output as text was perhaps one step to far in visualising the output, and this new augmentation rewinds the process to the tree representation of the code, before it's converted to text. The logic of the code is still present in this form, but without the trappings of textual syntax it's possible to present a logical process in a simpler way that aids comprehension by the performer. Furthermore, the system at the moment does not have a facility for tracking direct edits to the code output, so there isn't a strong case for viewing it as text anyway.

The new system presents the code tree graphically, as shown in figure 2. It uses two features to enhance comprehension. Firstly, numbers have a dual representation, as text and as a colour mapping on the background of their text box. This allows an easy overview of parameter values. Secondly, significant nonlinearities in the search space are caused when a function placeholder changes to point to a different component function. It's helpful to the player to know when this might happen, so function identifiers are also colour coded; when their underlying numerical value is sitting between two functions, they appear grey. As their underlying value moves in either direction towards a boundary where the function will change, the background is increasingly mixed with red. This gives an indication of when the code is in a more linear state, and also indicates when it's approaching a significant change.

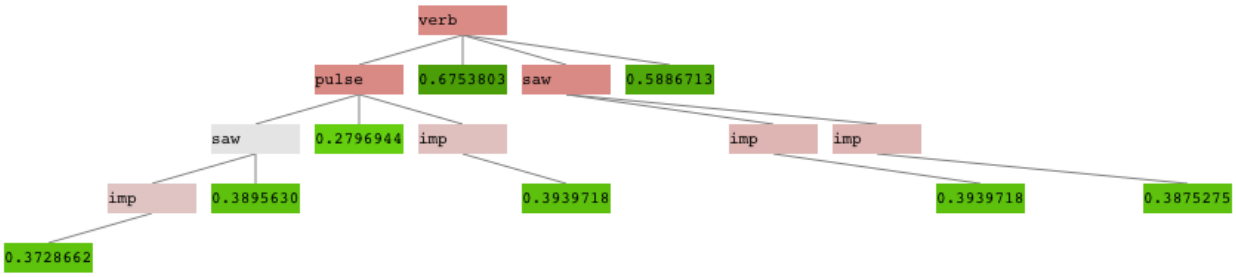


Figure 2: *A visualisation of the code tree*

The tree in figure 2 corresponds to the code below:

```
~nodes[\verb].(
    ~nodes[\pulse].(
        ~nodes[\saw].(
            ~nodes[\imp].(
                ~dc.(0.37286621814966 )),
            ~dc.(0.38956306421757 )),
        ~dc.(0.27969442659616 ),
        ~nodes[\imp].(
            ~dc.(0.39397183853388 ))),
    ~dc.(0.67538032865524 ),
    ~nodes[\saw].(
        ~nodes[\imp].(
            ~dc.(0.39397183853388 )),
        ~nodes[\imp].(
            ~dc.(0.38752757656574 ))),
    ~dc.(0.58867131757736 ))
```

5.2. Interacting with the Code Tree

Further to aids for visual comprehension of code, the code tree presents an opportunity for the programmer to further interact with the system. A new feature was added augment the code tree through mouse interaction together with gestural control. It works as follows:

- The player chooses a point on the code tree with the mouse that they would like to change.
- They also choose the size of the gene they will use for this change.
- When they click on the node, a new sub-tree is spliced into the main code tree. Now, all editing changes this sub-tree only, and the rest of the tree is kept constant.

The player can use this system to slowly build up code trees, with either fine-grained or coarse adjustments.

5.3. Discussion

The new augmentations clarify code representation, and allow new interactions that give the player more refined control over the system if this is desired. Within this new framework, there's plenty of room to explore new ways to combine code editing from multiple points of interaction: text editing, GUI controls, physical controllers and from computational presses. This new part of the system is work-in-progress, and it currently under evaluation by the author in composition and live performance scenarios.

6. Conclusion

A new *approximate programming* system has been presented, which aims to create a hybrid environment for live coding which combines traditional code editing, embodied physical control, GUI style interaction and open connections to realtime computational processes. It has been explored in two case studies, one that used the system for musical improvisation in SuperCollider, and one that ran autonomously creating audiovisualisations using GLSL. The two case studies show that the system, from the authors perspective, has particular strength in enabling embodied control of code, and in augmenting live coding with gestural control. There are two key limitations of the system; challenges in reading and interacting with the final code output that it creates, and the non-linearity of the search space. This paper offers initial solutions to enabling higher quality engagement with code outputs, by presenting the code tree graphically instead of presenting the code in textual form. A further augmentation allows the player to use a GUI to select and edit sections of the tree. These augmentations are currently under evaluation, and future work will explore this area further, along with methods for managing nonlinearity in the search space.

Overall, this project has plenty of research potential in interface design, audio and visual synthesis and performance aesthetics. The author intends to make a future public release to gather more data on how musicians and visual artists would use this type of technology.

References

Dahlstedt, Palle. 2009. "Dynamic Mapping Strategies for Expressive Synthesis Performance and Improvisation." In *Computer Music Modeling and Retrieval. Genesis of Meaning in Sound and Music: 5th International Symposium, CMMR 2008 Copenhagen, Denmark, May 19-23, 2008 Revised Papers*, 227–242. Berlin, Heidelberg: Springer-Verlag. doi:http://dx.doi.org/10.1007/978-3-642-02518-1_16.

Dorin, Alan. 2001. "Aesthetic Fitness and Artificial Evolution for the Selection of Imagery from the Mythical Infinite Library." In *Advances in Artificial Life*, 659–668. Springer.

Ferreira, Cândida. 2001. "Gene Expression Programming: a New Adaptive Algorithm for Solving Problems." *Complex Systems* 13 (2): 87–129.

Kiefer, Chris. 2012. "Multiparametric Interfaces for Fine-Grained Control of Digital Music." PhD thesis, University of Sussex.

———. 2014. "Interacting with Text and Music: Exploring Tangible Augmentations to the Live-Coding Interface." In *International Conference on Live Interfaces*.

Knotts, Shelly, and Alo Allik. 2013. "[Sisesta Pealkiri]." Performance, live.code.festival, IMWI, HfM, Karlsruhe, Germany.

Magnusson, Thor. 2011. "Ixi Lang: a SuperCollider Parasite for Live Coding." In *Proceedings of International Computer Music Conference*, 503–506. University of Huddersfield.

McLean, Alex, Dave Griffiths, Nick Collins, and G Wiggins. 2010. "Visualisation of Live Code." In *Electronic Visualisation and the Arts*.

Ogbourn, David. 2014. "Daem0n.Sc." Performance, Live Coding and the Body Symposium, University of Sussex.

Poli, Riccardo, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A Field Guide to Genetic Programming*. Published via `http://lulu.com`; freely available at `http://www.gp-field-guide.org.uk`. http://dl.acm.org/citation.cfm?id=1796422.

Copy-Paste Tracking: Fixing Spreadsheets Without Breaking Them

Felienne Hermans
Delft University of Technology
f.f.j.hermans@tudelft.nl

Tijs van der Storm
Centrum Wiskunde & Informatica
storm@cwi.nl

ABSTRACT

Spreadsheets are the most popular live programming environments, but they are also notoriously fault-prone. One reason for this is that users actively rely on copy-paste to make up for the lack of abstraction mechanisms. Adding abstraction however, introduces indirection and thus cognitive distance. In this paper we propose an alternative: copy-paste tracking. Tracking copies that spreadsheet users make, allows them to directly edit copy-pasted formulas, but instead of changing only a single instance, the changes will be propagated to all formulas copied from the same source. As a result, spreadsheet users will enjoy the benefits of abstraction without its drawbacks.

1. Introduction

Spreadsheet systems can easily be considered the most successful form of programming. Winston (Winston 2001) estimates that 90% of all analysts in industry perform calculations in spreadsheets. Spreadsheet users perform a range of diverse tasks with spreadsheets, from inventory administration to educational applications and from scientific modeling to financial systems. The financial business is a domain where spreadsheets are especially prevailing. Panko (Panko 2006) estimates that 95% of U.S. firms, and 80% in Europe, use spreadsheets in some form for financial reporting.

Researchers have argued that the *liveness* characteristics of spreadsheets have contributed to the widespread success of spreadsheets (Hermans 2013) and we know from interviews with users that liveness is important to them. They often start building a spreadsheet with the end goal in mind, and manipulate the formulas until they obtain the result they want.

The liveness characteristics of spreadsheets can be divided in two categories:

- Direct manipulation: instead of editing a separate plan or program to achieve some result, the spreadsheet user edits the "thing itself": there is almost no distinction between the actual data and the "code" of a spreadsheet. This feature addresses the "gulf of execution" which exists between the user's goal, and the steps that are required to achieve that goal (Norman 1986).
- Immediate feedback: after a change to the spreadsheet data or formulas, the user can immediately observe the effect of the edit. This feature bridges the "gulf of evaluation" which exists between performing an action and receiving feedback on the success of that action (Norman 1986).

Despite these attractive features for end-users, spreadsheets are well-known to be extremely fault-prone (Panko 2006). There are numerous *horror stories* known in which organizations lost money or credibility because of spreadsheet mistakes. TransAlta for example lost US $24 Million in 2003 because of a copy-paste error in a spreadsheet (Pryor). More recently, the Federal Reserve made a copy-paste error in their consumer credit statement which, although they did not make an official statement about the impact, could have led to a difference of US $4 billion (Durden). These stories, while single instances of copy-paste problems in spreadsheets do, give credibility to the hypothesis that copy-paste errors in spreadsheets can greatly impact spreadsheet quality.

This copy-pasting as in the stories above is not always done by mistake. Rather, we see spreadsheet users using copy-pasting as a deliberate technique. This is understandable, as standard spreadsheets do not support any form of data schema or meta model, so there is no way in which a new worksheet in a spreadsheet could inherit or reuse the model of an existing worksheet. Copy-paste is then often used to compensate for for the lack of code abstractions (Hermans et al. 2013). Finally, when faced with spreadsheets they do not know, users are often afraid to modify existing formulas, thus

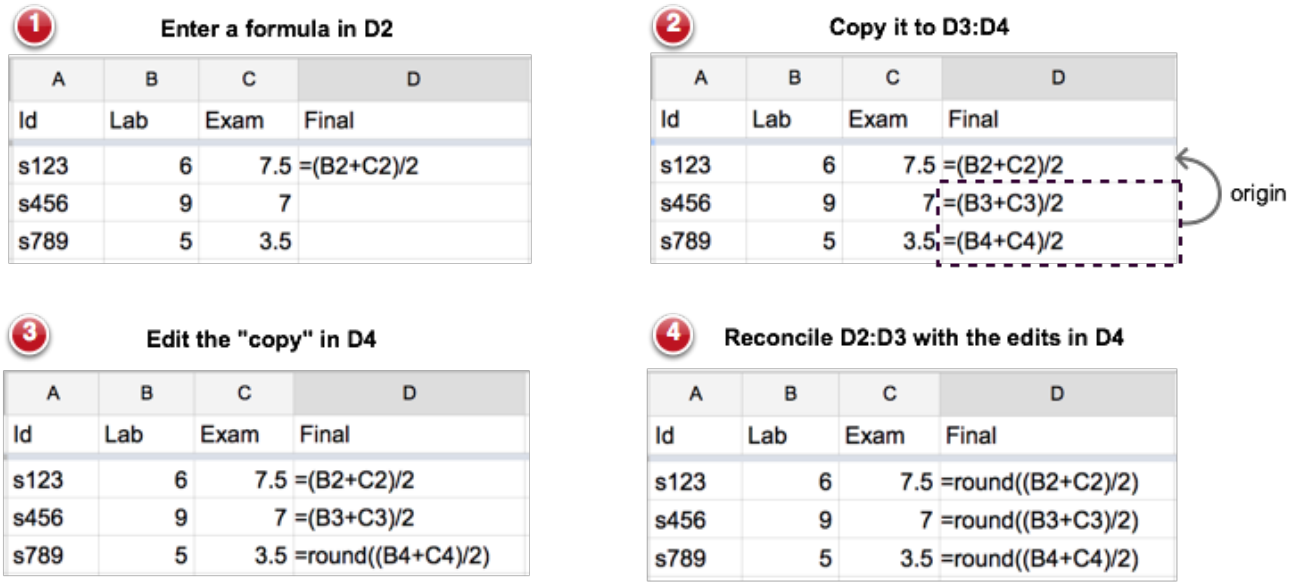
copy-paste them and add new functionality (Hermans et al. 2013) creating many versions of similar formulas, of which the origin can no longer be determined.

Existing research improving spreadsheets has focused on extending spreadsheets with abstraction mechanisms. An example of this is the work of Engels *et al.* , who have developed a system called ClassSheets (Engels and Erwig 2005) with which the structure of a spreadsheet can be described separately. The actual spreadsheet can then be guaranteed to conform to the meta description. Another direction is enriching spreadsheets with user-defined functions (UDFs) (Jones, Blackwell, and Burnett 2003). In this case, spreadsheets users can factor out common computations into separate cells, and refer to them from elsewhere in the spreadsheet.

Although these features improve the reliability of spreadsheet use, they have one important drawback, namely, that they break the "direct manipulation" aspect of spreadsheets. In a sense, separate meta models, or user defined abstractions, create distance between the actual user's artifact (data + formulas), and its computational behaviour. Instead of just looking at the cells, the user now has to inspect at least two places: the cells containing the data and the separate definitions of the abstractions (meta model or user defined functions).

In this paper we propose XanaSheet, a spreadsheet system that features an alternative method to manage abstraction, without diminishing directness. XanaSheet employs *origin tracking techniques* to maintain a live connection between source and destination of copy-paste actions. Whenever a copied formula is edited, the modifications are transformed and replayed on the original and all other copies. Instead of introducing another level of indirection using abstraction, XanaSheet allows users to edit classes of formulas, all at once. In a sense, the abstraction, or user defined function, is there, but it never becomes explicit. By retaining ease of use, this technique has the potential to eliminate a large class of copy-paste errors, without compromising the direct manipulation aspect that make spreadsheets so attractive.

2. Copy-Paste Tracking in Action



	A	B	C	D
1	Id	Lab	Exam	Final
2	s123	6	7.5	=(B2+C2)/2
3	s456	9	7	
4	s789	5	3.5	

	A	B	C	D
1	Id	Lab	Exam	Final
2	s123	6	7.5	=(B2+C2)/2
3	s456	9	7	=(B3+C3)/2
4	s789	5	3.5	=(B4+C4)/2

	A	B	C	D
1	Id	Lab	Exam	Final
2	s123	6	7.5	=(B2+C2)/2
3	s456	9	7	=(B3+C3)/2
4	s789	5	3.5	=round((B4+C4)/2)

	A	B	C	D
1	Id	Lab	Exam	Final
2	s123	6	7.5	=round((B2+C2)/2)
3	s456	9	7	=round((B3+C3)/2)
4	s789	5	3.5	=round((B4+C4)/2)

Figure 1: *Maintaining consistency among clones of formulas through copy-paste tracking*

Figure 1 shows an example user interaction with a spreadsheet containing student grades. In the first step the sheet contains just the Lab and Exam grades of three students, and a formula for computing the average of the two grades in D2. In the second step, the formula in cell D2 is copied to D3 and D4. D3 and D4 are clones of D2, and this relation is maintained by the system as an origin relation (visualized using the arrow). In the third step, the clone in D4 is modified to apply rounding to the computed average. Unlike in normal spreadsheets, however, this is not the end of the story and XanaSheet will reconcile the original formula of D2 and the other clone in D3 with the changes in D4.

A way to understand what is happening here, is to see spreadsheet formulas as materialized or unfolded abstractions. The abstraction in Fig. 1 is function `average(x,y)` for computing the average of two grades. In ordinary programming such a function could, for instance, be mapped over a list of pairs of grades to obtain a list of averages, like `map(average, zip(Lab, Exam))`. In the spreadsheet of Fig. 1, however, the abstraction `average` does not really exist, but is represented collectively by the set of all its inlined applications, e.g. `[(Lab[0]+Exam[0])/ 2, (Lab[1]+Exam[1])/2,`

`(Lab[2]+Exam[2])/2]`. In a sense, each application is a clone of the same implicit prototype, with parameters filled in with concrete data references. The tracking relation induced by copy-paste actions, identifies which clones belong to the same equivalence class. Therefore, editing one clone triggers updating the clones which belong to the same class.

In some cases it might actually not be desired to maintain the origin links between source and destination of copy-paste actions. XanaSheet supports these situations by providing a special "Paste and Detach" action which severs the copy from its original (similar to "Past and Match Style" common in many text editing systems). The example also assumes that when a user edits a formula she always intends to edit the whole class of clones. However, the system allows the user to edit only this copy, or all copies at once (similar to changing "Recurring events" in calendar applications).

3. Semantics of Copy-Paste Tracking

The previous section introduced copy-paste tracking from the perspective of the user. In this section we describe our considerations regarding the implementation. We have implemented an executable semantics of copy-paste tracking for simulating interactive editing sessions with a spreadsheet. The code can be found online here: https://github.com/Felienne/LiveSpreadsheets/tree/master/XanaSheet. We are currently working on an interactive prototype of XanaSheet.

A spreadsheet is a rectangular grid of cells where each cell is identified by its *address*, which are pairs An consisting of a column letter A and a row index n. User actions always operate on one of more of these addresses. The origin relation between cells is then modeled as a binary relation between such addresses. For instance, the relation $Org = \{\langle D3, D2\rangle, \langle D4, D2\rangle\}$ captures the origin relation visualized in Figure 1 (2). In this case, the relation states that the formulas in cell $D3$ and cell $D4$ are copied from cell $D2$.

Without loss of generality we assume users only use relative cell referencing in formulas. That is, a cell reference consists of relative row and column offsets starting from the current cell (Sestoft 2006). For instance, the reference to `B2` in Fig. 1 (1) is a relative cell reference, is represented as `C-2R0` ("two columns left, same row"). Relative cell referencing allows formulas to be moved around across the grid without having to adjust explicit column names or row indices.

Interacting with the spreadsheet not only updates the sheet itself, but also maintains the origin relation. We describe the effect of the most relevant edit operations on a cell c:

- *Entering a formula*: If c does not participate in any origin relation, it is is simply updated with the new formula, and the origin relation is updated with $\langle c, c\rangle$ to model the fact that a new formula is its own origin. As a result, the origin relation is always reflexively closed. Otherwise, c has an origin, say c', and the cells that need to be updated are $\{c'' \mid \langle c'', c'\rangle \in Org\}$. By definition, this includes cell c, and, by reflexivity of Org, the source cell c' as well. .

- *Copying cell c to c'*: The contents of c is copied to c'. If the contents is a formula, the origin relation needs to be updated as well. First, if c' has an existing origin, the corresponding pair is removed from the relation. Then the relation is extended based on the current copy operation: if c has an origin c'', add $\langle c', c''\rangle$, else add $\langle c', c\rangle$. The check for the origin of c ensures that the origin relation is always transitively closed.

- *Inserting/removing a row or column*: after updating the sheet, the origin relation is adjusted so that cell addresses refer to their new locations. For instance, when inserting a row at position i, the row components of all the cell addresses on rows $\geq i$ in the origin relation needs to be shifted one down. In the case of removal, all pairs in the origin relation that contain coordinates on the removed row or column are removed.

- *Entering data*: cell c is updated with the new data. All pairs containing c, either as source or target, are removed from the origin relation.

Note that copying a cell c to c' removes the origin entries of c' (if any). An alternative design could interpret copying a formula as a modification of the destination cell, and thus update all cells in the class of c'. In that case all such cells would get c as their new origin.

Although in this section we have just discussed copy-paste tracking for formulas, the same model can be applied equally well to copy-pasting of data. In that case, the origin relation helps against inadvertently duplicating input data. An interesting special case is the "paste as value" operation. Instead of copying a formula, this operation copies the computed value, thus completely disconnecting the destination cell from its source. Tracking such copy-paste actions would probably not be very useful: editing the pasted value would incur computing the inverse of the original formula, and updating the input data accordingly!

4. Related Work

Copy-paste tracking is a simple technique that is inspired by similar concepts in domains as diverse as term rewriting, hypertext, clone detection, prototypical inheritance , and view maintenance. Below we briefly summarize representative related work in those areas.

Origin tracking: Copy-paste tracking is directly inspired by *origin tracking* (Deursen, Klint, and Tip 1993). In general, origin tracking tries to establish a relation between the input and output of some computational process, such as a compiler, or program transformation. Origin tracking, however, has numerous other applications in visualization, debugging, and traceability. An application most similar to our work is presented in (Inostroza, Storm, and Erdweg 2014), where origin tracking is used to implement editable regions on generated code.

Transclusion: Ted Nelson's concept of *transclusion* (Nelson 1965) is a form of "reference by inclusion" where transcluded data is presented through a "live" view: whenever the transcluded content is updated, the views are updated as well. Our origin relation provides a similar hyper-linking between cells. But unlike in the case of transclusion, the relation is bidirectional: changes to the original are propagated forward, but changes to copies (references) are also propagated backwards (and then forwards again). A similar concept is used in Subtext, where copying is the primary mechanism for abstraction (Edwards 2005).

Clone tracking in software: Godfrey and Tu (Godfrey and Tu 2002) proposed a method called *origin analysis* which is a related to both clone detection and the above described origin tracking, but aims at deciding if a program entity was newly introduced or whether it if it should more accurately be viewed as a renamed, moved, or otherwise changed version of an previously existing entity. This laid the ground for a tool called *CloneTracker* that "can automatically track clones as the code evolves, notify developers of modifications to clone regions, and support simultaneous editing of clone regions." (Duala-Ekoko and Robillard 2007).

Prototype-based inheritance: Lieberman introduced prototypes to implement shared behaviour in object-oriented programming (Lieberman 1986). In prototype-based languages, objects are created by cloning and existing object. The cloned object then inherits features (methods, slots) from its prototype. The parent relation between objects is similar to our origin relation. However, we are not aware of any related work using this relation to propagate changes to clones back to their parents.

Bidirectional transformation: one way to look at copy-paste tracking is to see copies as views on the original formula similar to views in database systems. In particular, the copies are *updateable* views (Bancilhon and Spyratos 1981). Different manifestations of the view update problem have received considerable attention recently in the context of *lenses* (J. N. Foster et al. 2007) and bidirectional transformation (Czarnecki et al. 2009). In the context of user interfaces these concepts were pioneered by Meertens under the header of "constraint maintenance" (Meertens 1998). In a certain sense, copy-paste tracking supports a very basic class of constraint maintenance where clones are simply synchronized to be equal.

5. Conclusion

Spreadsheet systems are the most popular live programming environments. They adhere to the powerful direct manipulation style of simultaneously editing data and code. Nevertheless, spreadsheets are known to be extremely fault-prone, mainly because users have to use copy-paste instead of user defined abstractions. Existing research has tried to improve spreadsheets by introducing abstractions such as meta models or user defined functions, but this compromises the direct manipulation aspect that makes spreadsheets so attractive in the first place.

In this paper we propose XanaSheet: copy-paste tracking as way to both have our cake and eat it too. Instead of introducing another level of indirection, copy-paste tracking supports editing classes of formulas originating at the same source, all at once. As a result, we get the benefits of abstraction (reuse, sharing, "single-point-of-change"), without the incurring the burden of cognitive distance.

Outlook Duplication of knowledge is ubiquitous is computing. Copy-paste tracking can be generalized to a broader scope by seeing it as an example of abstractions that are presented to the user in a materialized, expanded, unrolled, referenced, or instantiated state. The relation between such views and the original is often many-to-one and the views are often read only. Copy-paste tracking could provide a model to make such user views of abstractions editable. Thus, copy-paste tracking in its most general form supports direct manipulation in interactive systems and allows users to maintain abstractions through their multiple concretizations. We conclude by providing a tentative list of examples where similar ideas could be applied:

"Copy" (many)	"Source" (one)
Reference	Declaration
Stack frame	Procedure call
Inlining	Procedure
Text output	Template
Object	Class
Styled element	Style sheet
Denormalized view	Normalized database
Unrolling	Loop

6. References

Bancilhon, François, and Nicolas Spyratos. 1981. "Update Semantics of Relational Views." *ACM Transactions on Database Systems (TODS)* 6 (4): 557–575.

Czarnecki, Krzysztof, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. 2009. "Bidirectional Transformations: a Cross-Discipline Perspective." In *Theory and Practice of Model Transformations*, 260–283. Springer.

Deursen, Arie van, Paul Klint, and Frank Tip. 1993. "Origin Tracking." *Symbolic Computation* 15 (5/6): 523–545.

Duala-Ekoko, Ekwa, and Martin P. Robillard. 2007. "Tracking Code Clones in Evolving Software." In *Proceedings of the 29th International Conference on Software Engineering*, 158–167. ICSE '07. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICSE.2007.90. http://dx.doi.org/10.1109/ICSE.2007.90.

Durden, Tyler. "Blatant Data Error at the Federal Reserve." http://www.zerohedge.com/article/blatant-data-error-federal-reserve.

Edwards, Jonathan. 2005. "Subtext: Uncovering the Simplicity of Programming." In *OOPSLA*, 505–518. ACM. doi:10.1145/1094811.1094851.

Engels, Gregor, and Martin Erwig. 2005. "ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications." In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 124–133. ACM.

Foster, J. Nathan, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem." *ACM Transactions on Programming Languages and Systems* 29 (3) (May): 17. doi:10.1145/1232420.1232424.

Godfrey, Michael, and Qiang Tu. 2002. "Tracking Structural Evolution Using Origin Analysis." In *Proceedings of the International Workshop on Principles of Software Evolution*, 117–119. IWPSE '02. New York, NY, USA: ACM. doi:10.1145/512035.512062. http://doi.acm.org/10.1145/512035.512062.

Hermans, Felienne. 2013. "Analyzing and Visualizing Spreadsheets." PhD thesis, Delft University of Technology.

Hermans, Felienne, Ben Sedee, Martin Pinzger, and Arie van Deursen. 2013. "Data Clone Detection and Visualization in Spreadsheets." In *Proceedings of ICSE '13*, 292–301.

Inostroza, Pablo, Tijs van der Storm, and Sebastian Erdweg. 2014. "Tracing Program Transformations with String Origins." In *Theory and Practice of Model Transformations*, edited by Davide Di Ruscio and Dániel Varró, 8568:154–169. LNCS. Springer.

Jones, Simon Peyton, Alan Blackwell, and Margaret Burnett. 2003. "A User-Centred Approach to Functions in Excel." In *In Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, 165–176. ICFP '03. New York, NY, USA: ACM. doi:10.1145/944705.944721. http://doi.acm.org/10.1145/944705.944721.

Lieberman, Henry. 1986. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems." In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, 214–223. OOPLSA '86. New York, NY, USA: ACM. doi:10.1145/28697.28718. http://doi.acm.org/10.1145/28697.28718.

Meertens, Lambert. 1998. "Designing Constraint Maintainers for User Interaction." http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.3250&rep=rep1&type=pdf.

Nelson, T. H. 1965. "Complex Information Processing: a File Structure for the Complex, the Changing and the Indeterminate." In *Proceedings of the 1965 20th National Conference*, 84–100. ACM '65. New York, NY, USA: ACM. doi:10.1145/800197.806036. http://doi.acm.org/10.1145/800197.806036.

Norman, Donald A. 1986. "User Centered System Design: New Perspectives on Human-Computer Interaction." In 31–61. Hillsdale, NJ: Lawrence Erlbaum.

Panko, R. 2006. "Facing the Problem of Spreadsheet Errors." *Decision Line* 37 (5).

Pryor, Louise. "TransAlta Corporation." http://www.louisepryor.com/2003/06/26/transalta/.

Sestoft, Peter. 2006. "A Spreadsheet Core Implementation in C#." TR-2006-91. IT University. http://www.itu.dk/people/sestoft/corecalc/ITU-TR-2006-91.pdf.

Winston, W.L. 2001. "Executive Education Opportunities." *OR/MS Today* 28 (4).

Live Coding / Weaving — Penelopean Mêtis and the Weaver-Coder's Kairos

Emma Cocker
Reader in Fine Art
Nottingham Trent University
emma.cocker@ntu.ac.uk

Abstract
Drawing on my experience as a critical interlocutor within the *Weaving Codes, Coding Weaves* project (2014 — 2016, http://kairotic.org/), in this paper I propose potential points of connection between Ancient weaving and live coding, considering both practices through the prism of the Ancient Greek concept of *technē*, a species of tactical knowledge combining the principles of *mêtis* (cunning intelligence) and *kairos* (opportune timing). Specifically, this enquiry addresses the human qualities of attention, cognitive agility and tactical intelligence activated within both live coding and live weaving, arguing that such practices might have potential as 'practices of the self', as a means for cultivating a more critical mode of human agency and subjectivity.

This paper elaborates ideas emerging from my experience as respondent/interlocutor within the current research project *Weaving Codes, Coding Weaves* (2014 — 2016), which is an interdisciplinary collaboration (funded by an UK Arts and Humanities Research Council Digital Transformations Amplification Award). Led by principle investigators Ellen Harlizius-Klück and Alex McLean (alongside collaborative developer Dave Griffiths and co-investigator Kia Ng), *Weaving Codes, Coding Weaves* explores the historical and theoretical points of resonance between the practice of weaving and computer programming. My own role — as interlocutor — addresses the insights gained if we bring these activities together, identifying and elaborating possible ways in which the practices of live coding and ancient weaving might connect, somehow *inter*weave. I approach the project through the prism of my own practice as a writer-artist with a background in Fine Art but without specialist knowledge or expertise related to either weaving or coding. Rather than tackling the research area from the perspective of a specific discipline (weaving or coding) and the technical preoccupations associated therein, my approach is one of attending to what is "inter". "Inter" — a contingent state of being *between*, betwixt, amongst, in the midst of. "Inter" signals towards what is produced *during*, emerging from within. The term also refers to the act of being put to ground, the condition of a body buried. Here, 'burying oneself' (myself) might describe the state of being immersed or embedded in a particular situation or site, working from the *inter*ior.

My embedded involvement in *Weaving Codes, Coding Weaves* echoes and develops the interlocutory role I performed in a previous research project (led by co-investigators Alex McLean and Hester Reeve) entitled *Live Notation: Transforming Matters of Performance*, also funded within the UK Arts and Humanities Research Council *Digital Transformations* theme, for exploring the possibilities of relating live coding (performing with programming languages) and live art (performing with actions). My response to that project was an article entitled, *Live Notation: Reflections on a Kairotic Practice* (Cocker 2013), where I elaborated an idea of 'live notation' (within both live art and live coding) as a contemporary manifestation of the Ancient Greek concept of *technē*, a species of knowledge combining the principles of *mêtis* (cunning intelligence) and *kairos* (opportune timing). Through the new research undertaken as part of the *Weaving Codes, Coding Weaves* project, I extend this enquiry in order to address how the practice of live coding and ancient loom weaving might also be considered through the prism of *technē*, with its attendant principles of wiliness and timeliness.

Still in its emergent research phase, in this paper I reflect on points of potential connection between live coding and ancient loom weaving, paying particular attention to the principles of Penelopean *mêtis* and the weaver-coder's *kairos* operating therein. It is perhaps useful to stress that the current phase of research is still in progress and as such this paper offers a conceptual proposition, which will be further 'fleshed out' through closer attention to the material technicalities of both live coding and ancient weaving as the *Weaving Codes, Coding Weaves* project evolves. Additionally, I am collaborating with Geoff Cox and Alan

Blackwell on the development of *Live Coding Alternatives*, a live coding workshop (as part of *Critical Alternatives*, 5th Dicennial Aarhus Conference, August 2015) for testing and exploring live coding as a creative, aesthetic and potentially political practice for constructing 'critical alternatives' within both computing and everyday life. I approach these contexts — as well as the *International Conference on Live Coding* (ICLC) itself — as live sites for testing and developing a proposition, for inviting further dialogue with the live coding community rather than presenting my argument as a *fait accompli.*

It is also perhaps also useful to situate this specific enquiry '*Live Coding/Weaving — Penelopean Mêtis and the Weaver-Coder's Kairos'* against the context of a broader investigation within my research activity, focused on identifying contemporary manifestations of *technē* (especially within artistic practice — and with an interest in *drawing* in particular), and in advocating how the particular qualities, capacities and knowledges produced therein might have a critical, perhaps even resistant potential (Cocker 2010). Within this ongoing investigation, I address how a more augmented, affective expression of human agency and subjectivity might be cultivated through the activation of *technē*, through attending to and nurturing the principles of *mêtis* (cunning intelligence) and *kairos* (opportune timing) within both creative practice and daily life. Here, *technē* is not used in its habitual sense, where it is taken to simply mean the skillful art of *making* and *doing*, the technical facility of craftsmanship. Whilst retaining the idea of *technē* as a means of making/doing, my intent is to explore how Ancient Greek weaving and philosophy might intertwine with live coding on both technical *and* conceptual levels. Making a return to how the term was used within Ancient Greek culture, I refer to *technē* as a disruptive — even subversive — species of tactical knowledge, capable of dealing with contingent situations and fully harnessing their capricious force (for example, a knowledge capable of seeing and seizing the potential of chance, randomness and indeterminacy and converting this towards unexpected direction). As Janet Atwill argues "Because *technē* defined itself in terms of intervention and invention, it is concerned solely with situations that yield indeterminacies that would allow one to discern the opportune moment and to 'seize the advantage'" (Atwill 1998, 70 — 71). This intimates towards the rather unexpected aspect of *technē*, for it is not a practice intent on stabilizing or limiting the contingency of unstable or indeterminate forces (nor resisting their pressures), but rather on the transformation of their force towards opportunity. For Atwill *technē* "challenges those forces and limits with its power to discover (*heuriskein*) and invent new paths (*poroi*)"; it "deforms limits into new paths in order to reach — or better yet, to produce — an alternative destination" (Atwill 1998: 69).

My assertion is that there are certain contemporary, creative processes (for example, live coding, as well as forms drawing, performance and indeed weaving) that display the properties of *technē*, that require a specific quality of alertness or attentiveness to the live circumstances or 'occasionality' of their own production, moreover, that these qualities (identified and *practiced*) have the capacity to be applied to other situations, indeed to the living of a life. In this sense, my own research investigation might be conceived in relation to Michel Foucault's late work on Ancient Greek ethics and the 'care of the self', his analysis of the various 'techniques of life', 'arts of existence' or 'practices of the self' through which it might be possible to 'make life a work of art', a material capable of being made malleable, transformed *through* practice (Foucault 1997,1992/1984, 2001). For the Ancient Greeks, the training and cultivation of *technē* was central to the construction of both self and civic life, the capacity of invention, intervention and conceptualizing of critical alternatives. Indeed, the Greeks developed a complex interlocking system of preparatory training and reflexive exercises (or *askēsis*) for testing mind and body, the convergence of athletics and rhetoric for exercising language as well as limbs (Hawhee 2004). My current enquiry asks — what are the human capabilities and capacities (related to *mêtis* and *kairos*) cultivated through the practice of live weaving and live coding; indeed, how might weaving and coding be *practiced* as *askēsis*, as potential practices of the self?

The connections between live coding and ancient weaving that I seek to excavate are less directly to do with the shared technology (the relation between the computer and the loom), nor shared notational systems (pattern and code) or mathematical algorithms, nor even the relationship or resonance between the resulting weaves (whether digital or textile). My focus is more on the capacities, knowledges and qualities of attention activated "inter", emerging *in between* the disciplinary lines, moreover, through the relation or negotiation between human and machine; through the live and embodied process of decision-making involved in both weaving and coding which conventional notational systems seem arguably unable to fully account for. What 'lost' or buried connections between coding and weaving emerge through disrupting or dislodging the position of the Jacquard loom in the historical development and conceptualization of these

practices? What knowledges or capacities become lost or devalued through the privileging of speed, productivity, economic efficiency and standardization that technological developments such as the Jacquard loom facilitate? How might these knowledges and capacities be recuperated or retrieved? Indeed, certain technologies actively create the conditions of ignorance or alienation, where a technology has the capacity to be used or operated in the absence of any knowledge of underpinning process, principles or structure. In doing so perhaps, what becomes cultivated is a reliance on templates, on a received standard or model developed for a specific (predetermined) purpose or function. The possibility of deviation from the norm, for bespoke options, for modification or adaptation becomes problematized, increasingly less of an option. In time, the capacity to imagine other ways of doing things might dissolve or dissipate; possibilities conform to the standard fit, the path of least resistance. Here perhaps, it is possible to argue how the acceptance of standards or templates alongside ignorance of underpinning structures and causes within one context, facilitates the same in other aspects of lived experience. Or else perhaps, more affirmatively, *can* the questioning of standards or templates alongside increasing awareness of underpinning structures and causes within one context facilitate the same in other aspects of life? The revelation and live reworking of digital code through the performance of live coding involves showing and sharing the unfolding logic of a language so instrumental to contemporary life, but in which few are fluent. Learn the language of the operator (the programmer or code-writer) or be forever subject to another's code. "Program or be programmed", (Rushkoff 2010); code or be coded; write or be written; weave or be woven.

The live coding and ancient weaving practices that I have encountered within the *Weaving Codes, Coding Weaves* project invite a more complex, nuanced or even entangled human/machine relation, where technology is not so much put to use as *worked with*, the process unfolding through attending to — even collaborating with — the resistance exerted by the technology or apparatus rather than conceiving it simply as a tool that needs to be brought under control, *mastered*. Creating the right tension — a process of improvisatory working emerges through cultivating an understanding of tolerance, how far something can be pushed or pressured before it breaks, indeed, when to instill breaks or rests. Gilles Deleuze names the power to affect other forces — *spontaneity*, and to be affected by others — *receptivity* (Deleuze 1999: 60). Somewhere between spontaneity and receptivity, somewhere between control and letting go, somewhere between affecting and being affected. Rather than giving over responsibility to the inevitability of a rule's logic, within live coding and live weaving practices the coder-weaver consciously adopts a medial position, actively maintaining the conditions that will keep the unfolding of action dynamic. Debra Hawhee conceptualizes the medial position of "invention-in-the-middle" as a *kairotic* movement involving "simultaneous extending outwards and folding back"; it is a "space-time that marks the emergence of a provisional 'subject', one that works on and is worked on by — the situation" (Hawhee 2002: 18). *Medial.* I think of the helmsman steering the boat, navigating the competing pressures and forces of the water and the wind, or else the *artist-pencil* drawing. Loom like boat like laptop — each an extension of human capacity, embodied prosthesis. Within these various practices (weaving, coding, drawing, indeed sailing), where does the capacity of body end and prosthesis/apparatus begin? Both live coding and ancient weaving foreground understanding of process and structure, refusing or even undoing the logic of a given, accepted model or concept in order for it to be reworked or modified. Yet, the mode for understanding process is not based wholly on abstraction as such or taken as a given, rather cultivated through use and experiment, through trial and error, by doing something as a way of knowing how something is done, moreover, for knowing how it might be changed, *swerved,* taken in a different direction. Or rather, understanding is cultivated through the oscillation or even 'shuttling' between 'discontinuous' systems of abstract notation and the 'continuous' experience of a lived process. What are the cognitive and bodily intelligences operating in this space between the discontinuous and continuous, between the abstract and the lived?

For the Ancient Greeks, the term *mêtis* described a form of *wily* intelligence capable of seizing the opportunities *(kairos)* made momentarily visible as the prevailing logic within a given structure or system yields. *Mêtis* is the art of preparing for what could not have been anticipated or planned for in advance; it the same skill used in catching the force of the wind or the turn of the tide, which as Marcel Detienne and Jean-Pierre Vernant note is "as prompt as the opportunity that it must seize on the wing, not allowing it to pass" (1991/1978: 15). Harnessing the properties of dexterity, sureness of eye and sharp-wittedness, *mêtis* "attempts to reach its desired goal by feeling its way and guessing"; it is a "type of cognition which is alien to truth and quite separate from *episteme,* knowledge" (Detienne and Vernant 1991/1978: 4). Reflecting on its role within Ancient Greek rhetoric, Detienne and Vernant describe *mêtis* as, "a type of intelligence and of

thought, a way of knowing; it implies a complex but very coherent body of mental attitudes and intellectual behaviour which combine flair, wisdom, forethought, subtlety of mind, deception, resourcefulness, vigilance, opportunism [...] It is applied to situations which are transient, shifting, disconcerting, and ambiguous, situations which do not lend themselves to precise measurement, exact calculation or rigorous logic" (Detienne and Vernant 1991/1978: 3 — 4). Knowledge through experiment might take the form of *doing and undoing*, the repeated labour of trying something out, over and over, again and again; tacit knowledge cultivated through the accumulation of tests and attempts. Here, repetition might be undertaken less towards the perfection of a given move or manoeuvre but rather towards the building of confidence and capacity. A working knowledge of a process or material such that it becomes ingrained in mind and muscle, activated at the fingertips and in the process of thinking *en acte*, live and emergent to the situation rather than pre-planned in advance. In these terms then, it could be argued that the nature of the knowledge is less of a tacit kind or of a 'know how', but rather a form of *knowing* closer to the "immanent *intensification*" of thinking which philosopher Alain Badiou (following Nietzsche) asserts "is not effectuated anywhere else than where it is given — thought is effective in situ, it is what ... is intensified upon itself, or again, it is the movement of its own intensity" (Badiou, 2005 58 — 9). A tactical knowledge performed through receptivity to the possibility of what remains unknown, open to the unfolding situation. Not a knowledge that is easily banked and transferable, but rather acquired *through* practice, moreover, activated only in-and-through practice.

To improvise within a given structure requires skillfulness and attention, a capacity for biding one's time and knowing when and how to act. Here then, rules or instructions are not to be diligently followed but rather have the capacity to be modified or adapted even while they are being executed, the tension of an unfolding thread of weave or code varied as it is being woven/written, or else undone and rewoven, enabling the possibility of a change of tack. For live coders including the collective *slub* (Dave Griffiths, Alex McLean and Adrian Ward) "the preferred option for live coding is that of interpreted scripting languages, giving an immediate code and run aesthetic" (Collins *et al.* 2003: 321). Here, audiences encounter projected code as a running command line whilst it is being modified and rewritten by the programmer and in some specific instances *by* the code or interpreting system itself. Cox notes, within certain live coding practices the, "running code is also able to edit its own source code ... That these self-modifications happen directly to the code being edited in real time puts the code visibly on the same level as the programmer" (Cox 2013: 61). The live running code makes manifest a developing logic based on the principles of *what if*, through the testing of the possibility of *this* or *this* or *this* or *this*. What then, is the specificity of thinking-in-action activated whilst improvising within a live running code, and how might it relate to *working on* the loom? Indeed, the *means* through which the same result is arrived can create different kinds of knowledge, different human capacities.

Rather than through the unfolding of trial and error — the generative testing of *this* or *this* or *this* — knowledge might be acquired through the inversion of a given process. Here, undoing comes *first*: reverse engineering of a weave or code necessary for seeing the underpinning structure, not only the surface pattern. This capacity for differentiating the pattern from the structure has an implicit political imperative or application, cultivating the ability to discern an essential sameness within certain options offered, for recognizing that certain choices are effectively cut from the same cloth. An active undoing then for knowing how something works or is structured, for seeing beyond the surface of things. Acts of appropriation, hacking and backtracking as a means of taking back control, no rather for *resisting* control, for reasserting the potential for creative improvisation within a seemingly standardized process, for recuperating human agency within systems whose options seem increasingly closed, prohibitive. A form of "creative consumption" (Michel de Certeau 1984) or the cultivation of a "minor language or practice" (Deleuze and Guattari 1986) wherein the prescribed 'codes' and patterns of the dominant culture are appropriated (hacked), modified or inverted (creatively reverse-engineered), and then redirected towards other (often less utilitarian) ends. Here then, existing patterns, rules and codes are not to be taken as given (as fixed or unchangeable) but rather appropriated as a found material with which to work, rework. The process of coding or weaving in these terms is not conceived as an algorithmic operation whose logic is simply imported, set in motion and allowed to run its course; rather both have the capacity to be unraveled and rewritten as events unfold. Not so much the Beckettian "fail again, fail better" model (Beckett 1999: 7) — a doing and undoing for reflecting a relation between utility and futility — but rather an affirmative *and* resistant practice.

Doing and undoing, undoing and re-doing: performed as a mode of deviation or subversion, of purposefully non-productive labour. I am reminded of Penelope, wily weaver of Ancient myth, wife of Odysseus in Homer's *Odyssey* — weaving by day and unweaving by night, willfully unraveling the weave such that by morning the task might begin afresh. Hers is the act of unweaving and reweaving to avoid the completion of a task, for refusing the teleology of outcome or commodity, the production of a product and its consequences. This might be seen in the writing of certain live coders, who rather than working towards some teleological endpoint, might even attempt to begin and end with nothing. For Penelope — the stakes of whose weaving were indeed quite high — the practice of unweaving and reweaving was performed as an act of quiet resistance, so as to thwart the terms of a situation from which there would seem to be no way out. For the contemporary live coder, might not the Penelopean logic of doing and undoing, or even undoing and then redoing, also be harnessed as an act of 'minor resistance', conceived as an attempt to thwart or subvert capture by capital, refusing the terms of easy assimilation. Here, code is woven only to be unraveled, or else the emphasis is process-oriented, code remaining only as a trace or residue of live action, for making tangible the process of decision-making.

Whilst the remains of code from a live performance enables the capacity for something to be repeated again or reworked, it would appear that both live coding / live weaving are somehow less about approaching the situation with 'a code prepared in advance', but rather have to be practiced *as a practice.* Weaving and unweaving of both code and thread as an open-ended process, not so much concerned with the production of product as *experience.* Yet, what *other meanings* and capacities — if not products — might be produced therein? I think of Luce Irigaray when she says, "one must listen differently in order to hear an other meaning which is constantly in the process of weaving itself, at the same time ceaselessly embracing words and yet casting them off to avoid becoming fixed immobilised" (Irigaray 1980: 103). What kinds of performativity, what politics, what philosophies, what poetics emerge therein? For Barbara Clayton, the Penelopean poetics of weaving and unweaving are generative, where "undoing does not signify loss or nullity, but rather life affirming renewal and the constant possibility of new beginnings" (Clayton 2004: 124) Not the repetitive practice of sameness then, but rather one of attending to difference, to the potential twists, variations and permutations of the thread or code. Here, a 'doing-undoing-redoing' perhaps akin to the Deleuzian conceptualization of a *plier/déplier/replier,* where the act of folding, unfolding and refolding "no longer simply means tension-release, contraction-dilation, but enveloping-developing, involution-evolution" (Deleuze 2006: 9). Or else the practice of undoing might be undertaken in order to create the conditions for doing over and again, repetition as a means of embodied and affective knowledge production, the knowledge that comes from a process *practiced,* cultivation of the art of knowing *when* as much as how.

Repetition of a process as training or as exercise, *askēsis* even? *Askēsis:* preparatory training and reflexive exercises connected to the cultivation of *technē.* Could live coding and indeed live weaving be conceived as *askēsis* for practicing or testing the art of timing or of timeliness, the capacity for responding to the new situation as it unfolds, attending to the gaps and deciding how to act. In one sense, both live coding and weaving can be considered *kairotic* practices based on the principles of intervention and invention-in-the-middle. *Kairos* is an Ancient Greek term meaning an opportune or fleeting moment whose potential must be grasped before it passes. It describes a qualitatively different mode of time to that of linear or chronological time (*chronos).* It is not an abstract measure of time passing but of time ready to be seized, an expression of timeliness, a critical juncture or 'right time' where something *could* happen. *Kairos* has origins in two different sources: *archery*, where as Eric Charles White notes, it describes "an opening or 'opportunity' or, more precisely, a long tunnel like aperture through which the archer's arrow has to pass", and *weaving* where there is "a 'critical time' when the weaver must draw the yarn through a gap that momentarily opens in the warp of the cloth being woven" (White 1987: 13). In *kairotic* terms, the importance of weaving is reiterated — as both a practice and also as a metaphor for articulating a particular quality of thinking-in-action (perhaps even 'loom thinking') (Jefferies 2001). A seizing of opportunity based on cunning intelligence and propitious timing, on the *kairotic* art of knowing when as much as where and how.

Yet, the opportunity of *kairos* has little power on its own; it requires the perceptions and actions of an individual capable of seizing its potential. As Debra Hawhee states, "*kairos* entails the twin abilities to notice and respond with both mind and body … the capacity for discerning *kairos* … depends on a ready, perceptive body" (Hawhee 2004: 71). For White, *kairos* involves a "will-to-invent" or form of improvisation,

that necessitates "adaption to an always mutating situation. Understood as a principle of invention … *kairos* counsels thought to act always, as it were, on the spur of the moment" (White 1987: 13) or perhaps in live coding terms through a process of coding "on the fly". As Nick Collins *et al* note, live coders "work with programming languages, building their own custom software, tweaking or writing the programs themselves as they perform" (Collins *et al.* 2003:1). Code is written as it is performed; a practice often referred to as 'coding on the fly' or 'just-in-time coding' (or what I would propositionally name '*kairotic* coding'). White states, "*Kairos* stands for a radical principle of occasionality which implies a conception of the production of meaning in language as a process of continuous adjustment to and creation of the present occasion" (White 1987:14 — 15). Based on the twin principles of *mêtis* and *kairos*, Ancient *technē* affords an alternative epistemological framework for addressing key questions of live-ness and dexterity critical to live coding as a practice.

A practice based on timing and timeliness, emerging between the principle of biding one's time and knowing when to act. The weaver-coder navigates a course of action by intuiting when to yield to the rule or code or even the technology itself and when to reassert control, by knowing when to respond and when to interrupt. Yet within the logic of *technē,* opportunities are produced rather than awaited. It is a practice or art deemed capable of setting up the conditions wherein *kairos* (the time of opportunity) might arise and in knowing (through a form of *mêtis* or intuitive intelligence) how and when to act in response. *Kairos* involves the making of the situation at the same time as deciding how to act. A gap is made in the weave at the same time as deciding how (and when) to shuttle the thread through. A language is generated simultaneous to its use. A live and present practice: the live toggling back and forth of the cursor and the shuttle, decisions made from inside the weave, from within the continuity of a process (a running code) rather than applied by as a process of design from without. A *kairotic* practice is not one of planning in advance or designing from a distance. There is no concealment of preparation, no cuts to be made after the fact — all is visible, all part of the work. The back and the front of a process are indivisible; moreover, the polar logic of self and technology, discontinuous and continuous process, hidden and visible becomes blurred. Preparation becomes folded into the practice itself, is part of (and not prior to) the process. Moreover, each action has the capacity to create the conditions or scaffolding for the next; what is at first the weft will later become the warp. Perhaps it is in this sense that such practices might be considered *askēsis* or training exercises*,* the principles cultivated therein — of one action setting up the conditions for the next — expanded as a 'practice of the self', life as a work of art practiced as the warp and weft of an unfolding code/weave. In this sense, my paper addresses the relation between ancient weaving and live coding through the prism of *technē*, a form of practical knowledge combining principles of opportune timing (the weaver-coder's *kairos*) and cunning intelligence (a specifically *Penelopean metis),* in order to ask more speculatively, 'how might live coding and live weaving be imagined as pragmatic technologies of the self and society for twenty-first century life'?

REFERENCES

Atwill, Janet. 1998. *Rhetoric Reclaimed: Aristotle and the Liberal Arts Tradition.* Cornell University Press, Ithaca and London.

Badiou, Alain. 2005. 'Dance as a Metaphor for Thought', *Handbook of Inaesthetics,* (trans.) Alberto Toscano. Stanford University Press.

Beckett, Samuel. 1999. *Worstward Ho.* John Calder, London.

Clayton, Barbara. 2004. *A Penelopean Poetics — Reweaving the Feminine in Homer's Odyssey.* Lexington Books.

Cocker, Emma. 2013. 'Live Notation: Reflections on a Kairotic Practice', *Performance Research Journal,* Volume 18, Issue 5, On Writing and Digital Media, pp. 69 — 76.

Cocker, Emma. 2010. 'The Restless Line, Drawing', *Hyperdrawing: Beyond the Lines of Contemporary Art*, eds. Russell Marshall and Phil Sawdon. I. B. Tauris.

Cox, Geoff and McLean, Alex. 2012. *Speaking Code: Coding as Aesthetic and Political Expression.* MIT Press.

Collins, Nick, McLean, Alex, Rohrhuber, Julian and Ward, Adrian. 2003. 'Live Coding Techniques for Laptop Performance', *Organised Sound* vol. 8 (3), pp. 321 — 30.

de Certeau, Michel. 1984. *The Practice of Everyday Life*, (trans). Steven Rendall. Berkeley and Los Angeles: University of California Press.

Deleuze, Gilles. 1999. 'Foldings, or the inside of thought (subjectivation)', *Foucault,* trans. Séan Hand. Continuum.
Deleuze, Gilles and Guattari, Félix. 1986. *Kafka: Toward a Minor Literature*. University of Minnesota Press, Minneapolis.
Deleuze, Gilles. 2006. *The Fold*. Continuum, London and New York.
Detienne, Marcel, and Jean-Pierre Vernant. 1991/1978. *Cunning Intelligence in Greek Culture and Society*, (trans). Janet Lloyd. Chicago: University of Chicago Press.
Foucault, Michel. 1992/1984. *The Use of Pleasure. The History of Sexuality: Volume Two.* (trans.) R. Hurley. Harmondsworth, Middlesex.
Foucault, Michel. 2001. *The Hermeneutics of the Subject, Lectures at the College de France 1981 — 1982*. Picador, New York.
Hawhee, Debra. 2004. *Bodily Arts: Rhetoric and Athletics in Ancient Greece*. University of Texas Press, Austin.
Hawhee, Debra. 2002. 'Kairotic Encounters', *Perspectives on Rhetorical Invention*, (eds.) Janet Atwill and Janice M. Lauer. University of Tennessee Press.
Irigaray, Luce. 1980. 'This Sex Which is Not One', *New French Feminisms*, eds. Elaine Marks and Isabelle de Courtivron. University of Massachusetts Press, Massachusetts. Originally published as *Ce sexe qui n'en est pas un* (Minuit, 1977)
Jefferies, Janis. 2001. 'Textiles: What can she Know?', *Feminist and Visual Culture*, eds. Fiona Carson and Claire Pajaczkowska. Edinburgh University Press, pp. 189 — 207.
Rushkoff, Douglas, 2010. *Program or Be Programmed: Ten Commands for a Digital Age,*
White, Eric Charles. 1987. *Kaironomia: On the Will to Invent*. Cornell University Press, Ithaca and London.

ANALYSING LIVE CODING WITH ETHNOGRAPHICAL APPROACH. A NEW PERSPECTIVE.

Giovanni Mori
University of Florence
giovanni.mori@unifi.it

ABSTRACT

In this article, I will analyse live coding technique under the magnifying lens of Ethnography. Using this perspective, I will try to delve into three main aspects: the effect on the audience/performer interaction of the screen projection during performances; the relationship between "hacker's ethic", borrowing a Pekka Himanen's definition, and live coders community; how some researchers are trying to establish contacts between formal and informal music milieu. In my view, an Ethnographical approach can help people interested in live coding to contextualise some implication of this technique's internal dynamics. Additionally, this perspective can be useful to build a bridge between some academic and non-academic computer music contexts, using live coding as glue between them.

1. INTRODUCTION

A new tendency in Ethnomusicology is to include, among interesting experiences to document and analyse, not only the so-called traditional or folk music but also emergent contemporary music practices. Live coding is, without doubts, among these last ones. Many researchers think that is useful to deepen the knowledge of contemporary cultural phenomena to increase the understanding of present social processes and dynamics since their inception. The debate on this is still ongoing, though.

In the case of live coding, I think that an Ethnographic recognition made by the newest techniques can be very fruitful, because it appears to be a cross-practice music experience involving many different cultural contexts. Either the artist/audience interaction or the effect of screen displaying or, finally, the cultural milieu that lay underneath, may be interesting for understanding many social dynamics.

Then, in this article, I will explain the ongoing research I am carrying out for my PhD, trying to summarise the main concepts emerged during my inquiry both through "traditional" ethnological field documentation, through reading written documents and through the newest technique called "netnography" that I will explain better beneath.

2. LIVE CODING: A BRIEF DESCRIPTION

Live coding is an improvisatory artistic technique. It can be employed in many different performative contexts: dance, music, moving images and even weaving. I have concentrated my attention on the music side, which seems to be the most prominent. For playing with this technique, the performer have to use a dedicated software platform and a programming language to send, in real time, the music instructions to the machine. There are many live coding music programs and interfaces, but, with few exceptions, all of them are characterised by a central role of the text for managing the sound result. In fact, the computer software, for reproducing sounds, interprets the text typed in the program's window. The performer express his virtuosity through high speed typing of complex instructions with the right grammar. Sometimes the musician can type program lines to see what happens, without having then a full control on the process. Therefore, there is a will of producing an indeterminate music results on a certain degree. This happens probably because we assist to an exploratory phase and the performer is aware that he/she cannot control the whole machine processes and in this way, it is possible to find some interesting but unpredictable music solutions.

There are also some other kinds of musical interfaces, as wearables or external devices, but all produce an alphanumerical result on the screen. Thus, text seems to be the main musical referent of this technique.

The musical material employed during events can be retrieved either from the performer's computer archive or it can be created from scratch, by using various techniques as, for example, additive or subtractive synthesis, filtering and so on. Computer is not utilised as a "traditional" musical instrument, with a cause and effect connection between the performer actions and the sound heard. Musicians put their musical thoughts in a written form during programming and then the computer software translates the code in sounds when every line is launched.

Live coding in music is practically a pure "instrumental" practice. With this term, I would point out the fact that in all the performances I have attended to and I have read about, there were not singers[1]. Nevertheless, live coded shows employ human voice samples, treating them in the same way of other musical material. In some cases, the human body carries out an active role during live coding performances. For example, there can be dancers, performers of different types of instruments or wearable electronic devices that track movements and other body features to transform them in data inputs.

Live coding concerts can take different forms. A constant, however, is that every performer shows, normally through a video projector, his or her computer screen to the public. This action bring the attendees inside the music making and in every single moment, they are aware of what is happening on the musician's computer. They do this for a matter of transparency: live coders do not wish to hide their musical actions because they feel themselves to be performers and they want that the public perceive clearly this. They do not want to be seen as someone simply standing behind a screen leaving the audience unaware of what he or she is doing. During many computer music performances, the artists may be suspected of checking emails or chatting with someone instead of playing, while the computer reproduces the music heard by the audience. Live coders think that with screen projection it is clear that they are playing and not doing something else. This act of transparency has probably something in common with conceptions coming from the world of popular music, where, during the last decades, authenticity, sincerity and transparency have become highly valued features (Auslander2008). The commitment in the art and the effort to obtain an "authentic" product and an "authentic" performance is typical, in particular, of underground popular music movements, less interested than mayor labels, at least in principles, in increasing their revenues exponentially. However, the need of authenticity from the listeners is becoming more and more urgent for pop stars too. See for example the Ashlee Simpson's lip-syncing scandal on Saturday Night Live, happened a few years ago, ended in some online petitions for convincing her to quit her career.

Quite underground as well, almost since the recent explosion of Android devices, is the so called "hacker's culture", into which I believe that live coding can be included. I am using the term "hacker" to address the meaning given to this word by Pekka Himanen in his book *The Hacker Ethics and the Spirit of Information Age* (Himanen2003). Hacker's community is multifaceted but everyone share some basic principles. One of the most important is that a good society has to be free and based on communities of pairs. A community works better and faster than a single alone in developing complex structures. The most famous example of this approach is Linux, the free Operative System *par exellence*. We should not interpret here the word "free" in economic terms. It is not a matter of cost but instead one of freedom. The code, in the hacker's view has to remain accessible; everyone should have the choice of accessing and modifying the code freely (Stallman2010). It seems that this belief is valid among live coders as well. Hence, there are also hackers who accept the employment of this working approach even in ICT companies for capitalistic purposes (Levy1984)[2]. As just hinted, live coders seem to adhere more to the hacker's movement branch, where it is important to share knowledge on equal basis, without requesting economic return for this, to give out intellectual products and let whoever wants to use and modify them freely. Another important aspect for hackers, as well as for live coders, is, finally, to build a community of passionate users devoted to the cause, whose feedbacks and suggestions contribute greatly to the developing of software. All the previous aspects are present in the live coding community. In fact, music programs are released under open source licence and the most famous program, Supercollider, is developed in a way similar to that of Linux. Moreover, there

[1] There are some rare cases of live coding coupled with human voice sung in real time, as for example Miko Rex or Silicon Bake.

[2] See also this interesting essay on hacker's culture based on Levy1984 but also on some other books on this subject (in Italian): http://www.altrodiritto.unifi.it/ricerche/devianza/tavassi/cap1.htm

are some online forums, as toplap.org or livecoderesearchnetwork.com, where the community exchanges opinions and informs all the people interested in live coding on the group's activities.

On the live concert hand, it seems that there is not a live coding ideal context. In fact, live coding shows can take place almost everywhere, from nightclubs to concert halls. This happens because live coding is a performative technique and not a proper music practice. Then, musicians can produce many different music forms, timbres and dynamics characteristic of diverse kinds of musical manifestation by employing it.

There are probably some connections between the software's structure and the music produced with that. For example, Threnoscope program is more suitable to produce long and droning sounds and Tidal, Gibber, Sonic Pi, etc., instead, are more likely to be used in a collective event called algorave or, generally, in music contexts that request a bodily reception. However, it is possible to produce drones or long sound with Tidal or Gibber, but their constraints do not help in doing this. Threnoscope has a double sided interface: one circular that represent an ideal sonic round space where the sound is spatialised via multichannel sound system, and one based on text where the performer types the program lines that represent the instructions for sound reproduction. Considering that this program needs a complex sound system for expressing its complete creative power, it is more likely to be used in academic contexts, where music with similar needs is often staged. The Tidal or Gibber case is different, because their authors designed those simples, flexibles and able to manage sounds very quickly. Their interfaces has a unique window where performers write the program lines. The light interface and the intuitive syntax enable them to implement parameters with ease. In fact, these programs are utilised to reproduce more or less complex rhythmic patterns and melodies, and to modify both in an extremely rapid way. In the most part of performances I have attended to, the sound result was under many aspects similar to those of dance and techno music: stratified structures, repetitive beat patterns, high level of sound reproduction and so on. Thanks to these similarities, the programs are widely used in events called, as already mentioned, algorave. A rave is a music party based on concepts as that of freedom, equality, anticapitalism and, in general, extremely libertarian principles (Hakim2007). Such principles resonate also in the live coding community, even though with less extreme connotations and outcomes. In this article, unfortunately, there is no room to think about the relationship between live coding and rave culture extensively.

The most important software in live coding field, however, appears to be Supercollider, which has the most wide music possibilities. In fact, many different programs borrow many features from that as, for example, programming language, architecture and design. This program is divided in two parts: one visible to the musician (SClang) and one invisible (SCserver). This last one is the machine that interprets in real time the instructions coming from the first one (the proper interface) and transforms them in sound. SCserver can be managed by using some other custom designed interfaces. In fact, many live coders use this piece of software through another program, which interact and exploit different SCserver features.

3. SOME REFLECTIONS ON LIVE CODING MUSIC PRACTICE

That was a brief, though not complete, description of some of the most important aspects of live coding performances. Probably now is time to reflect about implications of musical and performative actions taken inside this music community. However, before to begin the proper analysis, I would briefly introduce Ethnography, the inquiry method employed here.

Ethnography is characterised by three main rules: 1) direct observation and description of the social practices, 2) the research perspective is that of the researcher, that is partial, 3) this perspective is "full of theory" or, in other words, the researcher has a cultural baggage that influences his or her inquiry results.

Many techniques can be employed during an Ethnographic research. The most important in my research are the participative observation and the so-called netnography.

The first one entail a direct researcher participation in the cultural activities of a precise social group, for a sufficiently long period, going directly "on site". Ethnographer should establish a direct interaction with group's members to be able to understand and describe the purpose of people's actions in that particular context. In that way, knowledge is built through a process of identification by the researcher with the group's members (Corbetta1999). All this techniques are employed to obtain, firstly, an interpretation and description of a society *from the inside*, using the same words and the same concepts of "natives". Then, it

enable the “natives” to be conscious of their “tacit knowledge”: the embodied knowledge that people is not aware to own because they feel it as something natural. Finally, the researcher’s role should be emphasised because what he or she will tell depends on how he or she has conducted the field research.

The second technique to describe, Netnography, arise after spreading of Web 2.0. This term’s father is the sociologist and marketing researcher Robert Kozinets (Kozinets2009), who coined it to define a qualitative research[3] conducted on the “online” field. This technique enable the researcher to collect important behaviour information from the spontaneous chat of online users. The original aim of Netnography was to understand the consumers’ mood about commercial products, brands, prices and so on. With these insights, the marketing agency can construct personalised advertisement campaigns but sociologist too can extract important information on social interactions. Netnography has developed a scientific methodology to select useful data from the mass of online messages exchanged inside webtribes. A webtribe is simply a group of people, active on the social media, which speak about specific topics and creating a community around it. Netnographers employ non-intrusive and natural survey techniques, to limit the researcher’s interference with the normal field dynamics, letting the people to exchange opinions and evaluate products freely. Only in this way, researchers can have a faithful representation of people thoughts. These thoughts, in Netnography, undergone an interpretative analysis to extract from them both commercial and cultural insights. Commercial insights represent the people’s sentiment about a particular topic, how members employ a product and, finally, suggestions and complaints about something. Cultural insides include instead the study of “tribal culture” and “tribal ethics”: the first is how webtribe members represent themselves and how they see the world around; the second address the shared values among community members.

Although this technique has been developed for market research purpose, in my opinion it can be useful also to understand the live coding community members’ online activities because live coders as well is a webtribe grown around a “product” (the technique of live coding) that exchange opinions, evaluate, struggle to modify “the market” to match their needing.

Coming back to performance analysis, the most innovative aspect introduced by live coding, in my opinion, is the computer screen projection during concerts. This act has a clear scope: let the audience become aware that they are witnessing a live performance and not a fiction or an acousmatic concert. Some funny actions undertaken by performers, as asking through the screen questions to the audience, contribute crucially to construct this liveness effect. For example, Alex McLean has asked for a beer during his performance at the Lambda Sonic in Ghent, Belgium, typing his request on the computer screen and, consequently, on the projected image of his screen on the club’s wall. After a little while, the beer arrived brought by an audience member[4]. This weird request express the performer need not only for a beer during his show, but also, most importantly, for audience involvement on a different and unusual level. This appears to be a clear experiment of breaking the invisible wall between performer and listener during concerts. However, the main aim of screen projection is to show to those present the development of performer’s music thoughts in form of written algorithms and sound. Additionally, this entails that the musicians exposes themselves to the risks of every live performance. In most computer music concerts, the audience cannot be sure about the event's nature they are witnessing to, because calculator is not an acoustic music instrument, but acousmatic and potentially automatic. By “acousmatic”, I mean a context in which the sound source is impossible to recognise by direct visualisation (Schaeffer1966). In the computer music case, the audio output is the product of electric currents’ oscillation managed by software. Then the actual sound source is invisible and, especially in the computer case, the sound may be automatically driven.

A computer music concert may be compared, in some cases, to a sort of public records listening, with a marginal power of the musician on the sound output. In some other cases instead, the musician has a more important role, but what he or she is doing, is not evident and so it is barely distinguishable from the previous one on the visual aspect. What makes the difference between live coding and other computer music performances is that every live coder gives a proof of his or her crucial role as “event conductor”. The screen is the place where sounds take shape. Showing the ongoing music process is something comparable to play an acoustic instrument: the music making process becomes transparent and people in front of the

[3] A qualitative research use a definite group of research techniques for collecting precise information about a specific and limited case of study. A quantitative research, on the contrary, aims at collecting big amount of data on a wide research field and analyse them though the tool of statistics and mathematical models (Ronzon2008).

[4] See the performance’s video here: https://www.youtube.com/watch?v=DmTuQcTTORw

performer do not feel betrayed or cheated, but involved into the sound production. Performances can take place also via the internet, but this does not seem to influence the audience's liveness perception. It can be inferred that the public perceive the screen as the performer's real time representation, the performer's simulacrum. An example of this happened in Birmingham at the Network Music Festival. It was a telematic performance between three musicians. They interacted with each other on an online server and, through an Internet connection, they broadcasted the sound result in the concert hall. The audience, in attending at this performance, laughed at jokes, stayed silent during music "playing" and finally applauded as if they were in presence of the performer flesh and blood. This example confirms that in live coding concerts the attendees' centre of attention is the screen.

Therefore, it is evident that the performer has a crucial and active role in live coding performances. Nevertheless, as hinted above, it seems that musician's body tends to become less and less important, at least where the only sound production device is the computer. The listeners' attention is more focused on the screen, or better, on the screen's projection. This light beam appears to be the real music referent in this context and in fact, it has often the most prominent position in the performative space. People are attracted more from flashing projected images coming from computer than from the musicians themselves, because the screen conveys all the musical information that the performer cannot provide. This is also confirmed by the audience reaction when the concert is telematic. Apparently, the musician absence does not damage liveness perception. The interaction between performers on the server's chat window and with the audience confirm and reinforce this sensation.

All these statements seems to remain true even though the performative space changes and the music practice switches from a more rhythmic and informal context to a more static, reflexive and formal one. In fact, in performances like those made, for example, through Threnoscope or those taken by BEER ensemble, where audience members are not stimulated to move all their bodies following the music rhythm, the screen polarise the listener's attention probably on a higher degree. For example, during the performance held at Escola de Música do Conservatório Nacional in Lisbon by Thor Magnusson and Miguel Mira, my attention was attracted by actions represented on the screen projection and by those played by the cellist, who was very theatrical. In the same way, at algoraves my attention was polarised by the display's evolving images and not very much by the musician. The screen becomes a body extension, a medium of communication in itself and not only an interface. So, in the end, this action tends to eclipse the rest of musician's body. This is confirmed by, as I have already said, concerts played via the internet. Here musicians are somewhere else, but their screen is projected in front of us and becomes the only music referent.

Therefore, the screen projection is, in my opinion, the most important innovation introduced by live coding movement in computer music practice. However, there are at least another three important aspects to cite and analyse briefly.

The first one is that live coding has transformed the act of programming in an artistic performance. In fact, it seems to be one of the first experiences of transporting the software writing activity, which was until then a process of texting made *before* the real computer "action", in the real time domain. The verb "program" derives from Antique Greek *prò-gramma*, which means literally «to write before». Then, it is an action taken before the real event. With live coding is possible instead to introduce programming in many "on-the-fly" activities as music performances, but also many others. To become a widespread technique, it needs time to develop as science, to refine procedures and dedicated software.

Another important innovation is the introduction of hacker's ethic in music. I would like to introduce briefly what I mean with the word "hacker". First, I would like to state that people who define themselves hackers are not informatics criminals as widely implied by popular mass media. They are called crackers (crack + hacker). The term hacker addressed, at the beginning of computer era, a passionate person who likes to deepen the understanding of something by opening and exploring it directly, with his or her hands, and share this knowledge with other people to improve the community work. Only later, the hackers have started to switch their interest on computers. Even nowadays, the word "hack" alone address to an intelligent trick developed thanks to the deep knowledge of something.

During the movement's inception in the 1950, the first hackers coined the ruling principles of the group: passion, sense of community among members, desire to deepen personal knowledge and to share it with other group's members. (Levy1984). However, the community has changed with the technology developments. The computer has become an increasingly profitable business and this opened a fracture inside the movement. Some hackers tried to reconcile the aspect of gratuity, freedom and sharing

characteristic of the past, with that of business, causing some inevitable imbalances (privacy especially), and some others who remained faithful to the past principles. The two most famous hackers, Richard M. Stallman and Eric S. Raymond, represent two main tendencies in the community. The first is the most faithful to the original hacker's principles and he is a champion of activism for freedom of knowledge, users' privacy protection and community building[5]. The second is more right-wing libertarian and, even though he recognises hacker's ethic principles worth to be followed, he is less convinced about the importance of activism and more flexible about the basic principles' interpretation. He is also a very controversial person, condemned by many hackers for his ideas about the right to own weapons without limitations, about his ideas that this community should not be afraid of dealing with big ICT corporations as Microsoft and some other extremely right-wing ideas[6]. However, this article is not the appropriate space to talk about hacker community internal dynamics extensively. I would only to paint a little portrait of it and to place live coding community inside that. Apparently, it seems that live coders stand more to the Stallman's side than to the Raymond's one. Nevertheless, I have not had the occasion to delve into this aspect yet. It will be one of the subjects of my future research.

To come back now to live coding, I wrote in another article that there were previous examples of similar working method employed for musical purposes, but this happened before the hacker's movement birth, so probably it is not correct to label it as "hacker". However, live coding is the first music milieu in which recent hackers' lesson seems to be recognised and taken as example among its members. The most famous example of this lesson is the making of the Linux operative system by Linus Torvalds. Before live coding inception, there were only a few possibilities to modify and adapt the music instrument to every single personality. One of these methods was to build personalised instrument, either by artisans or by DIY techniques. However, every piece of them has its own constraints and affordances, especially in terms of pitch range, timbre modulation and number of simultaneous voices. With programming, all these obstacles are pushed away, or better, anyone can construct his or her preferred instrument affordances. Before live coding's inception, music was made quite exclusively by commercial music software. This kind of software is usually protected against improper usage and, by consequence, against the violation of original code. Then, the only solution to have customizable music software, apart from make illegal actions, is to employ free music software that is, by definition, freely "hackable" by anyone who wants to do it. Probably, some software engineers can have tried to develop his or her customised and customizable music software before, but they had to start writing it from scratch, because they did not had any pre-existent music program to modify. Live coding appears to be the first music movement that has employed hacker's approach to music software: reuse and rewrite existent programs and adapt them to personal needs. Hence, it has greatly contributed to this new music thread by developing many open source music programs and by adopting a collaborative and informal working method among the community members.

The last aspect to analyse about live coding is the relationship between popular and academic music. Considering that this technique is often employed among dance music practices, especially in Britain but also in North America, Germany and Spain, it has raised a question of how dance music is perceived in academic contexts. Alex McLean and Adam Parkinson (Parkinson and McLean 2014) have recently posed this problem in a conference speech, explaining that dance and academic music is frequently seen as opposed, even though they share, in McLean and Parkinson's opinion, some features and approaches. I do not want here to synthesise the whole article content, but only to emphasise the consequences of a particular statement: «From a research perspective, failure of an algorave can be illuminating. For example, if we find ourselves standing in a room looking at each other, issues of gender disparity which gravely undermines computer music culture becomes difficult to ignore». This couple of phrases addresses a very problematic situation that affects many university departments but not so much the audience in the dancehall. In the article, they demonstrate that this is not a live coding derived issue, but an academic one. In fact, when they have promoted algoraves through standard dance music promotion channels (radios, magazines etc.), says the previous authors, they attracted many non-academics that enjoyed live coded music, without gender distinctions. Nonetheless, gender imbalance remain an issue on the stage, where, both in live coded music and in dance music in general, the event's line-up is very often a list of masculine names with rare interruptions. This is evidently a general problem of many Western societies, not attributable to live coding. However, the McLean and Parkinson paper, in posing this issue, signal that the

[5] See Stallman's personal website: https://stallman.org/

[6] See Raymond's post on Linux Today:
http://www.linuxtoday.com/developer/1999062802310NWSM#talkback_area.

community perceive this imbalance and is reflecting to find possible solutions. To conclude, we can state that live coding, as cross-practice technique, can be used both in academic and dance music and so it can put this two worlds in contact and pose problems localisable on the edge between the two. It can become a bridge for connecting these two apparently distant worlds.

4. CONCLUSION

In this article, I have illustrated the most important features that live coding technique has brought in the computer music performative context. After a presentation of these innovations, where I spoke of improvisation, the screen projection, algorave context and so on, I tried to go a little bit under the surface, interpreting the effect of these new introductions in computer music performances. All this subjects have been discussed after an Ethnographical research, which is the paper's main information source.

I have especially emphasised the aspect of the screen projection, saying that this beam of bright code becomes a sort of musician's body substitute, because it seems that the audience is more attracted by this than by the musicians themselves during performances. The code on the screen, then, becomes the real sound source in the public's view and it is interpreted as the alphanumerical representation of performer's musical thoughts. It may be interesting to interpret this situation with a baudrillardian simulacrum's theory perspective, but here it is only possible to hint at this choice. It may be an interesting subject for a future article.

Among important things said, I would like to mention the relationship between live coding movement and "hacker's culture", which appears to be more than a simple inspiring source. In fact, many principles adopted by live coders, as the development of free programs, the building of an equal community and a collaborative approach, are shared between the two. Probably it can be appropriate to live coders as music hackers.

Finally, I have illustrated how live coding community reflects on the relationship between academia and informal musical practices as that defined "dance music". I have observed, referring to the McLean and Parkinson's article, that live coding can connect these two worlds, considering its double-faced nature of improvisatory technique developed in an academic milieu, but used both in formal and informal music contexts.

Therefore, to conclude, I think that this brief article, thanks to the Ethnographical approach used to collect information, has demonstrated the multifaceted nature of live coding technique and that it has begun to shed light on some important aspects of audience/performer interaction that may have an influence on the future of computer music performance's developments. Ethnographical inquiry method has resulted to be a crucial tool to collect information about the community members, its internal dynamics and to develop a comparative view between live coding and other artistic and cultural contexts.

REFERENCES

Books and articles

Auslander, Philip 2008. *Liveness: Performance in a Mediatized Culture*, New York, Routledge.

Collins, Nick, McLean, Alex 2014. "Algorave: live performance of algorithmic electronic dance music". *Proceedings of the NIME conference.*

Collins Nick et al. 2003. "Live Coding Techniques for Laptop Performance", Organised Sound 8(3): 321-330.

Corbetta, Piergiorgio 1999. *Metodologia e tecniche della ricerca sociale*, Bologna, Il Mulino.

Cortellazzo, Manlio, Marcato, Carla 2005. *Dizionario etimologico dei dialetti italiani*, Torino, Utet.

Hakim, Bez 2007. *TAZ. Zone Temporariamente Autonome*, Milano, Shake.

Himanen, Pekka 2003. *L'etica hacker e lo spirito dell'eta dell'informazione*, Milano, Feltrinelli.

Kozinet, Robert 2009. *Netnography: Doing Ethnographic Research Online*, Los Angeles, Sage.

Levy, Steven 1984. *Hackers: Heroes of the Computer Revolution*, New York, Delta.

Magnusson, Thor 2014. "Improvising with the Threnoscope: Integrating Code, Hardware, GUI, Network, and Graphic Scores", *Proceedings of the NIME conference.*

McLean, Alex, Sicchio, Kate 2014. "Sound choreography <> body code", *Proceedings of the second conference on Computation, Communication, Aesthetics and X*, pp. 355-362.

McLean, Alex, Parkinson, Adam 2014, "Interfacing with the Night: intersecting club music and academia", *Proceedings of Inter-Face: International Conference on Live Interfaces*, 2014.

Raymond, Eric S. 1996, *The New Hacker's Dictionary*, Cambridge MA, The MIT Press.

Ronzon, Francesco 2008, *Sul campo: breve guida pratica alla ricerca etnografica*, Roma, Meltemi.

Shaeffer, Pierre 1966. *Traité des Objets Musicaux*, Seuil.

Stallman, Richard 2010. *Free Software, Free Society: Selected Essays of Richard M. Stallman*, Boston, GNU Press.

Websites

XIX International Seminar of Ethnomusicology, Jan. 30th - Feb. 1st 2014: http://www.cini.it/events/xix-seminario-internazionale-di-etnomusicologia-living-music-case-studies-and-new-research-prospects

Alex McLean's project on live coding and weaving: http://kairotic.org/making-a-warp-weighted-loom/

Algorave official website: http://algorave.com/

Ashlee Simpson lip-syncing concert: http://www.veoh.com/watch/v15490419bqh6yfdc?h1=Ashlee+Simpson+lip+sync+screw+up+on+SNL

Discussion on one Kraftwerk concert: http://createdigitalmusic.com/2015/02/kraftwerk-live-rig-exposed-really-checking-email/

Discussion on some aspects of hacker's culture: http://www.altrodiritto.unifi.it/ricerche/devianza/tavassi/cap1.htm

Eric Raymond's post on Linux Today: http://www.linuxtoday.com/developer/1999062802310NWSM

Gibber official website: http://gibber.mat.ucsb.edu/

Ixilang official website: http://www.ixi-audio.net/ixilang/

Live Coding Research Network's website: http://www.livecodenetwork.org

Marije Baalman's project: https://www.marijebaalman.eu/?p=16

Richard Stallman's official website: https://stallman.org/

Sonic-pi official website: http://sonic-pi.net/

Supercollider official website: http://supercollider.sourceforge.net/

Petition against Ashlee Simpson: http://www.boston.com/ae/celebrity/ashlee_petition/

Tidal official website: http://yaxu.org/tidal/

Toplap blog website: http://toplap.org

PIETRO GROSSI'S LIVE CODING. AN EARLY CASE OF COMPUTER MUSIC PERFORMANCE.

Giovanni Mori
University of Florence
giovanni.mori@unifi.it

ABSTRACT

Pietro Grossi has been one of the first pioneers in computer music in Italy. His work, however, is still quite underconsidered because his art's concepts was judged utopistic, without a connection with contemporary cultural manifestations and harshly anti-academic. Instead, in my opinion, it seems to be now the right moment to revalue his work, in order to understand from where some computer music practices have their roots. In this article, I compare the concepts and the work methods developed by Grossi to those employed by live coders. My aim is to demonstrate that the Italian composer's concepts were not only ahead of his time but also anticipatory of future developments.

1. INTRODUCTION

Since my first live coding experience, I feel very enthusiast about this innovative technique. I have immediately perceived something different in this way of making computer music, compared to other similar types of performances. Here, liveness perception is very strong, audience is completely aware to attend to a live event. Listeners, instead, do not generally experience the same sensation, during other types of computer music concerts and this creates a sort of alienation and discomfort among audience members.

I have attended to my first live coding symposium at the University of Sussex in Brighton. Speaking with people and listening to the paper, I became increasingly aware that live coding appears to be very similar to the experiments made by the Italian computer music pioneer Pietro Grossi, who has done his first experiences in algorithmic composition about 50 years ago. After a review of my MA thesis, which was exactly pinpointed on Grossi's computer music and programs, I am feeling quite sure about this familiarity. Naturally, there are some differences between Italian composer's and his younger "colleagues'" approaches, due to the diversity of the technology employed. This article's aim is to help the rediscovering of Pietro Grossi's work and theory, and to bring his thoughts to the attention of those I consider in some ways his successors. I also believe that Italian composer's work may have had an influence, perhaps unconscious and indirect, on the formation of the live coding music practice.

2. INTRODUCTION TO PIETRO GROSSI'S COMPUTER MUSIC

Pietro Grossi has been an "underground" figure in the Italian and European music field during the second half of the XX century. He was born in 1917 in Venice, but he moved very young to Florence, where he became the first cellist in the orchestra of the Maggio Musicale Fiorentino, a well-known Italian music festival based in Florence, during the 1930s. At the beginning of the 1960s, he become interested in analogic electronic music and he was guested at the Studio di Fonologia Musicale in Milan for fifteen days. This studio was famous because there, Luciano Berio, Bruno Maderna and some other famous music composers were working. The result was *Progetto 2-3*. Some algorithms designed the sounds' architecture by intersecting a group of six sine waves whose mutual ratio was in slow evolution. After this experience in Milan, he was so enthusiast about this new music technology that he decided to found his own private phonology studio called *Studio di Fonologia Musicale di Firenze* or S2FM. This was one of the first private owned studios in Europe. Moreover, Grossi soon developed a large network of contacts with many important electronic music studios around the world.

Since the late 1950s, big mainframe computers became to spread throughout Western countries and, at the same time, many technicians and musicians started to experiment these machines' music possibilities. Among pioneers, we can surely list Leonard Isaacsson, Lejaren Hiller, Gottfried Michael Koenig and, last but not least, Max Mathews. Grossi began to be interested in computer music during the first half of the 1960s, when he hosted a radio program centred around "innovative music" in general (Giomi1999). However, the first Grossi's experience with calculator took place in Milan, in the Olivetti-General Electric Research centre. Here, aided by some internal technicians and engineers, he managed to compose and record some of his first computer music works. They were, for the most part, transcriptions of Western classical music. However, there were some exceptions, for example a track called Mixed Paganini. The name was a consequence of the original music material, which comes from the Fifth Paganini's *Capriccio* for solo violin. The Olivetti GE-115 computer played that piece in various ways: inverted, accelerated, retrograded etc. Practically, Grossi modified, aided by some rudimental music programs, the original sound material. People that was able to listen to Grossi's music complained about that because they felt this approach as a "profanation" of the art's sacredness and "aura". A later collection of Paganini's *Capricci*, recorded in Pisa, was reviewed by Barry Truax on Computer Music Journal (Truax1984).

In this early period, Grossi was not very satisfied by his work at the GE-115 computer because he cannot experiment freely. Moreover, the corporation's steering committee has lost much of the previous interest in the Venetian composer's music experiments. Then, Grossi began to look for a different place for developing his thoughts, and he finally managed to be admitted at the IBM Research Centre in Pisa, inside the CNR Institute (*Centro Nazionale per la Ricerca*: National Research Committee). Here he was able to work with a video terminal and alphanumeric keyboard, a very new technology for that time. Thanks to this device, he started to learn FORTRAN programming language and developed his first music software DCMP (Digital Computer Music Program). This was one of the first live coding programs in my opinion, because, using that, the performer was able to produce and reproduce music in real time by typing some specific commands and the desired composition's parameters. The sound result came out immediately after the operator's decision, without any delay caused by calculations. There were many reproduction choices inscribed in this software: it was possible to save on the computer memory pieces of pre-existing music, to elaborate any sound material in the hard disk, to manage the music archive and to start an automated music composition process based on algorithms that worked with "pseudo-casual" procedures. There were also plenty of choices for piece structure modifications. One of the most important aspects of Grossi's work was that all the interventions were instantaneous: the operator had not to wait for the computer to finish all the requested operations and then hear the results. Data calculation and sound reproduction were simultaneous. This simultaneity was not common in the computer music field of that time and Grossi deliberately chose to work in this way, losing much on the sound quality's side. His will was to listen to the sound result immediately. He was a musician and he probably could not stand to fill a chart or to write a long string of commands and parameters and then wait the whole calculation and recording process before being ready to play the result. He wanted "all and now", as he said very often (Giomi1999). With that expression, Grossi would address that when he inserted some commands on the computer screen, he did not want to wait anything before hearing the result. He defined himself as "lazy", *pigro*[1] in Italian. Lazy address, in Grossi's sense, to a person who is aware that his or her time is limited and do not want to waste time in doing useless things or in waiting for something when it is not necessary. His or her actions should be as efficient as possible, to save time for enjoy the life fully. Eric Raymond[2] has stated some similar for describing the hacker's philosophy. This approach should sound familiar to live coders as well, considering that they are musician but also programmer and many of them seems to have something in common with free software community, in Stallman's sense (Stallman2010).

The DCMP was compiled in the early phase of computer technology development. At that time, the calculation resources were low and, to obtain the just cited real time reproduction, it had to ask for very low quantity of data. Therefore, the Venetian musician chose to write very light software, able to modify only parameters that required a few calculation resources: pitch and duration. Timbre synthesis needed a big amount of data, so that choice was temporarily discarded and all the sounds were reproduced with square wave timbre. This waveform was generated by extracting the binary status of a motherboard's exit pin controlled by the software. This exit had only one bit, so the sound wave generated was the result of this bit

[1] Grossi as nickname in some of his records used *Pigro,* or better *PiGro*. It is a play on words made by using his name first letters: Pietro Grossi.

[2] Eric Raymond's post on Linux Today: http://www.linuxtoday.com/developer/1999062802310NWSM

status changing. In this way, the computer did not employ any resources for calculating the sound synthesis, saving them for music production process. Grossi was not very interested in the quality of sound output in this first phase in Pisa. What he cared particularly was to be able to work in real time, or, in other words, to have the choice to listen immediately to what he typed on the video terminal's keyboard (Giomi1995).

Some technology improvements permitted Grossi to implement timbre elaboration around the first half of the 1970s. In this period, the CNR steering committee decided to project and build a brand new and innovative sound synthesiser, digitally controlled but with analog sound synthesis technology. It was launched in 1975 and called TAU2 (*Terminale Audio 2a versione* – Audio Terminal 2nd version).

Grossi was the new machine's software developer. For making this, he took as a reference the previous DCMP, adding to it many new features and a brand new section dedicated to TAU2's management. The new program's name was TAUMUS. The new software and hardware system could play up to twelve different voices simultaneously. These twelve voices were divided in three groups, composed of four channels each. The operator could choose to assign a different timbre to every single group, which was modulated using additive synthesis with seven overtones. Every overtone could be controlled individually by software.

It may be interesting to mention that using these two music systems - DCMP and TAU2-TAUMUS - Grossi developed his most innovative artistic conceptions as those of, for example, infinite music and automated composition. With the first, Grossi addressed to a music production process without predetermined end: a music that flows forever, without interruptions. With automated composition, instead, he defined a music created without any human intervention after the start of elaboration process. Every decision was a computer's task, or better, a software's task. This last one is an extreme example of aided music composition. This fundamentalist approach may have stimulated the developing of automated process inside computer music practice. To make an example in the live coding field, if someone types on software's main window a command to let the computer reproduce a beat pattern, the computer will play it automatically, and the performer can dedicate his or her attention to the next task. That pattern will not stop without external intervention. Andrea Valle[3] states that these two approaches own to "instrument logic" and "machine logic". The instrument needs continuous stimulation to be operative. The machine, instead, need only to be programmed and then it will work respecting that program.

In the first period of music experiments with DCMP, the calculation resources available to Grossi were very low and this did not enable the musician to change the result in real time while the process was running. Then, it was necessary, if he wished to modify one or more parameters during the elaboration, to stop the entire process, make the desired changes and restart it from the beginning. When the process was an automated composition, it was not possible to vary any parameter at all, because the program took every decision autonomously. But, as it will be shown later, there are some evidences in Grossi's music applications, especially the latest ones, that shows his aim to develop the sound output in real time, in a surprisingly similar way to that of live coding performances.

3. GROSSI'S SOFTWARE AND HARDWARE

First, I would like to introduce the DCMP's architecture and functions. Grossi started to develop his program in 1969, when he began his career at the CNR in Pisa. The software was written in FORTRAN and designed to work on some different machines: IBM 7090, IBM 360 model 30, 44 and 67 (Grossi and Sommi 1974). DCMP was provided by many different commands, both musical and non-musical. The last ones were principally commands dedicated to the musical archive management: erase, move, rename files and so on. The first ones, on the other hand, were designed to modify pre-existing pieces or to compose and play new music by using both automated and non-automated compositional procedures. The system could play about 250 notes between the frequencies interval 27 Hz - 4900 Hz. The minimum step between two sounds was a third of a semitone. The whole pitch range, then, was approximately that of a grand piano. It was possible to play notes addressing to them in various ways: by using alphabetical or Italian system or by using the note's number in a default note chart. It was not possible to manage neither the timbre nor the dynamic nor, finally, the number of voices, because of sound synthesis method, as explained above (Grossi and Sommi1974).

All the music pieces created with the DCMP could be saved in the computer's archive. From there, the operator could recall any track with specific commands, upload the chosen one in the work memory and

[3] See Andrea Valle's interview: http://www.soundesign.info/interviste/intervista-con-andrea-valle

elaborate, modify and finally play it. The sound processing instructions were numerous: MIX, for example, mixed two tracks' parameters pre-uploaded in the working memory. It can exchange the note's durations of one track with those of the other, or sequencing two notes of the first and one of the second track, and so on. Some other interesting commands were: GOBACK, that reproduced the track placed in the work memory from the end; INVERT, that inverted the pitches' ratio in respect of a pre-determined value; SHUFFLE, that mixed randomly two tracks' parameters; and so on. The operator could stop the music reproduction in every moment, change any parameter, and restart the process. However, it is not possible to define this as performance, because the system needs to stop the elaboration process to act every kind of intervention. It were not interactive (Grossi and Sommi 1974).

There were another choice for creating music employing DCMP: automated composition. There were some commands dedicated to this function, but the most interesting one was, in my opinion, CREATE. The only action requested to reproduce music through this command was to type the desired total piece duration on the screen and start the process. The program then started to play random notes generated by specifically designed algorithms based on Markov and Lehmer chains. Those algorithms were studied to create as less repetitive flux of sounds as possible. The time limit for automated compositions were fixed by software at about 300 years, then almost endless for a human being (Grossi and Sommi 1974).

Thanks to the technology advancements, in 1972 Pietro Grossi and the CNR team began to project the TAU2 synthesiser and the TAUMUS software. The TAU2-TAUMUS system was able to reproduce about the same note range of DCMP system. There were many differences, however, between them. The most important one was that, by using the newest one, the operator could play many melodic lines with different timbres. In fact, the TAU2 was able to reproduce up to twelve voices simultaneously, grouped in three different set. Each set could assume a distinctive timbre. As I have already written, the sound synthesis technique employed was the additive synthesis, with seven overtones (Grossi and Bolognesi 1979).

Another crucial TAU2-TAUMUS' improvement in respect of DCMP was the *modelli modulanti* (modulating models): they were a sort of patches that acted on some musical parameter. The most interesting aspect is that these patches worked in real time, and the operator did not need to stop the process to change the sound output, as happened before. This was a critical innovation under the performative point of view, because then Grossi was able to play and to interact in real time with the software, by typing instructions on the keyboard without stopping the sound flux. It is clear, then, that his interest was to be able to work with sounds with a real-time interaction. In this way, the computer becomes a true music instrument and not only a sort of augmented music player.

All these aspects get Grossi very close to the newer live coding practice. Moreover, what I have explained shed more light on the Venetian's research direction, making us wondering about his potential goal if he was remained in a stimulant research milieu. In fact, in 1982 the TAU-TAUMUS system developing was abandoned for the increasing number of bugs and failures. After this long period in Pisa, he began to work at another CNR institute in Florence, called IROE, where he continues his experiments with a new synthesiser: the IRMUS. This last one, unfortunately, was very less powerful and ambitious than the TAU2, because it could reproduce only two voices simultaneously. Nevertheless, IRMUS was very flexible on the timbre side, thanks to the FM synthesis method adopted. Unfortunately, this synthesiser was not sufficiently powerful and stimulant for Grossi and he decided, after a brief period, to quit his music experiments (Giomi 1999).

Grossi, however, was a very curious person, so he could not stay without a computer and then he decided to buy one of the first popular PC on the market, the Commodore 64, and to make experiments in the computer graphics field. He came up to this decision after discovering that these machines were more powerful on visual than on audio processing side. Therefore, he started to adapt all of his programs to the new project (image elaborations in real time), recycling also his concepts about sound processing (Giomi1999). Even on this aspect, there may be a link between Grossi and live coding. In fact, in Lisbon, during a performance organised for the conference on Live Interfaces, I have seen that live coding is not only employed in music but also in image processing practices. Then, probably, further research on this aspect will underline many other affinities between Grossi and live coding movement.

I would like to return briefly to the TAU2-TAUMUS system, because at the end of the 1970s it faced an important improvement: it became possible to control the synthesiser remotely. This new feature was deemed so important that the software name becomes TELETAU. Then, it was possible to generate sounds and listen to the result immediately, ideally everywhere, thanks to a telephone connection between the

TAU2, the computer and a remote terminal (Grossi and Nencini 1985). However, similar experiences were possible also with DCMP and the first one took place as early as 1970 and was called “musical telematics”. Grossi made his first experience of this kind during a conference on technology in Rimini in 1970, where the musician reproduced many of his compositions and random sounds as well, by employing a video terminal connected via telephone to the CNR's computer in Pisa. RAI, the Italian public broadcasting company, lent its powerful FM radio bridges to send back sound signals from Pisa to Rimini. It is likely to be the first official experiment of musical telematics in the world (Giomi1999).

After this exciting event, Grossi went on developing his project along the others and the result was the just cited TAUMUS' update: TELETAU. In a similar way to that of the DCMP, this software enabled whoever connected to the BITNET computer network, to exploit remotely the CNR computer calculation resources and to immediately listen the sound results produced by TAU2. Practically, every person connected to this network was able to use the TAU2-TAUMUS system as if he or she was in Pisa in front of the machine (Bertini et al.1986).

Unfortunately, this service lasted only for a few years. There were many reasons for this decadence: the rapid ageing of the TAU2 synthesiser, with frequent breakdowns and bugs that increased dramatically the maintenance effort and costs; the rapid diffusion of personal computers; the high cost of data transmission and, last but not least, the low quality of sound output due to slow data connection.

This working practice was strongly promoted by Grossi because, since his career's inception as electronic musician, he struggled to develop a communitarian and egalitarian work method, based on shared knowledge, on free re-elaboration of pieces created by different musicians and on spreading as much as possible this conception to other similar working groups or single artists. In addition, this approach appears to me as anticipatory of that typical of live coding milieu. In fact, in my view, live coders too undertake to create an egalitarian community, based on almost the same philosophy promoted by Grossi. Nearly all the programs used by live coders are open source, or free software in the Stallman’s sense (Stallman2010), and people inside this community often modify and create customized software by reprocessing those of others. Moreover, I have assisted in Birmingham to an online performance at the Network Music Festival, where three live coders located somewhere around the world (Alex McLean, David Ogborn and Eldad Tsabary) met in a virtual place (the remote server) to play the concert we assisted to. I think that Grossi's work on musical telematics pointed out to similar kinds of music making.

Probably however, his artistic conceptions were too ahead of his time. The technology and probably the culture were not still ready to accept the detachment of music from human agency. The process of body dematerialisation has started long time before Grossi's work: radio and telephone inventions gave birth to all the subsequent technology transformations. Electricity carried out a central role in this process (McLuhan1964). Probably the development of mediatization, whose implication is described very well by Philip Auslander in his well-known book *Liveness*, is a natural developing of the mass media technology and requires a recontextualisation of the body's role in our contemporary society. The physical co-presence is no more necessary to communicate. It is apparent that, with the spreading of social networking and the Internet in general, the most part of communication between humans, at least in Western societies, passes through mass media. Live coding stands inside this process as protagonist, because its practitioners seems not to be astonished in front of a remote driven performance. Grossi was probably more tied to the performer’s materiality, because he never plays computer for a remote public in an acousmatic setting. He dematerialised the music instrument, but not the musician. Live coders instead do not have problem to express themselves through online connections. Nevertheless, we all express ourselves regularly through the same medium when we use social networks or the Internet in general and our identity is increasingly constructed through the Internet.

Finally yet importantly, Grossi took advantage of the TELETAU system by teaching to his pupils in Florence (actually he has taught at the Conservatory of Florence since the early ‘60s) without moving from his class. He was also the promoter of the first Electronic music course in an Italian conservatory in 1962 and, as well, the first Computer Music course in 1970s.

4. CONCLUSION

In this article, I tried to show the main characteristics of Pietro Grossi's computer music experimentations. Beside this, I have underlined the most striking similarities between the Italian algorithmic music pioneer and the live coders' works, in particular apropos of the respective processes of music making.

After having synthetically explained Grossi's approach, I would like to conclude this presentation with some reflections on the two central experiences illustrated in this article, to give more weight to these affinities.

I have said above that Grossi began to learn programming immediately after his arrival in Pisa, with a well equipped studio and a numerous team of engineers. He started studying programming languages because he knew precisely what he wanted and he was very focused on realising them. He strongly believed that teamwork was the better way to obtain a good and satisfying result, so he struggled to become an active member of this group. This is a strikingly similarity between Grossi and live coders, because also many of them have developed their own music software, sometimes aided by a community of developers, placing themselves on the edge between the world of musicians and that of programmers.

Another common feature between the Venetian composer and his "heirs" is that they all demonstrate a strong interest in sharing their knowledge on egalitarian basis, by using or writing programs without copyright licenses and by creating a multidisciplinary and collaborative work community. In fact, Grossi was one of the first artists that believed and worked in a way typical of what today is defined open source or free software field. He had very soon adopted a working method that anticipates the principles of the free software communities. To confirm this, it is sufficient to say that, starting from the work at S2FM (Music Phonology Studio of Florence), he gave a collective name to the records created there name because he thought that they were products of the whole community's work. Moreover, the whole studio staff often recycles pieces recorded by external artists to create something else, in a sort of proto-remixing practice. Grossi also gave his audio materials freely to musician who requested them, but also by sending tapes all around the world. Inside every parcel, he always inserted a little letter to invite the addressee to rework the tape content personally, without any limit and without asking any copyrights. Something similar happened as well for Grossi's software in a later period, when he was able to spread his programs out of the CNR studio in Pisa. In fact, the TELETAU system was explicitly created to broaden TAUMUS-TAU2 employment.

When Grossi began to work at home with his new Commodore 64, he went on following the previous path. He was in constant contact with many artists and, as soon as he managed to own an Internet connection, he designed his own website and shared all his new and old software, inviting visitors to download and modify it freely. Thanks to this pioneering behaviour, he has been recognised as one of the open source music software's ancestors among some hacker artists (for example: http://www.hackerart.org/corsi/aba02/taddepera/index.htm).

Another convergence point between Grossi's and live coders' approach is the TELETAU system: the use of a network to make music from remote places. Grossi was able to program a system that, by using a video terminal, it was possible to run the music software in Pisa remotely and to hear immediately the sound result. Live coders use sometimes a similar approach. The above-cited example I made about Network Music Festival in Birmingham (UK) suits perfectly here, but there are also some other examples as that of BEER ensemble, which work and play through a VPN connection to synchronise and to send music to each other (Wilson et al. 2014). In many live coding contexts, then, the network is a crucial tool for working in real time. In my opinion, Grossi was experimenting precisely in this direction. As I have hinted above, however, the live coder's case sometimes imply that the performer becomes invisible, located in a remote place in respect of that of the sound reproduction. Grossi instead wanted to be present in the same place where the audience heard the sound.

Another important similarity between Grossi and live coding is the user interface. In both cases, the music tool employed by the musicians is the text, and both require written commands and parameters to obtain precise audio results. Additionally, each refers more to the programmer's perspective than to the musician's one because there is not a direct correspondence between gestures and the sound heard (see note 3). In Grossi's case, however, choices are fewer than in the live coding one. Nonetheless, the performing practices of each example seems to be quite similar, because in every case, there are commands that recall precise algorithms, launched and modified normally by typing their name and parameters on the screen. Consequently, it seems that keyboard, screen and text itself have together a central role in playing and representing the musical ideas.

Finally, it may be important to cite briefly the *modelli modulanti*'s case, the above-cited patches that modify the final audio output in real time. The Grossi's effort to develop these little programs demonstrates, in my opinion, his intention to interact with the computer on the fly. By consequence, this action switches Grossi's musical practice from music composition to performative context. This crucial aspect of the Grossi's computer music work is worth to be emphasised here, because it carries the Venetian composer very close to live coders. The *modelli modulanti* demonstrate Grossi's strong interest to employ the computer as a real music instrument, or, in other words, as a device able to translate music thoughts in sounds in a performative way. Grossi himself confirm his interest about real-time music procedure, speaking about this aspect very often in his books and articles (Giomi1999).

To conclude, then, after having explained all these similarities between Grossi's and live coders' work (a predominant textual interface, experimentation of online performances, an egalitarian and communitarian approach to programming and to culture in general) it may be appropriate to define Grossi as a proto-live coder.

REFERENCES

Auslander, Philip, 2008. *Liveness: Performance in a Mediatized Culture*, New York, Routledge.

Bertini, Graziano et al. 1986. “TELETAU. A Computer Music Permanent Service”, *Proceedings of the International Computer Music Conference*, San Francisco.

Calliau, Robest, Gillies, James 2000. *How the Web Was Born: The Story of the World Wide Web*, San Val Inc.

Collins, Nick et al. 2003. "Live Coding Techniques for Laptop Performance", *Organised Sound* 8(3): 321-330

Giomi, Francesco, Ligabue, Marco 1999. *L’istante zero. Conversazioni e riflessioni con Pietro Grossi*, Firenze, Sismel Edizioni del Galluzzo.

Grossi, Pietro et al 1979. *Modalità operative del Taumus, software di gestione del Terminale Audio TAU2 2° versione*, Pisa, Internal CNR report.

Grossi, Pietro, Nencini, Graziano 1985. *Teletau. Software sperimentale per la Telematica Musicale. Release 0.0 – Luglio 1985*, Pisa, Internal CNR report.

Grossi, Pietro, Sommi, Giorgio 1974. *DCMP versione per il sistema 360/67*, Pisa. Internal CNR report.

Magnusson, Thor 2014. “Herding Cats: Observing Live Coding in the Wild”, *Computer Music Journal* 38(5): 8-16.

McLuhan, Marshall 2011. *The Gutenberg Galaxy. The Making of Typographic Man*, Toronto, University of Toronto Press.

McLuhan, Marshall 2001. *Understanding Media*, London and New York, Routledge.

Stallman, Richard 2010. *Free Software, Free Society: Selected Essays of Richard M. Stallman*, Boston, GNU Press.

Truax, Barry 1984, “Review of Pietro Grossi: 24 Capricci by Niccolò Paganini”, *Computer Music Journal* 8(1): 59-60.

Wilson, Scott et al. 2014. “Free as in BEER: Some Explorations into Structured Improvvisation Using Networked Live-Coding Systems”, *Computer Music Journal* 38(5): 54-64.

Livesolving: Enabling Collaborative Problem Solvers to Perform at Full Capacity

Steven L. Tanimoto
University of Washington
Seattle, WA 98195, USA
tanimoto@cs.washington.edu

ABSTRACT

Collaborative problem solving is a key methodology for tackling complex and/or contentious problems. The methodology is supported by computer and communication systems that bring human solvers together with computational agents and provide clear protocols for exploring and rating alternative solution approaches. However, these systems can be challenging to use due not only to the complexity of the problems being solved but the variety of abstractions involved in managing the solution process, e.g., problem representations, collaborations, and strategies. This paper offers new ideas to help the human users of such systems to learn and work more effectively. It also suggests how problem solving may sometimes be carried out in performance contexts similar to those of livecoding improvisational music. Most important here is the identification of seven forms of liveness in problem solving that may heighten a solving team's sense of engagement. Common themes among them are increasing solvers' awareness and minimizing latency between solver intentions and system responses. One of the seven livesolving forms involves solvers in tracing paths within problem-space graphs. This and the other six forms derive from experience with a system called CoSolve, developed at the University of Washington.

1. INTRODUCTION

1.1 Motivation

Problem solving has somewhat different interpretations in different disciplines. In mathematics it usually consists of finding "answers" in the form of numbers or mathematical expressions, and it sometimes means coming up with a multi-step proof of a formal proposition. In business administration, problem solving typically refers to the decision making that owners and CEOs do in order to manage their operations or expand their markets. In psychological counseling, problem solving refers to the resolution of personal relationship conflicts, family strife, or depression. In engineering, problem solving may require the design of a machine or part that will satisfy a set of specifications. In spite of having somewhat specialized connotations in particular fields like these, people solve problems almost continuously during their lives, some of them very minute and subconscious such as the avoidance of puddles and stones when walking, and some of them more deliberate, such as deciding what groceries or brands of items to buy at the supermarket. Our aim is to support solving in some of these areas, but primarily situations in which there is enough value in formalizing the problem that people will be willing to consciously think about the problem in its formalized representation. Puzzles often figure prominently in discussions of computer problem solving, and I use them here because they facilitate conceptual explanations. However, I mention a variety of other problems, such as understanding climate change, to clarify other points.

Three reasons for supporting problem solving are (1) to help get important problems solved such as climate change, poverty, or world peace, (2) to help teach people how to solve problems in new and potentially better ways, and (3) to facilitate the art and performance of problem solving -- e.g., "livesolving." In our own

group, a principal motivation is to find new ways in which technology can further empower people in problem solving. This paper focuses on forms of problem solving that resemble livecoding or live programming, as these terms have come to be known. Liveness in problem solving promises to contribute to either the effectiveness with which human solvers can accomplish the solving itself or the effectiveness with which the humans can artfully demonstrate solving activity. Livesolving is a new avenue by which to improve people's ability to solve problems and to share the share the solving process.

1.2 Definition

Livesolving refers to problem solving activity performed under circumstances associated with live programming: (a) the use of computational affordances that respond in real-time, (b) the potential presence or virtual presence of “consumers” such as audience members at a performance, and (c) improvisational elements of creative synthesis and non-retractability. Whereas live programming typically involves the editing of program code, livesolving involves the manipulation of problem structures such as formulations of problems (“problem templates”), state variables, and fragments of solution paths. By supporting and studying livesolving, we seek a better understanding of best practices in the design of methodologies and tools to support humans in complex, creative work.

2. PRIOR WORK

2.1 Theory and Systems for Problem Solving:

When artificial intelligence research started in the 1950s, it paid particular attention to problem solving. The "General Problem Solver" (GPS) developed by Allen Newell, J. C. Shaw, and Herbert Simon (1959), involved articulating a theory of problem solving that was subsequently developed and used both for designing computer agents and for modeling human problem solving (Simon and Newell 1971; Newell and Simon 1972). I refer to the core aspects of the theory as the "Classical theory of Problem Solving" (and just "Classical Theory" below). Numerous researchers have contributed to the theory, and a number of books have explained the ideas well (e.g., Nilsson 1972, Pearl 1984). The primary motivation for all this work was to be able to build intelligent, automatic problem solving agents. The secondary motivation was to find a new theory for human problem solving.

Others, including my group, have had a different motivation: to support human problem solvers with appropriate computer technology. At the Swiss Federal Institute in Lausanne, a mixed-initiative system was developed that supported a human problem solver in the formulation and solution of problems involving combinatorial constraints (Pu and Lalanne 2002). In my own group, the focus has been on supporting teams of solvers who collaborate on the solution to problems, with the assistance of computers. A system we developed, called CoSolve, embodies the classical theory and gives a solving team web access to a shared session that includes an explored subset of a problem space (Fan et al 2012). Two user roles are supported by CoSolve: "solving" and "posing." The solvers work in small teams to construct "session tree" objects that represent realizations of portions of problem spaces. Figure 1 is part of a screen shot from CoSolve showing a session tree for a Towers-of-Hanoi puzzle. The posing role in CoSolve engages users called posers in "scaffolded programming" by providing online forms that hold fragments of Python code to represent the components of a problem formulation. Related to posing are the activities of understanding and transforming information (e.g., see Russell et al, 1993, and Mahyar and Tory, 2014, on sensemaking and Kearne et al, 2014, on ideation.)

CoSolve was designed and built after a previous prototype, called TSTAR, proved to have so much latency for sharing of solving activity within a team that solvers did not feel that they were really working closely together when solving (Tanimoto et al 2009). CoSolve, however it managed to reduce the latency, still had enough latency that the solvers would sometimes get annoyed, and we are working on a successor system

in order to further reduce such latency. It is the latency issue, for the most part, that connects the topic of computer-supported problem solving with livecoding.

2.2 Forms of Live Programming:

Reducing the latency between coding (editing code) and execution (seeing the results) has been a fundamental goal of live programming (Tanimoto 1990, 2013). The various ways in which this takes form depend on the style of programming and the available technologies. Without going into details here, we can simply say that live programming has two key aspects: (1) editing the running program without stopping it, and (2) reducing the latency between code editing and execution of that code to an imperceptibly small interval of time.

Livecoding usually refers to a particular form of live programming in which the programming is part of a musical performance, the music being generated by the code as the code is modified. This kind of programming is a form of improvisational performance, and it would have this character even if the output were changed from music to dynamic visual art. Livecoding is of particular interest to those who study the psychology of computer programming, not only because it is an unusual context for programming, but because it puts the programmer rather than the language or the software at the center of attention (Blackwell and Collins 2005). Later in this paper, I'll consider the related case where the output is problem solving state, rather than music or graphics per se.

2.3 Cognitive State of Flow

Bederson (2004) has argued that human-computer interfaces that support cognitive tasks should be designed to keep the user in a psychological state of "flow." Such a state can be characterized as one of intense focus as a task progresses. In order to support such user focus, the interface must be designed to avoid undue latency and to anticipate the possible next states of user attention, providing some of the information that will be needed prior to any delays due to communication networks or computer processing.

3. FORMS OF LIVESOLVING

By considering the several ways in which users of our CoSolve system interact with it, one can identify corresponding ways to reduce the latency of those interactions. In this section, seven forms of livesolving are described that derive from this analysis.

CoSolve engages human solvers in posing problems, initiating solving sessions, and solving the problems. In this paper, I neither address the posing process nor the session initiation process, and thus the focus is on solving. Solving in CoSolve consists in the deliberate selection of existing states in a session, the selection of operators to apply to them, and in some cases, the selection or specification of parameters to those operators. One "turn" consists of selecting one state already in the session, selecting one operator, and if the operator takes parameters, selecting and/or writing the values of those parameters. When the user inputs this information, typically over the course of several seconds in time, the choices are relayed to the server which computes a new state and sends it back to the client. The new state, added to the session by the server, is then drawn on the client screen as a new part of the tree representing the session so far.

Here are seven forms of livesolving in the context of a system that supports collaborative problem solving. They are discussed in greater detail in later sections. Not supported in CoSolve (with the possible exception of #5, livesolving presentations), we are building them into a new system code-named "Quetzal." The order in which the forms of liveness are listed here may seem arbitrary, but corresponds to my estimates of importance to design of next-generation general solving systems.

The first promotes thinking strategically at the level of problem spaces, which without the immediacy of visualization and live interaction is conceptually difficult for solvers. The second dispatches with any unnecessary latency in adjustment of parameters, something needed in solutions of many problems. The next two can affect the relationships between a solver and her/her collaborators and/or audience members. Livesolving form number 5 is associated with the design of dynamic objects such as animations or computer programs. Form number 6 means that solving decisions that might be considered finished can be revisited without necessarily undoing all the work done since. Finally, livesolving form 7 directly supports an experience of flow that offers a potentially exciting new modality of solving that might be effective for some problems or as a training aid for human solvers.

1. "Drawing and co-drawing solution paths." The user traces a path (or two or more users co-draw a path) through a display of the problem-space graph. The system supports the drawing process and limits the trace to legal paths, according to the problem specification. The turn-taking time of CoSolve-style interaction is reduced to an almost imperceptible delay. The possible challenge for the user of staying on a legal path can be limited by the system through intelligent interpretation of the user's drawing intent, as explained later.

2. "Live parameter tuning." One part of the operator-application turn described above is the specification of parameter values. CoSolve requires that the user make a commitment to a particular parameter vector prior to operator application. This means that the user cannot easily make a series of small adjustments to a parameter value using feedback from the system. Live parameter tuning means that, upon operator selection, the system will display a new state showing the consequences of the current parameter values without requiring a commitment to those values, and furthermore, there is instant updating of the state and its display as the parameters are adjusted using appropriate input widgets such as sliders.

3. "Synchronous co-solving." Whereas CoSolve's group awareness feature allows each solver to become aware of other users' edits to the solving-session tree through small graphical indicators, actual updates to the user's session-tree display do not occur until the user requests them. On the other hand, with fully synchronous co-solving, any team member's edit to the solving-session tree is automatically propagated to each team member's view of the tree with only a small, deliberate delay introduced so that a smooth animation can provide continuity between the old and new views.

4. "Livesolving presentations." This is solving a problem, using computer technology, in front of an audience or video camera. While it doesn't require any new technology per se, it can benefit from a variety of standard technologies and affordances, such as means for highlighting, annotation, bookmarking, and linking to multimedia facilities.

5. "States alive." In CoSolve, states in a session are shown as nodes in a tree, with static images as graphical illustrations of each state. Additional details of state information are available to users by clicking, after which JSON code is displayed. The restriction that states be presented in this manner means that certain kinds of problems, such as animation design problems, cannot be solved in the most natural manner. To improve that, states can be permitted to have dynamic displays. Thus, if the state represents a dynamic process, such as a computer program, then states-alive liveness will allow the user to observe, within a session tree or problem-space visualization, a collection of running animations, reflecting the running programs the states represent.

6. "Ancestor modification." In CoSolve, each turn of operator application is a commitment. States can only be added to a session, not deleted. Furthermore, existing states cannot be modified. That limitation simplifies session management in collaborative contexts, and often there is no problem having a few extra states around, representing evidence of solver exploration prior to finding a good path. The difficulty comes

up when a long chain of operator applications hangs off of an ancestor that needs a change. CoSolve treats the ancestor as immutable, and in this sense of no longer supporting interaction, "dead." With ancestor-modification liveness, edits to ancestors are permitted, and they have the effect of raising such states from the dead and letting them be modified. There are two alternative policies for this. One is that the ancestor is immediately cloned, with the original remaining immutable. The other policy is that there is no clone, and the ancestor itself will be modified. Modification may involve simply re-specifying or re-tuning the parameter values used in the operator that created the ancestor. However, it may also involve a change requiring the use of an entirely different operator and possibly new set of parameter values as well. In either case, a change to the ancestor may require either of two kinds of changes to its descendants: (a) updating of the state variables affected by the changed parameter(s), (b) pruning of descendant subtrees due to violations of operator preconditions by the new ancestor or by other updated states.

7. "Driving an agent." Whereas the first form of livesolving (drawing and co-drawing solution paths) achieves its dynamics by the users' hand or stylus motions, in driving-an-agent liveness, the solving process proceeds with a form of momentum (speed, direction in the problem space, restricted deceleration) that means steering decisions must be made at a rate that avoids the wrong turns and dead ends that would be made by the agent's default behavior. Computer agents for problem solving come in many forms, and the most obvious application of this form of liveness works with a depth-first-search agent. However, it can apply to almost any solving agent that can accept input over time. To make this form of liveness effective, a good display must be provided to the driver (the user or users) so that each decision can be made on the basis of the best information available at the current point in the search.

4. LIVESOLVING WITH FINITE PROBLEM SPACES

In this and the following two sections, I discuss the seven forms of livesolving in more detail. I have chosen to categorize four of them roughly according to the nature of the problems they might be applied to and the other three by their "social nature." However, this classification is primarily for convenience of presentation rather than something necessitated by the affordances themselves.

4.1 Characteristics of Finite Problem Spaces

A finite problem space is one in which the number of possible states for a solution process is bounded. For example, in a game of Tic-Tac-Toe, a reasonable formulation leads to a problem space with 5478 legal states. (This rules out states having more Os than Xs, and many others, but not duplicates modulo symmetries.) Although mathematical finiteness does not strictly imply enumerability by a computer in a reasonable amount of time, we will consider, in bringing liveness to solving these problems, than the space of states can be mechanically explored, fully, and that the time required to do this is taken care of in advance of the livesolver's encounter with the solving process. The livesolver's job might therefore be to demonstrate the selection of a good sequence of states from the problem's initial state to one of its goal states. The fact that the states have been precomputed before livesolving begins does not necessarily mean that the solution has been found in advance (although that is possible).

4.2 Drawing the Solution Path by Tracing over a Problem-Space Graph.

Our first form of livesolving, drawing and co-drawing solution paths, is a good fit for working in finite problem spaces. The finiteness means that it might be feasible to determine a state-space map for the problem in advance, and to offer it to the livesolver(s), as a substrate, for solving by drawing.

As an illustration, let's consider the Towers-of-Hanoi puzzle. Part of a CoSolve session tree for a 3-disk version of the puzzle is shown in Fig. 1. The initial state is at the root of the tree, and states created by making moves from it are shown underneath it. Now let's consider a version with 4 disks. The state space

for this problem has $3^4 = 81$ states. The state-space graph for this problem contains one node for each state and an edge between nodes n_i and n_j (corresponding to states s_i and s_j) provided there are operators that transform state s_i into s_j and s_j into s_i. Various layouts for this graph are possible. Figure 2 shows a layout constructed by assigning to each node three barycentric coordinates determined by assigning values to the presence of the various disks on each of the three pegs. The smallest disk is counted with weight 1; the next with weight 2; the third with weight 4; and the fourth with weight 8. The initial state of the puzzle is plotted at the lower-left vertex of the equilateral triangle, because the weight of all disks is on the first peg. Similarly, the other two vertices of the large triangle correspond to states in which all the disks are on the middle peg or all the disks are on the right-hand peg.

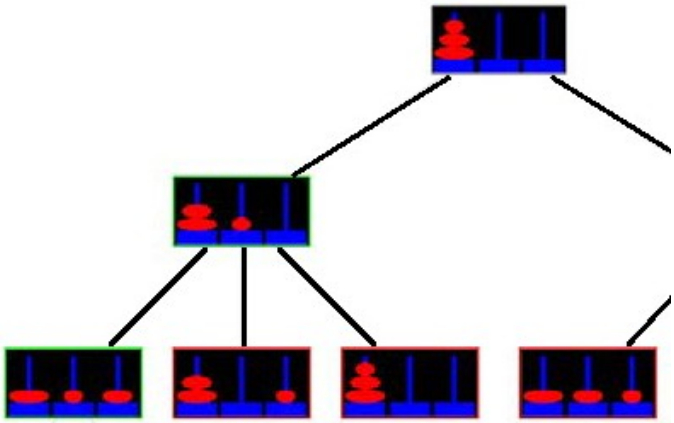


Figure 1. Portion of a CoSolve session tree for a 3-disk Towers of Hanoi puzzle. A solver generates one state at a time by making a selection of a node and an operator.

There are six operators in a standard Towers-of-Hanoi problem formulation. Each is of the form "Move the topmost disk from Peg i to Peg j." In Figure 1, the spatial direction in which a node transition is made corresponds directly to the operator involved. For example, from the initial state, moving the smallest disk from Peg 1 onto Peg 2 is shown by the line segment going from the node marked 0, diagonally up and to the right.

Livesolving with this state-space graph is tricky, because the solver would like to move directly from the lower-left vertex to the lower-right vertex, and yet no direct path exists. In fact, the shortest

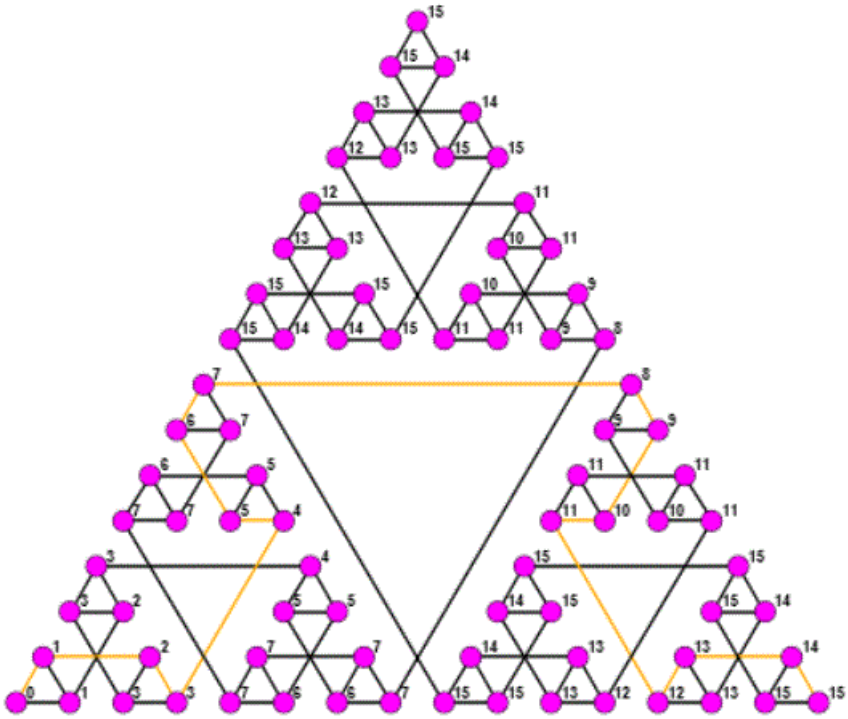


Figure 2. State-space graph for the 4-disk Towers-of-Hanoi puzzle used as a substrate for livesolving by drawing and co-drawing solution paths. A shortest solution path (the "golden path") is shown.

solution path involves some counterintuitive moves of disks from right to left now and then. Thus, there is an opportunity for livesolvers to develop skill at solving these puzzles and demonstrate the skill using this mode of livesolving, much as Rubik-cube experts like to show how quickly they can spin the faces from a random position and get all the marks on each face of the cube to have a uniform color.

One of the challenges for those who formulate problems is to come up with appropriate graph layouts (that we call problem-space visualizations) to permit this sort of solving by drawing. Finiteness of the problem space is a help, but that is not sufficient to make such layout easy. We discuss the challenges of layout further in the section on Livesolving in Infinite Problem Spaces.

4.3 "States-Alive" Liveness

In a puzzle such as the Towers of Hanoi, each state represents a static snapshot -- a possible configuration of disks on pegs that might reached in the course of attempting to solve. A static image is a suitable graphical representation for a solver working on the problem. On the other hand, there are problems that involve dynamic entities, such as the problem of designing an animation clip or creating a computer program that simulates bouncing balls on a table. With states alive, each state displayed in a solving session is backed by its own generative process, typically a running program operating in a separate thread of control. Particularly when the solver is comparing alternative states in order to select one for additional development, seeing the alternatives, "in action" can potentially improve the quality of the decision and have other benefits, such as making more apparent the properties of the states, not only for the solver, but for a livesolver's audience, if there is one.

One of the simplest examples of states-alive liveness occurs when the problem is to design an animated GIF image, and alternatives are displayed in their natural form -- as animated GIFs. An obvious potential drawback is that such displays, with multiple animated GIF images, can be annoying to users, because they tend to be visually busy and distracting, especially when the important decisions about which state to

expand, etc., have already been made. The answer should be that the states-alive facility comes with suitable controls for disabling and enabling the live displays that go with such states.

The use of separate threads of computation to support each of multiple states being shown simultaneously on the screen could easily consume available computation cycles or memory. Thus it is likely to work best when relatively few states will be displayed in this mode at a time. Finiteness of the problem space may help, at the very least, and policies or controls that limit the number of live states to a dynamically changing working set, would be appropriate. For example the live displays may be limited to one state and its immediate children, or to the states along one short path in the state-space graph.

4.4 Live Feedback During Solving

Crucial to maintaining a sense of flow to a user is providing timely feedback. In problem solving with CoSolve, the system generates a state display image each time the user creates a new state. With solving by drawing, the user should be receiving at least two forms of feedback: the display of the path as just drawn, plus the state visualization for the state most recently reached on the path. The time between user selection or specification of a new current state and its subsequent display is "state-display latency." By reducing this time to a few milliseconds, a system can better support a solver's experience of being in the flow. The finiteness of the problem space suggests that the state visualizations could all be precomputed and stored as properties of the nodes of the graph. This would allow showing these visualizations with low state-display latency, even if the user draws paths through the graph quickly.

In addition, feedback about the current node reached may include the display of values computed from the state: heuristic evaluation function values, for example. Even if these values or the state visualizations themselves are not precomputed, they may be computed lazily, scheduled for computation as soon as the node is added to a path under consideration. Yet the livesolver is not required to wait for them before exploring further. When computing resources are scarce, these visualizations should be computed strategically, to maximize the sense of orientation in the state space. For example, every other state along the path might be displayed if not every state can be.

5. LIVESOLVING IN INFINITE PROBLEM SPACES

5.1 Working with Parameters of Operators.

If the states of a problem have continuous variables in them, such as a temperature or speed, then the problem's state space is inherently infinite[1]. The space could be made finite by limiting such variables to a fixed set of values, but that may not always be appropriate. A consequence of having continuous variables in operator parameters is that the selection of appropriate parameter values can easily become a "flow killer" in a problem solving system. In order to avoid this, *live parameter tuning* should be supported. To use this, a user selects an operator and the system may respond by creating a new state using default values of the parameters. At the same time, controls such as sliders, dials, or image maps appear, as appropriate for each of the parameters required by the operator. The user then adjusts the parameter values, watching the new state update immediately to reflect the adjustments. A variation of this involves having the system present not just one new state, but a sample set of new states corresponding to a default set of values for each of the parameters. For example, an operator that adds a piece of furniture to a living-room layout may have three parameters: furniture type, x position, and y position. The x and y positions are values from a continuous range; the system can provide a sample set of (x, y) pairs that are generated as a cartesian product of a sample set for x with a sample set for y.

[1] Of course, real number are restricted in most computers to a finite set of rationals, but this is an artifact of implementation rather than problem modeling.

In addition to the use of default parameters and fixed sample parameter vectors, intelligent agents may take a solver's specification for a sampling distribution to use as the basis for parameter value sampling. An example of this is the specification of a "beam" of parameter values: a collection of tightly spaced values within a narrow subrange of the parameter's full allowable range.

5.2 Ancestor Modification

A variation of the live parameter adjustment affordance when creating new states (as described above) is to permit this in more cases. One of these is to change the parameters of an old state. If the state has no children, this is equivalent to what we have already discussed. If the state has children, then these children will generally be affected by any change to this or any ancestor. For each descendant of a modified state, any of several cases may apply: (a) no change, either because that influence of the change is blocked by some other ancestor, between the changed one and the descendant in question, (b) changes to the descendant propagate down, with no violations of operator preconditions on the path between the ancestor and the descendant, or (c) at some state between the ancestor and the descendant, the precondition for the operator originally used to create the state is now violated. In this third case, the logical update to the session tree is to remove the subtree whose root's operator precondition now is violated. However, that can be quite disruptive to the solving session, particularly if that deletion happens as an unintended accident of parameter adjustment. Flow may be better maintained by avoiding the possibility of such an accident. This can be done by automatically cloning any ancestor that has been selected for modification, along with all its descendants. If this policy is too expensive in terms of state duplication, a less costly cloning policy is to clone only when a descendant is about to be deleted, and then to remove the clone if the parameter is adjusted back in a way that removes the precondition violation. While ancestor modification complicates the implementation of the system, it offers substantial flexibility to the solver, allowing the "resurrection of otherwise unmodifiable states from the dead."

5.3 Path Drawing Without Precomputed Graphs

A problem space that is infinite cannot have its entire graph realized in advance or at any time. However, a portion of it may be computed in advance, and more of it may be computed during a solving session. The following challenges may arise:

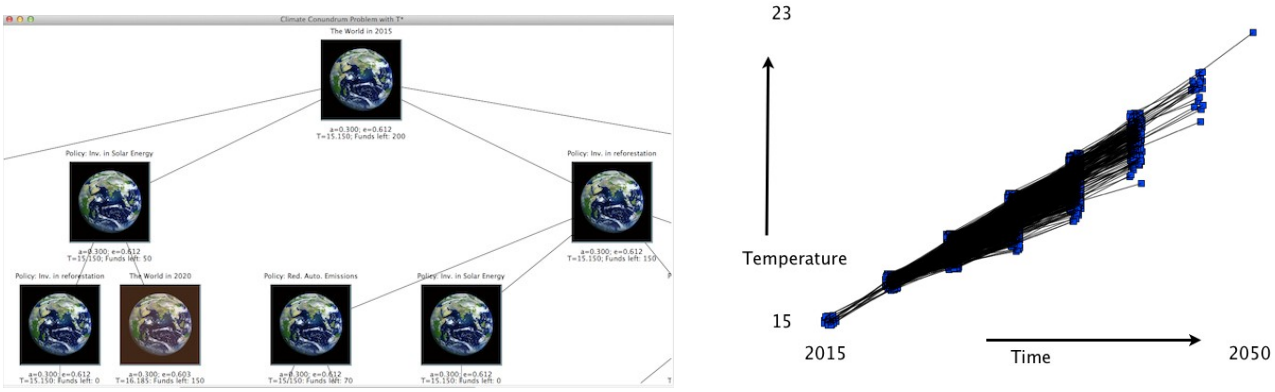
The time required for the system to apply an operator and produce a new state may cause enough latency to threaten the user's sense of flow. When this is the case, the sense of liveness can sometimes be saved by (a) suitable display of partial information, progressively as the new state is computed, or (b) covering the latency by posing questions to the user or offering more readily available alternatives.

The mapping of states to layout positions may cause collisions among realized states. While the Towers of Hanoi graph of Figure 2 has a clear minimum spacing between adjacent nodes, in an infinite problem space, this may be difficult, if not impossible to achieve. The fact that parameters of operators, and state variables themselves, may take on values in continuous ranges (e.g., real numbers between 0.0 and 1.0), means that states may differ by arbitrarily small differences in any state variable. If the designer of a problem-space layout wishes to have a minimum separation of nodes on the screen of 25 pixels, then two fairly similar states s_1 and s_3 might be positioned at x_1 = 100 pixels and x_2 = 125 pixels, respectively. If s_1 and s_3 differ simply because there is a state variable v whose value in s_1 is v_1 and whose value in s_3 is v_3, (where v_3 - v_1 is close to zero), there still may exist a state s_2 having its value of v half way between v_1 and v_3. Now where should the layout place the display of s_2? It could dynamically adjust the layout and double the spacing between s_1 and s_3, in order to place s_2 directly between them without violating the minimum spacing constraint. However, this raises the problem of layouts changing during the solving process, which may increase the cognitive load on the solver. If layout positions are not to change during the session, then

collisions on the screen are likely, as the realized portion of the infinite state space grows, and interface mechanisms are required to allow seeing and selecting any of the items that pile up on each other. One approach for this is to have multiple layouts or projections, with the user in charge of moving among different views of the same problem space. Another approach is to allow the layout to be locally dynamic, with techniques such as force layout, but overall mostly static.

Hand-in-hand with more sophisticated visualization techniques for the problem space are more sophisticated capabilities for the drawing tool supporting the livesolving. As the user draws a path through the 2D view of the problem space, the path must intelligently adapt to be a legal path through the actual problem space which may be of arbitrary dimension.

System policies supporting intelligent drawing assistance may include on-the-fly search, so that the state pointed to by the solver is automatically connected by a shortest path to the nearest landmark state, unless the distance from such a state exceeds a threshold set in the current drawing policy. Landmark states are defined by the solver by clicking on them and marking them during the course of solving. Another drawing policy that can be used is for the system to deal with drawing ambiguity (i.e., which state the user intends to move to next) by maintaining a set of nodes to represent the possible next state; the ambiguity may be gradually removed through additional user hints, or the



(a) (b)

Figure 3. Visualizations for the Climate Conundrum problem: (a) session tree, and (b) problem-space graph. The graph layout supports solvers' understanding of progress towards the goal, but it is poorly suited to path drawing, due to its densities of nodes and edges.

ambiguity may be permitted to persist, with the understanding that the system will eventually end up with multiple possible candidate solution paths.

Even if a problem-space visualization fails to provide a clear sense of direction or sense of proximity to any landmarks in the problem space, the ability to explore the space by drawing a path can still have appeal to livesolvers. At the very least, a graphical representation of the paths explored can provide a kind of visual session history whose topology (e.g., tree structure) indicates alternatives tried and points at which backtracking or branching occurred.

5.4 Effects of Dynamic Realization of the Problem Space.

Real-time solving with full state displays may be impractical, due to high state-display latency in complex problems. To keep the solver in the flow, it may be possible to use techniques such as (a) planning via homomorphic problem spaces (Pearl 1984), (b) low resolution visualizations on fully computed states, or (c)

sampling policies for visualizations – e.g., every other state on a path is visualized, or every *n*th is visualized, or a state is visualized to degree $f(k)$ if its importance exceeds k. Alternatively, visualizations may be computed lazily with different priorities, so that with enough time, a full path is shown in detail, but rapid work by a human will be supported by having the most-important visualization and state-computation work performed first.

6. SOCIAL LIVESOLVING

In this section, I discuss three of the aforementioned livesolving forms: synchronous co-solving, livesolving presentations, and driving an agent.

6.1 Synchronous Co-solving

Synchronous co-solving is collaborative solving activity in which a single representation of the solving session is shared among the solvers. When solvers work from geographically distant locations, the shared representation is accessed via client views that must be updated by the system. CoSolve's mechanism for these updates pushes notifications to solvers that changes have been made and then updates a solver's view if and when the solver clicks to request the update. Our new system Quetzal, on the other hand, pushes the new views themselves as soon as any team member makes a change. The potential exists for such changes to be disrupting to individual solvers, as, of course, too many cooks can spoil a broth. This is true for groupware in general (Gutwin et al 1995). But were synchronous co-solving not to be supported, close collaboration would be more difficult. We compensate, in Quetzal, for the potential disruption, by smoothly animating each of the changes to the session-tree view over a 2-second period. The potential disruption can also be reduced by clearly labeling incoming nodes as new and identifying their authors using avatars, names, or color coding.

If a solving environment gives to agents the status of "team member," then the potential for disruption is amplified. Agents can move too quickly to create new material in a solving session, and also, there is a high likelihood of their work being considered to be mostly garbage. Consequently, special affordances are required for managing the contributions of agents, such as running them in sandboxed subsessions. New or novice team members can also be managed with such devices.

6.2 Livesolving Presentations

Livesolving presentations share several elements with livecoding performances. First, the presence of an audience demands that there be an effect element of communication during the activity; the audience is expected to derive something from the experience, be it entertainment, education, or other information. Furthermore, livecoding is traditionally improvisational. In order for livesolving to also be improvisational, the tools must support extemporaneous decision making and short response times. In music, live-performance improvisation involves real-time commitment to musical elements that cannot be retracted (unlike studio recording session in which mistakes can be corrected). A jazz musician who hits a "wrong" note (wrong in the sense of unintentional or off-sounding) will typically turn this bug into a feature by developing it, recontextualizing it, and perhaps, gracefully abandoning it. A livesolver who arrives at a dead end in the problem space should likewise make the bug into a feature by building a narrative context around the dead-end that makes it a natural part of exploring the problem space. The audience might judge the performance either in terms of speed to solution or in terms of "narrative richness" or honesty in exposing the solvers' thought process. Computer support for this form of livesolving may include the display of graphical signals to the audience that new states are being explored (e.g., green) backtracking is occurring after reaching a dead-end (red) or that unforced backtracking is occurring (e.g., orange). Music livecoders often include colored code highlighting in their editors to reflect re-evaluation, syntax errors, or other conditions.

6.3 Driving an Agent

Driving an agent is listed here under social livesolving, but it may or may not involve relationships that feel social. The relationship between a driver and a car is not traditionally a social one. However, with automatic driving, the user and driving agent may have a relationship involving communication in spoken English, which may seem social. Other social relationships in driving exist, too: driver-navigator, driver-backseat driver. In the case of direction-finding, the navigator in a car may be a person or a Garmin GPS unit. In livesolving, agents may be making decisions about the order in which states of a problem space are explored, but a user driving that agent may steer the agent to alter the default decisions. When the operators take parameters, the choice of parameter values may be a large enough task that it can be shared by multiple drivers. One livesolver determines the type of furniture to be placed in the living room while another selects an (x,y) position for it by clicking or touching on the room plan. Computer support for driving an agent means offering interfaces to livesolvers appropriate to the particular kind of agent being driven.

7. DISCUSSION

In this section I address three questions. First is "Where did the forms of livesolving come from?". The second is "What can make solving live?" The reason to ask this is to offer a review of issues already raised, but in a slightly different light. Third is "How does livesolving compare with livecoding?" This question is prompted by the theme of the conference.

7.1 Where did the forms of livesolving come from?

After our group built two tools, TSTAR and CoSolve in order to facilitate deliberate, human problem solving, the question arose of what more we could do, through providing computer technology, in order to support human problem solving. Our two approaches involved (1) looking for ways to reduce latency in our current computer-mediated solving process, and (2) looking for alternative kinds of solving experiences that could be supported by the same classical theory and underlying support engine. The ideas took shape in the course of trying to answer the other two questions in this section.

7.2 What Can Make Solving Live?

Solving a problem, with help from others, including computers, can feel live when the solver experiences the kind of feedback expected when working among a group of engaged individuals. Having a conversation in which one's partner responds promptly and thoughtfully is one example. Working with a tool that performs its assistance without undue delay is another. The goal of our work is to enable solvers not only to experience liveness, but to achieve cognitive flow -- a state of mind in which mental transitions from one idea to another proceed at a rate that is fast enough to inhibit distractions from setting in, yet slow enough to permit comprehension and the development of good judgment. The seven forms of liveness presented above offer some means to this end. However, there are surely many other factors that affect the ability to achieve flow that crop up in reality such as the solver's freedom from external anxieties, the time available, his or her motivation to solve the problem at hand. We also require that the problem be formulated according to the classical theory, and the formulation process is a topic for future work.

7.3 Comparing Livesolving and Livecoding

Livesolving and livecoding, as presented here, are closely related. An obvious difference is that livecoding depends on code editing as its fundamental activity. Livesolving depends on problem-space exploration via operator application. From a theoretical perspective, they are equipotent. Livecoding in a Turing-complete language can result in an arbitrary computable function. Livesolving in a suitable formulated problem space can result in the composition of an arbitrary computational object, such as any computable function. Either

can be done in a performance context, and either can be used to generate music or any other computable process. The difference, then, is in the style of interaction that the solver or coder has with the system. The ways of thinking about the process, also, tend to differ. In a typical livecoding performance, there is not any goal to end up with a masterpiece of code at the end of a performance; rather the code evolves in order to generate the music required at each point in time, and at the end of the performance, the final version of the code may simply generate silence. In livesolving, however, there usually is an understood goal state, and there are criteria for success and failure related to it.

8. CONCLUSIONS

Problem solving using computers is an activity similar to programming in that it involves computational thinking, yet different from programming because of the structures and affordances typically provided. Exploiting the classical theory of problem solving, existing systems such as the University of Washington's CoSolve facility have shown how computational resources can be used by human solving teams to solve problems in ways that garner certain advantages. These systems, however, don't adequately support solvers in reaching their full potential as solvers. Here, I have presented seven forms of liveness in problem solving to help overcome those obstacles. Some of these forms involve how solvers specify the exploration of the problem space, while others involve the ways in which they interact with other solvers on the team or with an audience. Future work on livesolving will involve evaluation, comparisons to software design patterns, extensions to posing, the design of agents and agent interfaces, and enabling the scaling of facilities for large teams, complex problems, and more intelligent and fast computational support.

Acknowledgments

Thanks go to the CoSolve developers, especially Sandra Fan, Tyler Robison, and Rob Thompson. Thanks go also to Alan Blackwell, Sam Aaron, Luke Church, Advait Sarkar, Henri Angellino, Helmut Prendinger, and Tim Bell for supporting the development of these ideas during my 2013-2014 sabbatical. I'd also like to express my appreciation to the reviewers for their thoughtful comments.

REFERENCES

Bederson, Benjamin. 2004. "Interfaces for Staying in the Flow." *Ubiquity*. ACM. (September).

Blackwell, Alan and Collins, Nick. 2005. "The Programming Language as Musical Instrument." *Proc. PPIG05* (Psychology of Programming Interest Group), pp.120-130.

Fan, Sandra B., Robison, Tyler, and Tanimoto, Steven L., 2012. "CoSolve: A System for Engaging Users in Computer-supported Collaborative Problem Solving." *Proc. International Symposium on Visual Languages and Human-Centric Computing*, Innsbruck, Sept., pp.205-212.

Gutwin, Carl; Stark, Gwen; and Greenberg, Saul. 1995. "Support for Group Awareness in Educational Groupware." *Proc. Conference on Computer Supported Collaborative Learning*, pp. 147-156, Bloomington, Indiana, October 17-20, Lawrence Erlbaum Associates.

Kearne, Andruid; Webb, Andrew M.; Smith, Steven M.; Linder, Rhema; Lupfer, Nic; Qu, Yin; Moeller, Jon; and Damaraju, Sashikanth. 2014. "Using Metrics of Curation to Evaluate Information-Based Ideation." *ACM Transactions on Computer-Human Interaction*, Vol. 21, No. 3.

Mahyar, Narges and Tory, Menlanie. 2014. "Supporting Communication and Coordination in Collaborative Sensemaking." *IEEE Trans. Visualization and Computer Graphics*, Vol. 20, No. 12, pp.1633-1642.

Newell, Allen; Shaw, J.C.; and Simon, Herbert. 1959. "Report on a General Problem-Solving Program." *Proc. International Conference on Information Processing*. pp.256-264.

Newell, Allen; and Simon, Herbert. 1972. *Human Problem Solving*. Englewood-Cliffs, NJ: Prentice-Hall.

Nilsson, Nils J. 1971. *Problem Solving Methods in Artificial Intelligence*, New York: McGraw-Hill.

Pearl, Judea. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley.

Pu, Pearl, and Lalanne, Denis. 2002. "Design Visual Thinking Tools for Mixed Initiative Systems." *Proc. 7th International Conference on Intelligent User Interfaces* (IUI '02). ACM, New York, NY, USA, 119-126.

Russell, Daniel M.; Stefik, Mark J.;, Pirolli, Peter; and Cart, Stuart K. 1993. "The Cost Structure of Sensemaking." *Proc. INTERCHI '93*, pp.269-276.

Simon, Herbert A., and Newell, Allen, 1971. "Human Problem Solving: The State of the Theory in 1970." *American Psychologist*, 1971, pp.145-159.

Tanimoto, Steven L. 1990. "VIVA: A Visual Language for Image Processing." *Journal of Visual Languages and Computing*, (1), 2, pp.127-139.

Tanimoto, Steven L.; Robison, Tyler; and Fan, Sandra. 2009. "A Game-building Environment for Research in Collaborative Design." *Proc. International Symposium on Artificial Intelligence in Games*.

Tanimoto, Steven L. 2013. "A Perspective on the Evolution of Live Programming." *Proc. LIVE 2013, Workshop on Live Programming*, IEEE Computer Society.

Live Patch / Live Code

Charles Celeste HUTCHINS
University of Kent
celesteh@gmail.com

ABSTRACT

Modular synthesiser live-patching has gradually been accepted into the big-tent of live coding practice, due to a number of natural similarities to computer-based live coding. These similarities especially include flexibility, complexity and sudden stops. In my own performance of live-patching, I have sought out other areas of digital live coding practice to apply to the modular synthesiser. These include starting from a blank slate, showing my cables (sometimes with projection), graph changes and the use of conditional triggers to create audio events.

1. Introduction

The definition of live code has been gradually growing over the years of contemporary practice. In 2008, McLean struggled to name any non-digital live coding practice. (McLean 2008) Then, in February 2011, there was a very brief discussion on the TopLap email list as to whether live patching modular synthesisers was an analogue form of live coding (Rohrhuber 2001). By June of that year, Nick Collins was treating the question as settled in *Leonardo*, listing live electronics band Loud Objects as an example of live coding. (Collins 2011) He notes, when talking about live algorithms, 'Computers remain a primary touching point, but let us also keep in sight examples that center on human rule-making independent of digital computability.' (Collins 2011) Given this very broad potential pool of music-making activity, live coding has become extremely inclusive. Thor Magnusson referred to 'herding cats' in the title of one his papers, as a reference to this wide diversity. (Magnusson 2014) Indeed, I heard a joke recently that it would be faster to make a list of all the things that *aren't* live coding, rather than the things that are.

2. Synthesiser as a Live Code Platform

Given how open live coding has become, it may become useful to discuss points of commonality between divergent live coding practices. In many ways, live synthesiser patching seems different from live coding with a computer, given the vastly different interface. There is a greater amount of tactility, which, to me at least, feels closer to playing a traditional musical instrument than creating an algorithm. It is also worth noting that the synthesiser is specifically designed for music making, unlike a computer, and thus has a limit on the kinds of operations and calculations it can do. In this way, a synthesiser is both a chainsaw and an idea. (TopLap 2010)

Thus chainsaws vs ideas become less of a binary opposition and more a continuum. Tools, such as chainsaws, are perceived as extensions of the body of the user. (Cardinali et al. 2009) My experience of playing the tuba very much matches this. By contrast, my experience of text-based computer live coding feels abstract and unembodied. Live-patching the synthesiser lies in between. The afore-mentioned tactility does give it some feelings of embodiment. But plugging in cables and deciding how to route signals primarily involves abstract decision-making. Playing a patch, therefore, is chainsaw-like. Making the patch is idea-like.

2.1. Commonality

A modular synthesiser, therefore has many points of similarity with digital live coding practice. The specific design of a synthesiser for only music making may not parallel computer hardware, but the specificity certainly parallels the way some live coding languages have developed. While you can't use your synthesiser to check your email, this is also impossible in several live-coding languages and with early analogue or other non-digital computers. And, indeed, the modular synthesiser does have a general purpose element within the musical sphere. Common practice, when I was an

undergraduate, was to spend time in the studio, carefully making the perfect sound. This is also how I was taught to write code.

Historically, modular synthesisers do not seem to have been seen as live electronics systems to the same extent as were some particular pieces, such as *Runthrough* by David Behrman (1971), which is realised live with DIY electronics. The programme notes for that piece state,

> No special skills or training are helpful in turning knobs or shining flashlights, so whatever music can emerge from the equipment is as available to non-musicians as to musicians . . .. Things are going well when all the players have the sensation they are riding a sound they like in harmony together, and when each is appreciative of what the others are doing (Behrman 1971).

The piece is based entirely on discovery and has no set plan or written score (ibid). The piece-ness relies on the equipment. Scot Gresham-Lancaster describes pieces like this one as "direct use of electronics as a 'score' " (Gresham-Lancaster 1998).

Historical live electronic pieces like *Runthrough* do differ from live-patching in that a modular synthesiser is designed to be a more general purpose tool and its use does not imply a particular piece. Interestingly, however, some live coding platforms, like ixi lang, have been compared to graphic scores due to their use of space (Magnusson 2011) and thus do have a certain piece-ness about them. However, unlike ixi lang and *Runthrough*, understanding the interaction between synthesiser modules is not designed for ease of use for non-musicians. It is a specialist skill which does imply that expertise is possible. However, the idea of finding a sound and following it is similar to live electronics and all of these practices make use of improvisation. These are all points of commonality between live patching and digital live coding.

Additionally, the interface differences between modular synthesis and computer languages do not create as large a gap as it might at first seem. While synthesiser modules are not like text-based code, neither are all languages. Scheme Bricks, a live coding language by Dave Griffiths (2012) is an example of a coding language that relies very heavily on graphic elements rather than dense lines of text. A step closer to the synthesiser patching interface (and not uncoincidentally so) are the patching languages MAX/MSP and Pd. A version of one of these languages which had only a set and limited number of unit generators and control generators would not be logically different than a modular synthesiser. Even the model of making connections is the same.

2.2. Complexity

Both digital systems and analogue modular systems allow a user to create a graph of unit generators to generate sounds electronically. In both systems, the graphs and their control mechanisms can quickly grow in complexity, past the point where the player can easily understand what is happening. The late Click Nilson hints at this in his 2007 paper. In a footnote, he quotes a personal communication from Julian Rohrhuber, 'The main difficulty of live coding I find is to keep aware of the relation between text and sound - this is a balancing act between the ability to change and the ability to understand a certain situation' (Nilson 2007) While certainly live patching lacks text, there is some critical number of patch cables beyond which it becomes increasingly difficult to keep all the connections in mind. Patch cables turn into a kind of electronic spaghetti, where even the colour codes of my cables are not helpful in comprehending what is connected to what. I attempt to compensate for this by physically tracing my hands along the wires, but even that has its limits. Given the extreme flexibility for feedback and chaos with analogue signals, the routing can become intensely complex. Pd programmers are familiar with a similar effect of a visual spider web forming on screen.

This complexity also leads to a situation analogous to 'crashing', wherein the synthesiser suddenly stops making sounds in performance, in a way that can be difficult to predict. Experimenting with an oscillator's 'hard sync' switch during a set may lead to a sudden and interesting timbre change or a sudden silence. Similarly, long running LFOs can drop below a threshold where they unexpectedly cause things to fall silent. It is often not at all obvious what has caused the sound to stop. In situations where there are a large number of wires and multiple modes of interconnected sound-generation, it can be more beneficial during performance to treat a sudden, unexpected silence as a stopping point and start anew with new sounds after a brief pause to remove cables.

Although it would be impossible to exactly replicate a previous live patch in performance, I do sometimes try to repeat steps that have lead to a 'crash' in past situations, as they make a sudden and final-sounding way to end. I've seen many digital code-based live coders end sets through crashes and at least some of them do court disaster on purpose. Click Nilson has written of the desirability of a '"code out" ending of deliberate coded crash'. (Nilson 2007) And some live coding languages have embraced crashes as language functionality. ixi lang, for example, has a built-in 'suicide' command, which creates a percentage chance of crashing on each bar. (Magnusson 2012)

3. TopLap Influences

Many of the points of similarity above are inherent in live patching, due to a synthesiser's evolution from the analogue computer. However, the courting of crashes is one way in which my practice has been specifically influenced by the live code movement and the TopLap Manifesto in particular. (TopLap 2010)

3.1. Show us your screens

Like blank slate digital live coders who start positioned behind a laptop with a blank screen, I start with all of my patch cables hung around my neck. Thus we both experience a short pause from starting the set to starting making sound. I also emulate the 'show us your screens' directive, so I turn my synthesiser at an angle to the audience, so the majority of friendly experiencers can see my actions to at least some extent. This orientation is a compromise position such that my instrument is largely visible, but I am not turning my back on the audience. For many lines of sight, it is probably more visible than simply standing in front of the synthesiser. I have experimented also with using a web cam to project a video image of the synthesiser as I patch.

In order to make things more transparent, at the Live Code Festival in Karlsruhe in 2013, I paired this with a manually triggered set of slides of super-imposed text. These started with a description of what colour cable I intended to use for what kind of signal. Copying the Royal Conservatory of the Hague's coding system, I projected that I would be using blue for Control Voltages, black for Audio Voltages and red for Triggers. In addition, I projected the name and a photograph of the module that I was using when plugging into it for the first several connections I made.

While audience feedback was generally positive about these good intentions, it was difficult to find a web cam angle that was not blocked by my body as I used the instrument. Even more disappointing, the combined low quality of my web cam and of the projector meant that the different cable colours were not visible to people watching the video display. In low light situations, such as one often has on stage, black cables often become completely invisible. Although a better camera may solve colour issues, I have instead reconsidered what information I should be giving to audiences. Although McLean notes that programmers can take steps such as 'using a language with a simplified syntax, choosing highly descriptive variable names and so on', (McLean 2008) I have never seen a show-us-your-screens live coder project any explanations as to their syntax colouration or what various commands mean.

3.2. Graph Changes

When discussing approaches to digital live coding, Magnusson writes,

> The argument . . . is that a true 'liveness' requires that the performer is not simply manipulating pre-written code in real time, but is actually writing and modifying algorithms during the execution of the program. This is arguably a prerequisite of live coding, as simply running code and changing parameters in prewritten code is more comparable to operating buttons, sliders, and knobs on a screen interface or a MIDI controller. (Magnusson 2014)

This demand for algorithm modification also appears in Collins's live coding analogues, wherein he describes several examples of changing the algorithm mid-play. In one such example, specifically, 'a live-coding twist would be to perturb the algorithm halfway through'. (Collins 2011) Perhaps, however, it would be useful to take a step back and examine what is meant by 'algorithm' in an analogue context.

Plugging in cables creates a unit generator graph. The output of one circuit is connected to the input of another, eventually creating a complicated sound generation and control process. This is the creation of an algorithm, just as combining unit generators in a patching language creates an algorithm. However, playing this patch, by twiddling knobs on a synthesiser is exactly like operating knobs on a MIDI interface. Indeed, the physical interface is 'pre-written'. This is not to say that a physical interface is necessarily alien to live coding. Marije Baalman's live-coded piece, *Wezen-Gewording*, uses sensors attached to her hands, to capture the physical gesture of her specifically performative gestures and of her typing code. (Baalman 2014) However, setting something up and relying solely on parameter changes, while musically valid, is dubious under the heading of 'live coding.' Therefore, Baalman changes her code over the course of the piece. And in my practice, I do not rely on the algorithmic rules established at the start of the set. Instead, I 'perturb the algorithm' by setting up and switching to parallel signal paths.

Figure 1: *An image I projected in Karlsruhe.*

3.3. Decisions

Many computer programs involve decision making. This is not strictly necessary for audio processing, which only needs follow a signal graph. However, I seek to include decision making in my patches, in order to create stronger similarities to coding practice. My synthesiser includes a MOTM 700 voltage controlled router – a module that specifically does binary signal switching based on a comparison of two signals. However, although this module is an obvious choice for seeking to automate some decisions, this kind of behaviour is also found via triggers and envelope generators. When one signal changes, this changes another signal. This seems roughly analogous to an if statement. Because I often use both random generators and FM chaos, I cannot always predict when a trigger will occur. Thus the synthesiser picks up a degree of autonomy and behaves in an unpredictable way, increasing a distance between it and a chainsaw. The instrument may be a tool, but a patch is more like an expression of an idea. (TopLap 2010)

3.4. Scheduling

Many live coding languages offer the possibility of scheduling future events and some have argued that this is therefore a necessary component of live coding or live coding-related practice. In essence, this is a situation where the performer decides to invoke or stop a particular algorithm at a future time. A synthesiser user who wishes to mimic this behaviour has several options, including step sequencers or using LFOs to trigger future events or graph changes. As with deccisions, one can make the timing less predictable by using random generators or chaos. This is not something I tend to use either in my live coding or my live patching practice, but is certainly possible to implement.

4. Conclusion

While discussing live coding metaphorically, McLean wrote, 'We can think of live coding in the sense of working on electricity live, re-routing the flows of control . . ..' (McLean 2008) In modular synthesiser patching, this practice is not a metaphor. Flows of control are literally plugged and unplugged. Complex systems develop and are replaced by new complex systems. It's no longer necessary to retreat into our imaginations to experience live coding without a computer.

References

Baalman, Marije. 2014. "Wezen – Gewording." Performance; STEIM. https://www.marijebaalman.eu/?p=404.

Behrman, David. 1971. "Runthrough." In *Sonic Arts Union*. LP; Mainstream Records. http://www.ubu.com/sound/sau.html.

Cardinali, Lucilla, Francesca Frassinetti, Claudio Brozzoli, Christian Urquizar, Alice C Roy, and Alessandro Farnè. 2009. "Tool-Use Induces Morphological Updating of the Body Schema." *Current Biology* 19 (12): R478–R479.

Collins, Nick. 2011. "Live Coding of Consequence." *Leonardo* 44 (3): 207–211.

Gresham-Lancaster, Scot. 1998. "The Aesthetics and History of the Hub: the Effects of Changing Technology on Network Computer Music." *Leonardo Music Journal*: 39–44.

Griffiths, Dave. 2012. "Scheme Bricks." http://github.com/nebogeo/scheme-bricks.

Magnusson, Thor. 2011. "Algorithms as Scores: Coding Live Music." *Leonardo Music Journal* 21: 19–23.

———. 2014. "Herding Cats: Observing Live Coding in the Wild." *Computer Music Journal* 38 (1): 8–16.

———. 2012. "Ixi Lang." Accessed March 3. http://www.ixi-audio.net/ixilang/.

McLean, Alex. 2008. "Live Coding for Free." *Floss+Art. London: OpenMute*: 224–231.

Nilson, Click. 2007. "Live Coding Practice." In *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*, 112–117. ACM.

Rohrhuber, Julian. 2001. "[Livecode] Analogue Live Coding?" http://lists.lurk.org/mailman/private/livecode/2011-February/001176.html.

TopLap. 2010. "ManifestoDraft." http://toplap.org/index.php/ManifestoDraft.

SuperCopair: Collaborative Live Coding on SuperCollider through the cloud

Antonio Deusany de Carvalho Junior
Universidade de São Paulo
dj@ime.usp.br

Sang Won Lee
University of Michigan
snaglee@umich.edu

Georg Essl
University of Michigan
gessl@umich.edu

ABSTRACT

In this work we present the SuperCopair package, which is a new way to integrate cloud computing into a collaborative live coding scenario with minimum efforts in the setup. This package, created in Coffee Script for Atom.io, is developed to interact with SuperCollider and provide opportunities for the crowd of online live coders to collaborate remotely on distributed performances. Additionally, the package provides the advantages of cloud services offered by Pusher. Users can share code and evaluate lines or selected portions of code on computers connected to the same session, either at the same place and/or remotely. The package can be used for remote performances or rehearsal purposes with just an Internet connection to share code and sounds. In addition, users can take advantage of code sharing to teach SuperCollider online or fix bugs in the algorithm.

1. Introduction

Playing in a live coding ensemble often invites the utilization of network capability. Exchanging data over the network facilitates collaboration by supporting communication, code sharing, and clock synchronization among musicians. These kinds of functions require live coding musicians to develop additional extensions to their live coding environments. Due to the diversity of the live coding environment and the collaboration strategies settled for performances, implementing such a function has been tailored to meet some ensemble's requirements. In addition, networking among machines often requires additional configuration and setup, for example, connecting to specific machines using an IP address. In order to overcome these constraints, our goal is to realize a platform that facilitates the collaboration among live coders with minimal efforts of configuration, utilizing cloud computing.

There are many advantages to replacing a traditional server-client system with a cloud server. First of all, the collaboration scenario could be extended to the live coding ensemble whose members are distributed over different locations, enabling a networked live coding performance. Not only does this enable telematic performances, but it will also make a live coding session take place in a distributed manner, which will change the rehearsal process of live coding ensembles, whether it is remote or co-located. In addition, using the cloud server minimizes the amount of setup needed for networking as long as each computer is connected to the Internet. The amount of setup required is equivalent to creating a shared document in a Google Drive.

To that end, we present **SuperCopair**, a package for the Atom text editor, that offers code sharing and remote execution over the Internet. In this paper, we introduce the background that the idea is built upon, articulate our motivations for using cloud computing, and describe the implementation of the system. Finally, we suggest multitudes of new performance practices enabled by the system.

2. Networked collaborative live coding

Networked collaboration in live coding was present from the inception of live coding where multiple machines are clock-synchronized exchanging TCP/IP network messages (Collins et al. 2003). Many live coding ensembles also utilize network capability to share data and communicate within the ensemble (Collins et al. 2003; Rohrhuber et al. 2007; Brown and Sorensen 2007; Wilson et al. 2014; Ogborn 2014a). However, most of the time, they are based on the local network communication and not designed for remote collaboration attempted in the tradition of Network Music. Remotely connected music systems not only create a number of unique challenges and aesthetic opportunity as a performance in public, but also provide a base for musicians in different localizations to collaborate over the network synchronously.

Telepresence performance recently emerged as a new collaboration practice in live coding. Swift, Gardner, and Sorensen (2014) conducted networked performance between two live coders located in Germany and United States using an SSH server located in Australia. Extramuros, a language-neutral shared-buffer, is a web-browser based system to share code among connected machines (Ogborn 2014b). Gibber, a live coding environment on a web browser, supports collaborative editing and remote execution similar to Google Docs (Roberts and Kuchera-Morin 2012). Commodity softwares (such as Google Docs, CollabEdit, or DropBox) can be useful for remote collaboration and are convenient since users do not need to perform any configuration. However, these systems were either not designed or offer at best limited support for remote music performances.

3. Designing a collaborative solution

Although in the past it was difficult to think of thousands of people interacting at the same time on a musical system, the actual situation is in the quest of the best way to use the services and technologies offered day after day. For the last several years, we have witnessed an impressive advancement in the quality of services of cloud computing systems. Cloud servers distributed worldwide are connected through fiber optic broadband, and its cloud services have many advantages for computer music and collaborative works. We are taking some benefits from these characteristics in this study.

The cloud computing suggests itself as the next logical step in network capability, ready to be used for musical applications and performances. 'CloudOrch' is one of the first attempts to utilize the advantages of cloud computing in musical ways (Hindle 2014). The idea was to deploy virtual machines for client and server users, create websockets for intercommunication, and stream audio from cloud instruments to both desktop computers and mobile devices using web browsers. The author dubbed his idea 'a sound card in the cloud' and presented latency results from 100 to 200~ms between the Cybera cloud and the University of Alberta, Canada using the HTTP protocol. Using cloud computing as opposed to using server-client option has multiple advantages. Once an online instance is configured, a user can connect to or disconnect from the cloud at any time and can share the same resources within a group of connected users and take advantage of the reliable and scalable resources provided. Indeed, this solution can be useful for live coding network music.

The authors had already discussed models and opportunities for networked live coding on a past work (Lee and Essl 2014). The paper introduces diverse approaches in networked collaboration in live coding in terms of the type of data shared: code sharing, clock synchronization, chat communication, shared control and serializable data (such as audio). We draw upon ideas of existing systems that enable 'code sharing' and 'remote execution' re-rendering program state by evaluating code fragments in both the local machine and the remote machines in many live coding environments and extensions (Brown and Sorensen 2007; Rohruber and Campo 2011; McKinney 2014; Roberts and Kuchera-Morin 2012). These systems are similar in the sense that they need a separate server installed and configured by their users.

It is a general trend to have software application distributed over the Internet. Cloud computing is the central tool to realize the distributed softwares. However, the use of cloud computing is underdeveloped in computer music and we believe that it is the next logical step to put computer music applications in the cloud as a mean to realizing network music. The cloud computing provides a set of services that are beneficial to scale the computer music performance. For example, we can imagine a small scale ensemble co-located in the performance space, in which case the cloud computing will create a virtual machine based on the data center nearby the performance location. In the opposite case where large-scale participants are expected on a collaboration session, the cloud service will easily scale its computational power, network traffic bandwidth and storage space automatically to meet the spontaneous needs, although it will have some monetary cost.

In terms of network latency, we have achieved, an average round-trip time of 230~ms between Brazil and United States, and a minimum of 166~ms (Carvalho Junior, Queiroz, and Essl 2015). These tests were done using mobile devices connected to Pusher, a cloud service described below, but it can be extended to almost any device connected to the Internet. The strategy of transferring code (textual data) and re-rendering the program state remotely instead of streaming audio makes the latency less critical particularly for the scenario of live coding. However, it should be noted that the sound outcome from local machines and remote machines may not have exactly the same sound for many reasons (e.g., latency, randomness, asynchronous clock, packet loss).

The use of cloud computing resources became easier after the introduction of some cloud services that create an abstraction of the cloud computing set up and offer simple APIs for users, as we can find on Pusher. Pusher offers a cloud computing service that delivers messages through web sockets and HTTP streaming, and support of the HTTP Keep-Alive feature. The service has a free plan with some daily limitations such as 100,000 messages and a maximum of 20 different clients connected. Another limitation of the free plan is that we can only exchange messages through the US-East cluster server situated in Northern Virginia. The paid plans are more flexible and they make possible to have more users connected, send more messages, and use other clusters. In spite of that flexibility, all plans have a hard limit of 10

messages per second for each user. This limitation is due to the overhead of message distribution among a thousand users, but it really suits most needs to common use cases. Every message has a size limit of 10 kilobytes, but one can request an upgrade if larger messages are needed. Although it has limitations, we do not need to set up any cloud instance to benefit from the cloud computing infrastructure provided by this service.

The service works with push notifications, so every message sent is going to be received by all devices assigned to the same channel. A SuperCollider programmer can evaluate the whole code, a selected part, or just a line using keyboard shortcuts. SuperCollider programming language supports real time audio synthesis and is used extensively by live coders. These characteristics turn the language very suitable to be used with a push notification cloud service.

4. SuperCopair

The solution presented in this paper was created as a package to the Atom.io IDE. Defined as 'a hackable text editor for the 21st Century' on its site[1], Atom is a text editor created using web technologies and has its development powered by the github community. This IDE has numerous packages for many programming languages and presents some solutions for coding, debugging, and managing projects. Atom packages are programmed in CoffeeScript, which is a programming language that can easily be converted to Javascript and can also integrate its libraries. The developers can install Atom packages to enable various functionalities in the IDE such as: communicate through chats, use auto-complete in certain programming language syntax, interact with desktop and web applications, integrate with the terminal command line, and have many options based on other packages. These features have motivated the development of SuperCopair package for Atom.

SuperCopair is based on two Atom packages: atom-supercollider and atom-pair. The first package turns Atom.io as an alternative SuperCollider IDE and permits users to openly communicate locally with SuperCollider audio server through OSC in the same way we can do on SC-IDE. Moreover, the users can take advantage of other Atom packages additionally to quarks packages. The latter package is used for pair programing through the Internet. The atom-pair package is based on Pusher cloud service and its default configuration is based on the community free plan, but a user can modify the settings and use the user's own keys within the personal free or paid plan.We decided to merge both packages to add new features for collaborative live coding, and finally had dubbed it the SuperCopair package.

The main idea is that all participants have the opportunity to evolved into a collaborative performance.

The IDEs for SuperCollider have, by default, shortcuts to evaluate a line, a block, and to stop all sound process that is running. In addition to these options, the SuperCopair package includes methods and shortcuts to broadcast these events and execute them on all users connected at the same pairing session. Through the shortcuts, one can decide to evaluate selected code either only in the local machine, or in all computers of the session. One can also turn on and off a broadcast alert option in the settings in order to be asked or not before evaluating every broadcast event sent by another user in the same session. This allows each individual has control over which code to be evaluated in the local machine.

The broadcast events are diffused through the cloud service and are expected to be evaluated as soon as each device receives the event message. The message includes the code to be evaluated and the user identification. A representation of a session using SuperCopair package is shown at Figure 1.

4.1. Package installation and use

One can have the package properly installed and ready to be used in two different ways. Via the Settings View in Atom.io, the user can search and install the package. It is also possible to install the shell commands during Atom.io setup and use the line below to install the package:

```
apm install supercopair
```

After the installation, the user needs to start a new session before inviting others. An instructional step-by-step setup is presented on the package page. Then one can get initiate a performance by opening a SuperCollider file and starting a pairing session. The session ID string needs to be shared with collaborators so they can use the string to join the same session.

The shared session ID is based on the channel created at the cloud service and it contains application keys. It is recommended to change the app keys after each session. As the keys are linked to the account used during the performance, other participants can use the keys for other activities and the number of events will deducted from the main account.

[1] Atom.io website: http://atom.io/

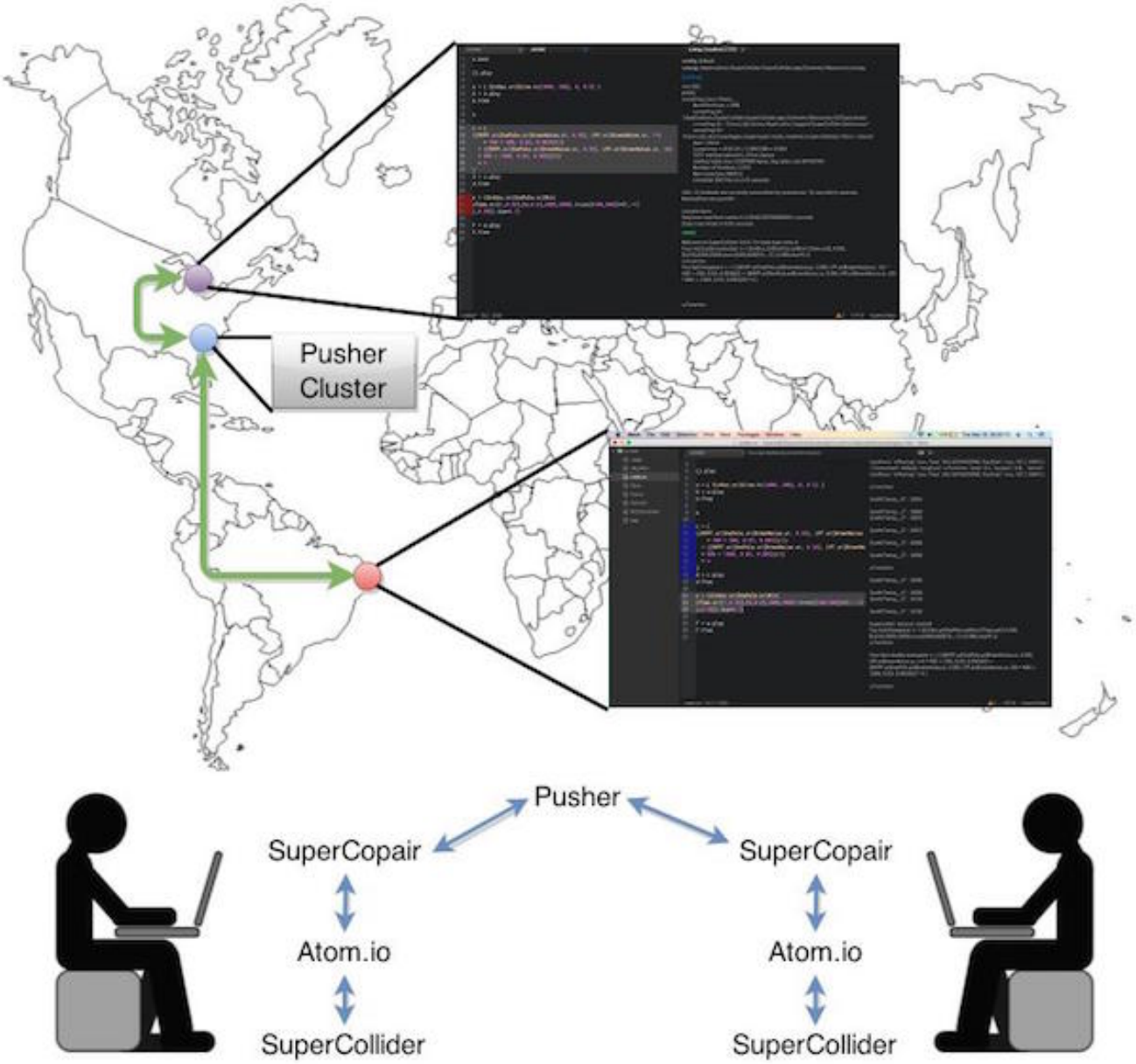


Figure 1: *Example of a session using SuperCopair and the architecture of interactions. The central bullet is the localization of the cluster server in Northern Virginia, and the other bullets represents users connected to the cloud server. The screen has the code on the left and SuperCollider post window on the right. The architecture presents only two users but it can be replicated to many, with diffusion on Pusher cloud service.*

The users who joins later will see the most recent version of the shared code. The users are identified by different color markers, and they can identify what each member is writing on the file based in these colors. A pop up provides information about users joining or leaving the session. Furthermore, a message including the identification of the user and also the code evaluated is shown at SuperCollider post window right after each broadcast event is evaluated. In case the broadcast alert option is on, a dialog will appear whenever an event message is received from another and ask if the user would accept or reject the code evaluation. The alert dialog will have the sender's id and the code sent via broadcast. When a live coder leaves the session, he or she can keep the most recent updated file to save or edit offline.

The delay achieved on the free plan depends on the distance between every member and the US East Coast cloud server. This free plan from Pusher cloud service allow 20 different clients per day on the same session and 100 thousand messages per day, however we have higher limits on paid plans. The session will stop after reaching the daily limit of messages for all plans, but the daily quota is reset after midnight UTC. It is important to keep these details in mind while facing any problem or high latency. The user may want to try a paid plan to have distributed data center, clients more than 20, and larger sized messages.

4.1.1. Shortcuts

The users have some special shortcuts depending on the operating system, and they are related to these specific functions:

- Compile library (open post window if needed)
- Clear post window
- Evaluate selection or current line locally
- Panic ! Stop all music
- Broadcast a code evaluation to everyone (including oneself) in the session
- Broadcast a code evaluation to others (excluding oneself).
- Broadcast stop command to everyone (including oneself) in the session
- Broadcast stop command to others (excluding oneself).

These shortcuts can be used to interact with other participants during a performance. The broadcast methods will only be shared with users on the same session, so it is also possible to create multiple sessions and interact with different crowd teams at the same time using a distinct Atom.io window on the same computer.

4.1.2. Practices and performances

The authors attempted to test the application multiple times in the co-located setup and also tried remote sessions by recruiting SuperCollider users. From one of our practices, there were live coders from Ann Arbor, MI, and San Francisco, CA, in U.S., and also São Paulo, SP, and Fortaleza, CE, in Brazil. During the session, participants (including the author) shared the session ID using an Internet Relay Chat (IRC) channel and we had a brief discussion about the package before starting to code. Some users reported that it could be dangerous to use headphones if we had switched off the alert for broadcast events, because some user may send a louder code to be synthesized. In the end, the session is successfully carried out without many problems and we are on the improvement of the package based on comments and suggestions from the participants. Atom.io installation was cumbersome for some users of Linux due to recompilation requirements at some distributions, and a step-by-step guide is under construction. Additionally, Mac users need the newest versions of the system in order to install Atom.io, but the users can also use a virtual machine with Linux and get rid of this limitation.

The practice addressed above is to simulate a networked live coding performance where multiple remote performers join a SuperCopair session from each one's our location, that may not be the concert space. In the local concert space where the audience is, a laptop connected in the session is placed without a performer on stage. Each performer would evaluate code in broadcast mode so that the computer on the stage will generate the collection of sound that remote live coders make via the Pusher. At this performance, the spectators at the local concert hall may have a wrong impression about the performance in the beginning because there is no one on stage except the laptop. As the audience will watch the video projection of the laptop screen, they will understand the main idea of remote performers live coding locally, from the multiple concurrent edits and some live coders' explanatory comments shown on the editor.

5. Discussion and conclusions

Here we present advantages and opportunities enabled with SuperCopair in network live coding: remote collaboration, telematic performance, and crowd-scale networked performance. One interesting advantage of the application is that it supports remote pair programming. We have witnessed that users can teach the basics of a language each other or help in bug fixing using online pair programming. Beginners can invite someone from anywhere in the world for a pairing session and start coding together on the same file and also synthesize the same code in real time while learning some tricks about live coding. Additionally to the forums and mailing lists, one can invite professionals to help on fixing algorithms for audio synthesis and have another kind of experience like pair or group programming on the same code to come up with a solution collaboratively. This package supports only SuperCollider namespace, but in the near future we can have similar packages for Csound, Chuck, or any other computer music programming language.

SuperCopair also offers novel forms of networked performances based on message streaming. The work of Damião and Schiavoni (2014) is a recent attempt to use network technologies in order to share contents among computers for a musical performance. The authors send objects and strings through UDP using OSC and can control other machines running the same external on Pure Data. They opted for UDP and Multicast to get better results on message distribution if compared to TCP and broadcast, which usually include three way handshaking and relay server. Although their decision has been based on solid arguments, our solution takes advantages of HTTP persistent connections using a single TCP connection to send and receive messages, and we also bring a reliable option for broadcast delivery using cloud services capabilities.

The package presented in this paper can be extended as an alternative for other networked live coding APIs. One can cite the Republic quark package that is used to create synchronized network performances. and the extramuros (Ogborn 2014b), a system for network interaction through sharing buffers of any kind of language. The last solution needs to be configured depending on the language andit does not present any easy way to share control (e.g. stop synthesis on SuperCollider) at the moment. Another constraint of both solutions is the need to create and configure a server on one computer to receive connections from clients, and additionally it would be necessary to open network ports or change firewall settings before starting any interaction with remote users.

SuperCopair realizes accessible configuration of network music performances, utilizing the cloud services. There is no need to configure a server or manage any network setting, e.g. routing, firewall, and port. We expect that even inexperienced users will be able to create a session with lots of people. As long as one can install the Atom editor and the SuperCopair package, the creation and participation at remote performances become an easy sequence of one or two shortcuts. Eventually, SuperCopair will simplify the steps to create a collaborative performance and remote rehearsals, and be used by people without network knowledge.

6. References

Brown, Andrew R, and Andrew C Sorensen. 2007. "Aa-Cell in Practice: an Approach to Musical Live Coding." In *Proceedings of the International Computer Music Conference*, 292–299. International Computer Music Association.

Carvalho Junior, Antonio Deusany de, Marcelo Queiroz, and Georg Essl. 2015. "Computer Music Through the Cloud: Evaluating a Cloud Service for Collaborative Computer Music Applications." In *International Computer Music Conference*.

Collins, Nick, Alex. McLean, Julian. Rohrhuber, and Adrian. Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (03): 321–330.

Damião, André, and Flávio Luiz Schiavoni. 2014. "Streaming Objects and Strings." *Live Coding and Collaboration Symposium*.

Hindle, Abram. 2014. "CloudOrch: a Portable SoundCard in the Cloud." In *New Interfaces for Musical Expression*, 277–280.

Lee, Sang Won, and Georg Essl. 2014. "Models and Opportunities for Networked Live Coding." *Live Coding and Collaboration Symposium*.

McKinney, Chad. 2014. "Quick Live Coding Collaboration in the Web Browser." In *Proceedings of New Interfaces for Musical Expression (NIME)*. London, United Kingdom.

Ogborn, David. 2014a. "Live Coding in a Scalable, Participatory Laptop Orchestra." *Computer Music Journal* 38 (1): 17–30.

———. 2014b. "Extramuros." https://github.com/d0kt0r0/extramuros.

Roberts, C., and J.A. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference (ICMC)*. Ljubljana, Slovenia.

Rohrhuber, Julian, Alberto de Campo, Renate Wieser, Jan-Kees van Kampen, Echo Ho, and Hannes Hölzl. 2007. "Purloined Letters and Distributed Persons." In *Music in the Global Village Conference (Budapest)*.

Rohruber, J., and A. de Campo. 2011. "The Republic Quark." https://github.com/supercollider-quarks/Republic.

Swift, Ben, Henry Gardner, and Andrew Sorensen. 2014. "Networked Livecoding at VL/HCC 2013." In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, 221–222. IEEE.

Wilson, Scott, Norah Lorway, Rosalyn Coull, Konstantinos Vasilakos, and Tim Moyers. 2014. "Free as in BEER: Some Explorations into Structured Improvisation Using Networked Live-Coding Systems." *Computer Music Journal* 38 (1): 54–64.

very long cat: zero-latency network music with live coding

David Ogborn
McMaster University
ogbornd@mcmaster.ca

Shawn Mativetsky
CIRMMT, Schulich School of Music, McGill University
shawn.mativetsky@mcgill.ca

ABSTRACT

very long cat are a new network music duo combining tabla and live coding, rehearsing and performing via the Internet, and employing an eclectic range of techniques and technologies. The specific structure of the ensemble's network music setup, with one musician live coding and monitoring their own performance with a calibrated delay, allows both musicians to experience each other's performances as synchronized. This poster focuses on the evolving technical configuration of this hybrid ensemble, with an emphasis on the constraints imposed by the insistence on "zero latency" in a live coding ensemble (some sound transformations are not feasible, and others are implemented in a characteristic way).

1. Introduction

very long cat are a new network music duo combining tabla and live coding (Collins et al. 2003), rehearsing and performing via the Internet, and employing an eclectic range of techniques and technologies. The specific structure of the ensemble's network music setup, with one musician live coding and monitoring their own performance with a calibrated delay, allows both musicians to experience each other's performances as synchronized. This poster focuses on the evolving technical configuration of this hybrid ensemble, with an emphasis on the constraints imposed by the insistence on "zero latency" in terms of the sound results (some sound transformations are not feasible, and others are implemented in a characteristic way).

2. Tabla Traditions

The tabla is the most popular percussion instrument in India and is especially prevalent in the northern regions of the country, as well as in Pakistan. The tabla is an integral part of North Indian (Hindustani) classical music, where it can be performed solo, in kathak dance accompaniment, or to accompany melodic instruments (sitar, voice, violin, etc.). There are six different stylistic schools of playing (gharanas), many varied sounds and techniques, and a repertoire made up of a multitude of cyclic and cadential compositional forms (Courtney 2000). The second author of this poster performs tabla in the style of the Benares gharana.

In the Benares gharana, the art of solo tabla performance is greatly respected and very highly developed, with more than twenty forms of composition in use, many of them unique to this tradition. The compositions can be broken down into two main types: theme-and-variation forms which require spontaneous improvisation by the performer, and fixed, composed pieces. In addition, there is a well-defined procedure and structure used for joining the various types of compositions together to form a logical and pleasing performance. Repertoire and performance practices are handed down from generation to generation through oral tradition (Shepherd). The North Indian tabla is connected with live coding through the fact that both traditions, despite their very different age, are centred around the art of improvisation.

3. A 'Zero Latency' Network Music Configuration

It is in the nature of transmitting audio signals from one point to another that it takes time for them to arrive and be decoded/transcoded. Our everyday audio computing infrastructure exacerbates this latency through multiple layers of transmission and buffering. Latency is thus a central and perennial challenge of network music situations (Carôt and Werner 2007; Whalley 2012) and has been the focus of a number of distinct research streams.

One stream of network music research seeks to characterize the effect of latency on network-distributed musical performances. For example, a recent rhythmic clapping study (C. Chafe, Cáceres, and Gurevich 2010) involved pairs of rhythmic

clapping performers separated physically and by carefully controlled delays ranging from 3 to 78 ms. The study revealed the existence of four distinct impacts of bi-directional delay/latency on ongoing musical tempo. Between 10 and 21 ms, tempo was stable, with a tendency to accelerate at lower latencies, a tendency to decelerate at higher latencies, and a marked deterioration of synchronization at even greater latencies (66 ms and higher).

Other network music research "accepts" the inevitability of latency and constructs new performance spaces and expectations around the specific temporal characteristics of the network (Tanaka 2006; J.-P. Cáceres and Renaud 2008). Some systems even extend this acceptance of latency as far as measuring it and projecting it onto other parameters: The peerSynth software reduces the amplitude of sonic contributions from particular network nodes as the latency to that node increases, just as in acoustic transmission longer delays are produced by greater distances which also simultaneously produced reduced amplitudes (Stelkens 2003). The Public Sound Objects system uses measurements of the latency to adjust the overall tempo, motivated by the insight that longer latencies are more sustainable at lower tempi (Barbosa, Cardoso, and Geiger 2005).

In the very long cat ensemble, we take advantage of the fact that one of us is live coding to sidestep the problem of latency. Specifically, our configuration results in sonic events being aligned (synchronized) in the same way at both ends of the ensemble. We call this absolute alignment of the audio layer "zero latency" although it should be recognized that in terms of non-sonic elements of the overall performance situation, such as the interplay of actions and decisions between the two performers, latency still exists. Additionally, instantaneous transformations of the live tabla sound must be avoided if the "zero latency" audio synchronization is to be upheld. Even so, the experience of producing tightly synchronized ensemble rhythms in apparent defiance of network latency and geographic separation is musically rewarding, and represents a live-coding-specific "road less travelled" in network music practice.

Both performers in very long cat use jacktrip (Caceres and Chafe 2009) to send two channels of full-resolution, uncompressed audio to each other. The live coding performer uses jacktrip to transmit the sonic result of his live coding, while the tabla player transmits the signal captured by two microphones, one close to each drum. After much experimentation with a number of high-end microphones, AKG C414 microphones were chosen, as they reliably capture the tabla's sound across a wide range of timbres. Each player monitors the signal they receive from the other player as soon as possible.

The tabla player simply plays along in sync with what he hears. However, the live coding performer monitors the sounding result of their live coding at a small, calibrated delay equal to approximately the time it would take for network audio via jacktrip to do a complete round trip between the two locations. It is as if the local monitoring of the live coding were to line up in time with itself as transmitted and then transmitted back. Typically, this delay is 111 milliseconds, a figure arrived at by playing identifiable beats together and adjusting the delay until the synchronization is aurally pleasing. Essentially, this technique exploits the gestural ambivalence of live coding (intentions and results are not, as a rule, tightly synchronized) to overcome the fundamental problem of latency in network music. The live coding performer does not need to hear the results of their work immediately, and so can "afford" to monitor their work at a small delay.

During rehearsal and telematic production situations, our bi-directional stereo jacktrip connection is solid enough that dropouts (brief periods of missing audio data) are quite infrequent and do not disturb our co-performance relationship. However, these infrequent dropouts are frequent enough to damage recordings of these sessions. For this reason, we record 4 channels of audio at both ends of the situation: each performer records the signals they are producing/transmitting as well as the signals they are receiving from the other player. The brief dropouts are only found in the received signals, so a "perfect" recording can be reconstructed by combining the "local" signal from each player. In each 4-channel recording, the presence of the received signal acts as an indication and record of how time was aligned at each performance site.

Both players use a Max patch that has been developed and maintained to provide for the requisite delay in monitoring (at the live coding end of the ensemble) and the 4-channel audio recording. This Max patch, the live coding in SuperCollider, and jacktrip are thus connected as follows:

4. Live Coding around Latency with Live Electronics

The live coding in very long cat is performed in the SuperCollider language (McCartney 1998). In general, the live coding consists alternately of pattern constructions in SuperCollider's standard pattern classes, or "modular synthesis" style notation of live electronics using the JITlib notations advanced by Julian Rohrhuber and others.

A central technique is the "live electronic" transformation of the signal from the live tabla player. For example, the envelope of the (stereo) live tabla signal can be extracted and then re-applied to synthetic sounds that are specified as the result of oscillators, low-frequency oscillators, etc. The ensemble insists on only applying those techniques that allow both players to experience the sonic results as synchronized in the same way. An immediate consequence of this is

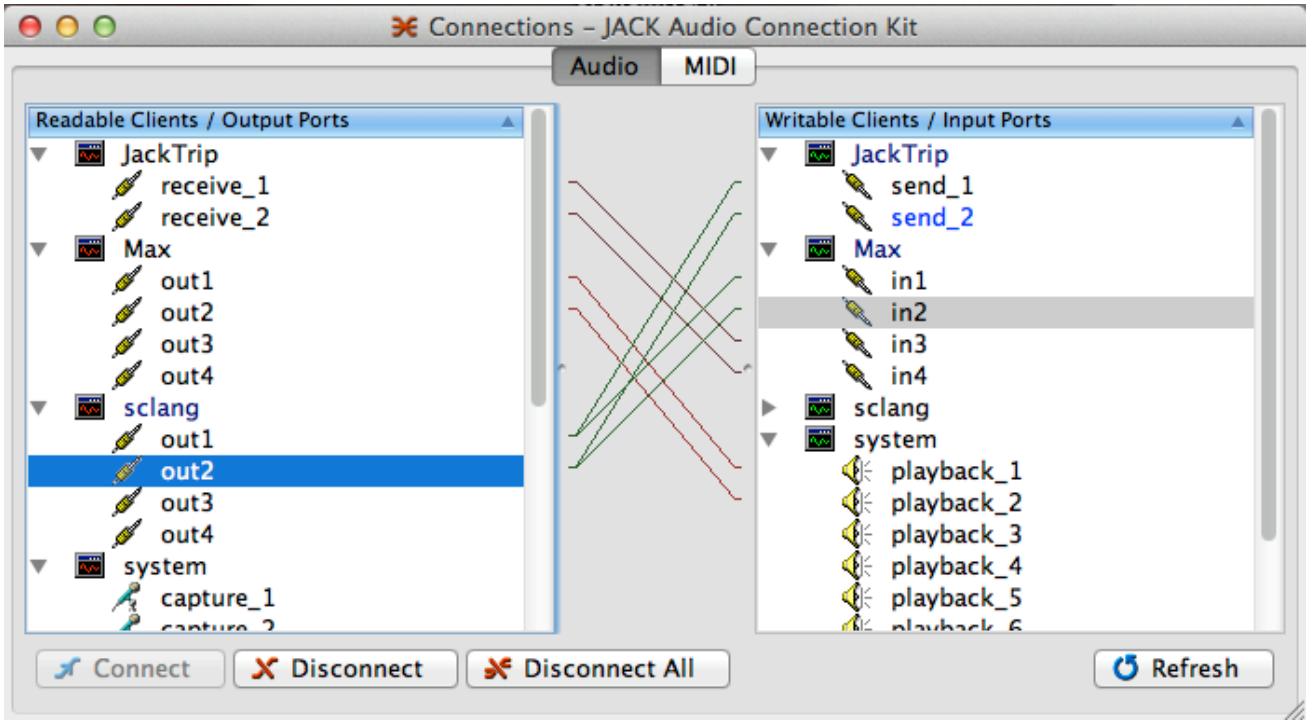


Figure 1: *Jack connections on the live coding end of the ensemble*

that no immediate transformations of the tabla sound can be used, as the result of such transformations would not be immediately audible to the tabla player at the other end of the jacktrip connection.

Instead, the transformations of the live tabla sound must always take effect at a delay. At a minimum, if the zero-latency requirement is upheld, this delay must be equal to the round trip delay of the jacktrip transmission. In practice, however, we tend to use delay times that are lined up to metric structures (like a beat at a normal tempo, or a bar, or longer).

In the SuperCollider live coding, the delay time has to be modified from the straightforward intended delay time on account of the latency of the network music situation. For example, if the tabla player strikes the drum at a given time, this arrives at the live coding end only after a half-round-trip delay. Whatever the live coding performer sends back will arrive at the tabla player only after a further half-round-trip delay. Thus, in order to apply a delayed live electronic effect to the live tabla player's performance, the live coding performer must delay the input from the tabla by the intended delay time minus a full round-trip delay time.

This modified delay is accomplished via a SuperCollider extension that is maintained specifically for the ensemble, with has the added benefit of supporting theatrically-named class methods. This allows the projected code to accurately display the intention of delaying the sound by a given metric amount, while concealing the modification to the delay time that is made necessary by the network music situation:

```
*delay {
    | signal,time |
    ^DelayN.ar(signal,time-roundTrip,time-roundTrip);
}
```

Given the preceding method in the VLCat class/extension, the following is an example of a simple sound transformation whereby a delayed signal from the tabla player affects the envelope of a synthetic signal four beats later, with this intention clearly signalled in the code (the 4/TempoClock.tempo, and experienced as a one-bar delay at both ends of the telematic link:

```
~a = { LFTri.ar([440,550],mul:-20.dbamp) }
~b = { Amplitude.ar(Envelope.ar(VLCat.delay(~tabla.ar,4/TempoClock.tempo),0.1,0.01) }
~c = { ~a.ar * ~b.ar }
~c.play
```

5. Supporting Videos

Delay Study (February 2015): https://www.youtube.com/watch?v=2gehI46oNvk

Groove Study (February 2015): https://www.youtube.com/watch?v=2i9CG2Ayl8A

5.1. Acknowledgements

We gratefully acknowledge the assistance of the Centre for Interdisciplinary Research in Music Media and Technology (CIRMMT) at McGill University, and the Arts Research Board at McMaster University.

6. References

Barbosa, Álvaro, Jorge Cardoso, and Gunter Geiger. 2005. "Network Latency Adaptive Tempo in the Public Sound Objects System." In *Proceedings of New Interfaces for Music Expression 2005, Vancouver, BC*. http://mtg.upf.edu/files/publications/9d0455-NIME05-barbosa.pdf.

Cáceres, Juan-Pablo, and Alain B. Renaud. 2008. "Playing the Network: the Use of Time Delays as Musical Devices." In *Proceedings of International Computer Music Conference*, 244–250. http://classes.berklee.edu/mbierylo/ICMC08/defevent/papers/cr1425.pdf.

Caceres, J.-P., and C. Chafe. 2009. "JackTrip: Under the Hood of an Engine for Network Audio." In *Proceedings of the International Computer Music Conference*, 509–512.

Carôt, Alexander, and Christian Werner. 2007. "Network Music Performance-Problems, Approaches and Perspectives." In *Proceedings of the "Music in the Global Village"-Conference, Budapest, Hungary*. http://www.carot.de/Docs/MITGV_AC_CW.pdf.

Chafe, Chris, Juan-Pabla Cáceres, and Michael Gurevich. 2010. "Effect of Temporal Separation on Synchronization in Rhythmic Performance." *Perception* 39: 982–92.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (03) (Dec). doi:10.1017/S135577180300030X. http://www.journals.cambridge.org/abstract_S135577180300030X.

Courtney, David. 2000. *Advanced Theory of Tabla*. 1st ed. Houston: Sur Sangeet Services.

McCartney, J. 1998. "Continued Evolution of the SuperCollider Real-Time Synthesis Environment." In *Proceedings of the ICMC*. http://quod.lib.umich.edu/cgi/p/pod/dod-idx/continued-evolution-of-the-supercollider-real-time-synthesis.pdf?c=icmc;idno=bbp2372.1998.262.

Shepherd, Frances. *Tabla and the Benares Gharana (PhD Dissertation)*.

Stelkens, J. 2003. "PeerSynth: a P2P Multi-User Software with New Techniques for Integrating Latency in Real Time Collaboration." In *Proceedings of the International Computer Music Conference*.

Tanaka, Atau. 2006. "Interaction, Experience, and the Future of Music." *Computer Supported Cooperative Work* 35: 267–288.

Whalley, Ian. 2012. "Internet2 and Global Electroacoustic Music: Navigating a Decision Space of Production, Relationships and Languages." *Organised Sound* 17 (01) (Apr): 4–15. doi:10.1017/S135577181100046X.

extramuros: making music in a browser-based, language-neutral collaborative live coding environment

David Ogborn
McMaster University
ogbornd@mcmaster.ca

Eldad Tsabary
Concordia University
eldad.tsabary@concordia.ca

Ian Jarvis
McMaster University
lostinthegroove@gmail.com

Alexandra Cárdenas
University of the Arts in Berlin
tiemposdelruido@gmail.com

Alex McLean
University of Leeds
alex.mclean@icsrim.org.uk

ABSTRACT

The extramuros software was developed to explore live coding and network music, bringing live coding musicians together around shared text buffers. Originally developed to support a globally distributed live coding ensemble, the extramuros software has found additional application in projecting laptop orchestra performances to remote sites, in zero-installation workshop and performance settings, and in facilitating the efficient display of code by an ensemble. As the software works by connecting shared text buffers to audio programming languages through specific network connections augmented by pipes, it is a language-neutral approach. This paper describes the overall architecture of the extramuros system, relating that architecture to perennial network music concerns for bandwidth, security, and synchronization. Examples of early use in workshops, rehearsals and performances by laptop orchestras and other small telematic ensembles are discussed, leading to a concluding discussion of directions for future work.

1. Introduction

Music is deeply collaborative, and network music (Bischoff, Gold, and Horton 1978; Kim-Boyle 2009; Fields 2012) wherein musicians of different types and genres collaborate (usually but not always over the Internet) is a burgeoning field of research. The main currents of research include a focus on extending traditional musical performance over the network (Alexandraki and Akoumianakis 2010), as well as exploring the network as an inherent part of a "new" instrument (Gremo 2012; Cáceres and Renaud 2008). Live coding can also be deeply collaborative. Code is a highly efficient means of communicating intentions, with efficiency potentially understood as any or all of efficiency in comprehension by a human reader, efficiency in transmission over a network, or efficiency in translation into a sounding result. The extramuros software was developed to explore live coding and network music, bringing live coding musicians together around shared text editing interfaces ("shared buffers").

In the basic operation of extramuros, a server application is run on one computer. Some number of performers use a web browser to connect to the server and edit shared text, with each person's editing more or less immediately visible to all the others. At any computer where the sounding result of the code is desired (for example, all of the individual laptops in a globally distributed laptop ensemble), a client application is run that receives evaluated text from the server and sends it to the audio programming language of choice over standard UNIX-style pipes. Anyone can edit anyone else's code at any time, and all code is sent to all listening clients (and thus, potentially executed on audio programming languages connected to all listening clients).

As the extramuros system simply collects text that is piped to another application, it is "language-neutral". This neutrality is especially pertinent in the specific context of the live coding community, which uses a large and growing number of distinct languages, with individual artists and researchers bringing forward new languages continually (whether as ephemeral experiments or as more persistent pieces of community infrastructure). extramuros' language-neutrality allows it to be a site that brings together "partisans" of different languages, because their familiarity with the extramuros web-based interface can represent a bridging factor next to their unfamiliarity with a given language. The extramuros server does not evaluate or render any code, but rather just sends such text code back to the local clients that pipe it to an existing audio programming language. Thus, as new text-based languages enter use, extramuros will continue to be immediately useful with them. Both Tidal (McLean and Wiggins 2010),] and SuperCollider (McCartney 1998) have been used heavily with current versions of extramuros.

This paper discusses some diverse applications of extramuros, the system's software architecture and how that architecture responds to some of the perennial challenges of network music, and then presents some qualitative feedback from early workshop deployments. The paper concludes with a discussion of directions for future development of this, and related, systems.

2. Distributed Ensembles and Other Applications

extramuros was originally developed with the intent of allowing a distributed ensemble of live coding performers to rehearse, perform and learn together, connected by the Internet. It was first used in an ongoing series of "shared buffer" performances with the language Tidal (McLean and Wiggins 2010), and then subsequently used in the first distributed algorave (Collins and McLean 2014) performance (Colour TV, a.k.a. Alexandra Cárdenas and Ashley Sagar, at the 2014 Network Music Festival).

While supporting such globally distributed ensembles was the primary intent behind the creation of extramuros, the approach taken here supports a number of applications or situations that go beyond this, including "projected ensembles" and "zero installation ensembles". The software also allows for a comfortable relationship between large ensembles and the live coding practice of sharing one's screen.

2.1. Projected ensembles

While not the original target of development, extramuros can be used for the "projection" of an ensemble's performance to a second location. For example, a laptop orchestra (Trueman et al. 2006; Smallwood et al. 2008; Wang et al. 2008) whose performance practice involves routinely playing together in the same room, can perform through the web browser interface, with the sounding result of that performance rendered both at their location and at another site, or sites.

The Cybernetic Orchestra at McMaster University (Ogborn 2012b; Ogborn 2014) employed this technique to participate in the 2014 Network Music Festival in Birmingham, UK. The orchestra performed a live-coding roulette in SuperCollider on a circular array of 8 loudspeakers, "projecting" the code from a closed location in Hamilton, Canada to the public festival site in the UK. As only small pieces of text code were traveling from Canada to the UK, this projection of 8-channel audio plus high resolution video required almost no bandwidth at all.

2.2. Zero installation ensembles

An unanticipated benefit of the approach taken here was the utility of extramuros in workshop and educational settings where participants arrive with diverse laptops lacking the specific audio programming environments used in the activities. Because, in such settings, people are together in a room, only a single computer needs to have an appropriately configured audio programming environment (and hardware) with the other computers accessing this through the browser interface. Moreover, the utility of this zero-installation aspect of the approach is not limited to workshop and other "entry-level" settings, but extends to any collective performance setting where quick and robust installation is required.

The Cybernetic Orchestra has taken advantage of this zero-installation aspect in at least three distinct ways: (1) to give live coding workshops to members of the community, (2) as a way of getting new members of the orchestra live coding during their first minute at a rehearsal, and (3) as a mode of performing together in public. In early February 2015, the orchestra performed at the world's first ever algoskate, an event combining live-coded dance music as in an algorave with ice skating. The orchestra's 8 performers were distributed in a long line alongside a skating rink, with each performer placing their computer on top of a large 200-watt, 15-inch sound reinforcement style speaker. All but one of the computers were connected to a local area network via Cat6 Ethernet cables (run through the snow) with the one computer connected to the network via Wifi. Because the event was part of a busy winter festival (the city of Hamilton's WinterFest) this outdoor setup had to take place very quickly (in less than an hour). The fact that the members of the ensemble only had to open their computers and web browser freed their attention to run cables through the snow to speakers and switches. During the algoskate, one member of the ensemble put on their skates, connected to the orchestra's network via WiFi and performed Tidal live coding on their smartphone (with zero configuration) while skating around the rink, thus entering the history books as the first person to skate and live code at the same time!

2.3. "Show Us Your Screens"

In the above-mentioned algoskate, another benefit of the extramuros approach was on display (pun intended). The web browser interface provides a single, unified visual method of sharing code with the audience. While the classic TOPLAP

demand to "show us your screens" is easy enough to make in the case of a solo or duo performer, it becomes more fraught as the number of performers increase. When everyone is coding Tidal or SuperCollider through a shared web browser interface, this interface provides a single visual object that can be shared with the audience, with no requirement for elaborate or bandwidth-hungry network video compositing.

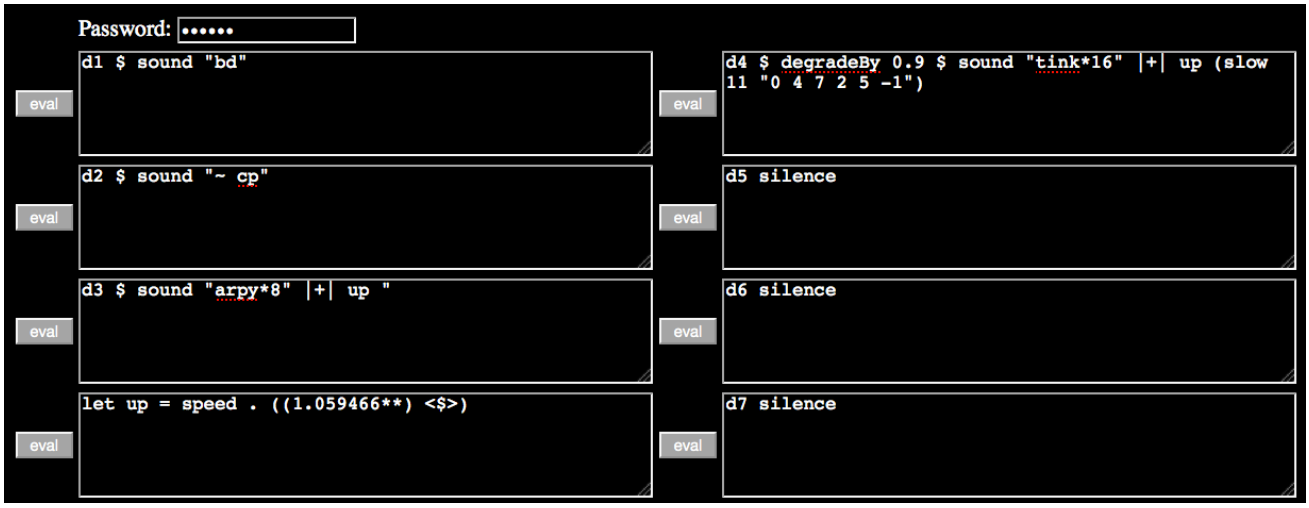


Figure 1: *A typical extramuros browser window with 8 text editing regions*

We can speculate that this type of application could have an impact on the evolution of live coding in large ensemble, "laptop orchestra" contexts. While many laptop orchestras are connected in some way with live coding, few have made it a central and defining activity. This could be in part because of the lack of an established practice of making a unified visual experience of live coding activity with so many people. While systems like Republic (Campo et al. 2012)), Utopia (Wilson et al. 2014), or EspGrid (Ogborn 2012a) allow performers in an ensemble to see what other performers are doing, copy their code, etc, they don't necessarily or inherently provide a unified visual focus and so performers tend to be in a situation of guessing what the other performers are doing (until closer investigation through their system's affordances provides the answer). Without special care, this becomes a situation of laptop performers closeted behind their screens, not fully co-present in the virtual space of the codework.

3. Network Music Challenges and Architecture

Performing collaboratively in networked contexts raises many challenges. The extramuros architecture responds to common network music challenges around security and timing/synchronization, while the fact that small pieces of text and operations on text are the primary pieces of transmitted information also makes the software quite economical in terms of bandwidth requirements. Before discussing the nature of these security and synchronization considerations, it will be helpful to outline the architecture of extramuros in slightly more detail. The extramuros server and clients are implemented in node.js. The share.js library (an operational transform library) is used for collaborative text editing, with common web browsers providing readily available virtual machines for the collaborative editing of text code. The 0mq library is used for reliable and scalable connections from the single server application back to the, potentially quite numerous, audio "render" clients. The rendering of code into audio results is out-sourced via UNIX pipes to external audio programming languages:

When a performer edits any of the editable text regions provided in the browser by the extramuros server, this involves back and forth communication between the browser and the server via websockets, mediated by the share.js library. This ongoing communication ensures that all browsers who have that page open see the current state of all the editable windows, and can potentially intervene and change them. When a performer clicks an "evaluation" button next to any given editable region, this triggers an HTTP request to the server application, which if successful, triggers the output of text in the client application that is then piped to some audio programming language.

3.1. Security

Information security issues in network music come in two forms: sometimes common security measures represent obstacles to the performance of network music, while at other times it is required that network musicians implement additional security measures in order to protect their performances.

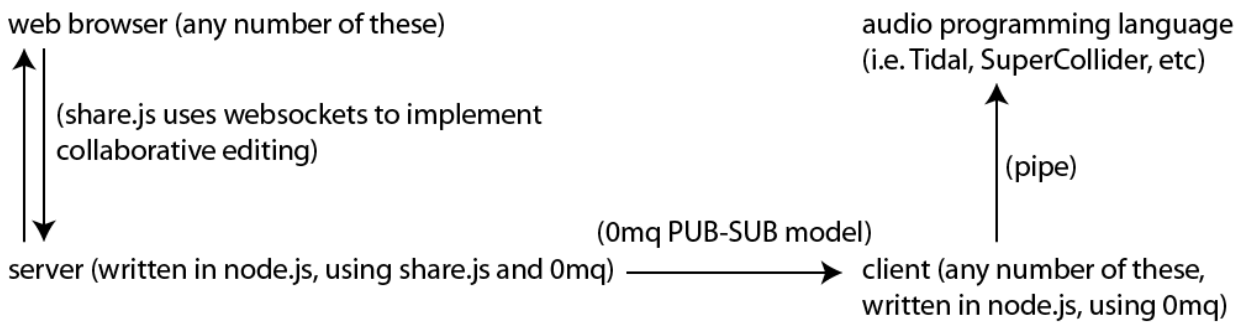


Figure 2: *Basic extramuros architecture diagram showing direction of data flow during editing and evaluation*

Firewalls are a common obstacle in network music performances, as the latter often take place in institutional or domestic contexts where access to the configuration of the firewall is prohibitive. In the case of extramuros, whose server listens for incoming TCP connections, firewalls are rarely a problem when the server application is run at publicly reachable location: all of the connections from other points/hosts to the server are "outbound" from the perspective of those other points/hosts, and firewalls tend to be configured to allow outbound connections on arbitrary ports without special reconfiguration. Provided a copy of the extramuros server is running somewhere, this allows potentially as many as all of the working/rendering/listening locations to be behind home or institutional firewalls. (At the first public "shared buffer" performance with extramuros, one of the performers was working on headphones at a café in a theme park, where one has to presume there would be no possibility of negotiating a momentary reconfiguration of the firewall!).

Use of extramuros entails open access to a web page requires some security measures in order to prevent sabotage. For example, on one occasion, an extramuros/tidal workshop was given to first year students at Concordia University in two groups. Once Group 2 was live coding, a mysterious code was hijacking the music with selfish, intrusive sounds, which were not generated by any of the participants. Apparently a mischievous student from Group 1 was having "some fun" from an unknown location. Identity still unknown, he/she apologized anonymously in the chat window.

Security is addressed primarily through a rudimentary password system. The web browser interface includes a field to enter a password. This password is included in the transmission back to the server whenever one of the buttons to evaluate code is pressed and the server quietly ignores any such evaluation requests that arrive without the correct password. If no password is entered, an in-browser alert pops up to remind the performer to enter a password, a measure suggested by early experiences, where reloading the browser window and forgetting to re-enter the password was a common mistake.

The current mechanisms do potentially open the door to security problems on the audio rendering/listening/client computers, and more attention to the security implications of the system are without a doubt an important direction for future work on the system. A part of the "solution" may be to establish practices and guidelines in each of the languages used with the system so that languages are addressed in a "secure mode", i.e. with no access to the file system outside of a certain folder or set of user permissions.

3.2. Synchronization

In the broadest sense, synchronization (i.e. latency) in extramuros is addressed by the fact that the same code is rendered at each render location. Depending on the nature of the structures deployed, the result at each client location could be more or less identical. When it is a matter of specifying patterns (in other words, quantized information at a relatively low frequency) as in the case with typical uses of Tidal, in most cases the different locations will render the same thing. There is the possibility that a piece of code might arrive just before a cycle boundary (i.e. a quantization point) on a given client machine and just after the same boundary on another client machine, so that briefly people will hear different things at each location. However, in a live coding context it is likely that a new evaluation will arrive a moment later with a similar relationship to the cycle boundary and then the identity of the two listening experiences (at least as signals at the DAC, not of course as identical acoustic and psychoacoustic experiences) will converge again.

In other code situations, without a high-level pattern focus and quantization, it is possible for the signals produced to diverge more. Consider, for example, the following SuperCollider/JITlib situation:

```
~a = { SinOsc.ar(0.1) } // someone makes a 0.1 Hz LFO on all render machines
~b = { SinOsc.ar(0.5) } // then a moment later, another LFO on all render machines
~c = { ~a.ar * ~b.ar } // then, the product of the two LFOs
```

The shape of oscillator ~c in the code above is highly dependent on the specific amount of time that elapses between the evaluation of ~a and ~b. Depending on the use to which ~c is put, this could result in substantially different sonic results. Collective live coding with the current extramuros architecture might take account of this dynamic by avoiding those kind of interdependencies between synthesis nodes:

```
~c = { SinOsc.ar(0.1) * SinOsc.ar(0.5) } // should be very close on all machines
```

However, this represents a rather unnatural constraint, and suggests that fine-grained time synchronization mechanisms, like EspGrid, could be quite important to the future development of networked live coding. Triggering code evaluation in the browser could schedule a packet of code to be evaluated at a more precise moment in the short-term future, as there are numerous situations where the sonic result can be quite different depending on small differences in the time of execution of code:

```
~a = { SinOsc.ar(1000) }
~b = { SinOsc.ar(1000) }
~c = { ~a.ar + ~b.ar } //
```

In the example above, the spectral energy at 1000 Hz could go from doubled (effectively, energy of ~a plus energy of ~b) to completely phased out, depending on a mere 0.5 millisecond change in the time of execution of ~b relative to ~a. At 5000 Hz, a 100 microsecond difference in time of execution (4 samples at 44100 Hz) can either double or eliminate the energy. The implementation and use of sample-accurate networked synchronization methods thus has a special relevance to projects like extramuros, that mirror the execution of code to multiple points.

Precise mechanisms for the controlling the logical time of execution of code, however, will not exhaust the broader question of coordinating the state of distributed machines. When code is executed in an extramuros browser window that calls upon any form of pseudo-random generator, the behaviour of the multiple client languages in the extramuros network will diverge. Intuition suggests that this is the type of challenge (i.e. random processes distributed over network architecture) that may ultimately be addressed by new languages designed from the ground up around the idea of collaborative, networked editing.

4. Workshop Feedback

On several occasions, electroacoustic (EA) majors at Concordia University (Montréal) live-coded in Tidal with extramuros in pedagogical settings, including a first year EA studio class, a second year EA ear training class, and a Concordia Laptop Orchestra (CLOrk) session. In an anonymous survey conducted with CLOrk members afterwards, students commented that live coding in this setting was “logical” and “flexible”, that it produced “instantaneous results”, and that it had “almost limitless potential” for “collaborative, digital expression” and “possibilities for composing [that] are quite vast.” Students also commented in the survey that this experience was “fun” because it “produced sound right away,” but that their ability to express themselves was between “very limited” to “none at all”, due to their inexperience with text-based programming. A student noted, “It would take a lot of time to feel flexible or expressive.” (This response is, of course, a response not only to the extramuros interface but also to the initial experience of programming audio operations in a text-based environment.)

According to students’ survey responses, this minimal expressive ability also led to minimal interactive ability in their first extramuros/Tidal tryout. Nonetheless, the visibility of the shared code on every laptop screen succeeded in generating at least some interactive engagement, as evident in these students’ survey comments: “it was interesting to have ensemble code-visibility” and “viewing the code of others gave me ideas about what gaps to fill, or when to be quiet.” Commenting on musicality, students noted that the effortless synchronization “worked” and that it was “a big plus.” One student felt that “abrupt entrances were annoying,” which were likely due to the inability to hear the code until it is evaluated. The student proposed that “some type of auditioning options” could be useful.

The majority of the students commented favorably on the effects of this first experience on their interest in live coding. Comments ranged from simple approvals, such as “it’s pretty cool,” “it has great potential,” and “Yeah, I would totally try it again” to insightful realizations such as “Yes, I never had a special interest towards live coding, but now I realize the amount of focus and the interesting symbiotic relationship with code.” While acknowledging the practical advantages of the zero installation and quick access allowed by extramuros, students asked to be tutored through the installation process in order to be able to practice at home and be more prepared, expressive, and flexible in upcoming group sessions.

5. Conclusions and Future Work

A major limitation of the current system is that it provides little in the way of feedback on the success or failure of code evaluations, or other states of the audio rendering environment (for example, audio levels). To remedy this, a mechanism is being created whereby arbitrary Open Sound Control messages arriving at the (single) server application, will come back to all of the web browser environments as calls to a JavaScript stub function. This should allow the development of distributed error messages, visual monitoring of audio levels, as well as more sophisticated forms of visualization of collective live coding activity.

It is quite common in the community that currently exists around extramuros for informal communication, during rehearsals and performances, to take place in an IRC channel or other external chat mechanism, but these mechanisms are not always strongly visible/present to participants. While the extremely spartan nature of the software has helped it to come into existence and support a number of artistic outputs, in the longer term the facilitation of communication and co-presence is doubtless an area of the software where considerable improvements could be made. Depending on the performance context it may be desirable to have video, audio, and/or data streams from each of the collaborators (Akoumianakis et al. 2008).

The language-neutrality of extramuros, based on the use of text code, is a project strength, and enables it to be used with diverse live coding communities, and also potentially as a thread bridging those communities. At the same time, text code comes with all kinds of cognitive demands that could be lessened through the exploration of structure editing techniques. These could be incorporated into extramuros without sacrificing the underlying language-neutrality in the form of something like the Emacs "major modes" for given languages, with the various structure constraints and affordances all calculated in the browser, and plain text still distributed to the audio rendering/listening clients as in the present architecture.

5.1. Acknowledgements

Special thanks to Research and High Performance Computing Services at McMaster University, to the McMaster University Arts Research Board for supporting the project Live Coding and The Challenges of Digital Society (of which this work is a component), and to Ash Sagar, Holger Ballweg, Scott Wilson, Mike Hodnick, and all of the members of the Cybernetic Orchestra and the Concordia Laptop Orchestra for using and discussing extramuros. Kudos also to the creators and contributors of node.js, share.js and 0mq!

References

Akoumianakis, Demosthenes, George Vellis, Ioannis Milolidakis, Dimitrios Kotsalis, and Chrisoula Alexandraki. 2008. "Distributed Collective Practices in Collaborative Music Performance." In *Proceedings of the 3rd International Conference on Digital Interactive Media in Entertainment and Arts*, 368–375. ACM. http://dl.acm.org/citation.cfm?id=1413700.

Alexandraki, Chrisoula, and Demosthenes Akoumianakis. 2010. "Exploring New Perspectives in Network Music Performance: the Diamouses Framework." *Computer Music Journal* 34 (2): 66–83.

Bischoff, John, Rich Gold, and Jim Horton. 1978. "Music for an Interactive Network of Microcomputers." *Computer Music Journal* 2 (3) (Dec): 24. doi:10.2307/3679453.

Cáceres, Juan-Pablo, and Alain B. Renaud. 2008. "Playing the Network: the Use of Time Delays as Musical Devices." In *Proceedings of International Computer Music Conference*, 244–250. http://classes.berklee.edu/mbierylo/ICMC08/defevent/papers/cr1425.pdf.

Campo, A. de, Julian Rohrhuber, Hannes Hoelzl, Jankees van Kampen, and Renata Wieser. 2012. "Towards a Hyperdemocratic Style of Network Music." In *Paper Presented at the SuperCollider Symposium.*

Collins, Nick, and Alex McLean. 2014. "Algorave: Live Performance of Algorithmic Electronic Dance Music." In *Proceedings of the New Interfaces for Musical Expression Conference, London, UK*. http://nime2014.org/proceedings/papers/426_paper.pdf.

Fields, Kenneth. 2012. "Syneme: Live." *Organised Sound* 17 (01) (Apr): 86–95. doi:10.1017/S1355771811000549.

Gremo, Bruce P. 2012. "Tele-Media and Instrument Making." *Organised Sound* 17 (01) (Apr): 73–85. doi:10.1017/S1355771811000537.

Kim-Boyle, David. 2009. "Network Musics: Play, Engagement and the Democratization of Performance." *Contemporary Music Review* 28 (4-5): 363–375. doi:10.1080/07494460903422198.

McCartney, J. 1998. “Continued Evolution of the SuperCollider Real-Time Synthesis Environment.” In *Proceedings of the ICMC*. http://quod.lib.umich.edu/cgi/p/pod/dod-idx/continued-evolution-of-the-supercollider-real-time-synthesis.pdf?c=icmc;idno=bbp2372.1998.262.

McLean, Alex, and Geraint Wiggins. 2010. “Tidal—Pattern Language for the Live Coding of Music.” In *Proceedings of the 7th Sound and Music Computing Conference*. Vol. 2010. http://server.smcnetwork.org/files/proceedings/2010/39.pdf.

Ogborn, David. 2012a. “EspGrid: a Protocol for Participatory Electronic Ensemble Performance.” In *Audio Engineering Society Convention 133*. Audio Engineering Society. http://www.aes.org/e-lib/browse.cfm?elib=16625.

———. 2012b. “Composing for a Networked, Pulse-Based, Laptop Orchestra.” *Organised Sound* 17 (01) (Apr): 56–61. doi:10.1017/S1355771811000513.

———. 2014. “Live Coding in a Scalable, Participatory Laptop Orchestra.” *Computer Music Journal* 38 (1): 17–30.

Smallwood, Scott, Dan Trueman, Perry R. Cook, and Ge Wang. 2008. “Composing for Laptop Orchestra.” *Computer Music Journal* 32 (1): 9–25.

Trueman, Daniel, Perry Cook, Scott Smallwood, and Ge Wang. 2006. “PLOrk: the Princeton Laptop Orchestra, Year 1.” In *Proceedings of the International Computer Music Conference*, 443–450. http://www.scott-smallwood.com/pdf/plork_icmc2006.pdf.

Wang, Ge, Dan Trueman, Scott Smallwood, and Perry R. Cook. 2008. “The Laptop Orchestra as Classroom.” *Computer Music Journal* 32 (1): 26–37.

Wilson, Scott, Norah Lorway, Rosalyn Coull, Konstantinos Vasilakos, and Tim Moyers. 2014. “Free as in BEER: Some Explorations into Structured Improvisation Using Networked Live-Coding Systems.” *Computer Music Journal* 38 (1): 54–64.

Deadmau5, Derek Bailey, and the Laptop Instrument – Improvisation, Composition, and Liveness in Live Coding

Adam Parkinson
EAVI, Goldsmiths
a.parkinson@gold.ac.uk

Renick Bell
Tama Art University
renick@gmail.com

ABSTRACT

Dance music superstar Deadmau5 and the improvising guitarist Derek Bailey represent, through their writing and practice, two very different approaches to performing live. By critically considering the practice of live coding in relation to these divergent approaches, we discuss live coding with regards to where the liveness lies and how the laptop and software are treated as a musical instrument. Each practice uses the laptop as a musical tool in a very different way. Live coding uses the laptop as an instrument in a manner that draws upon the techniques and strategies of free improvisation, in contrast to Deadmau5's notion of laptop as playback device and the live performance as spectacle. We discuss Deadmau5's practice in relation to Francisco Lopez's notion of the possibilities of electronic performance, and ideas about labour and liveness.

1. Introduction

In this paper, we will explore the practice of live coding in relation to two very different but related practices: the performance practice of dance music "superstar" Deadmaus, and the "non-idiomatic" free improvisation discussed in particular by British guitarist Derek Bailey in his book Improvisation: its Nature and Practice in Music (Bailey 1993). For the former, the "live" show is a synaesthetic, perfectly coordinated, precisely pre-planned immersive audio-visual experience in which the performer simply presses "play", whereas for the latter, the "live" show is continually composed in real-time. Sketching a continuum between these two practices provides a way of framing the practice of live coding. Doing so shows live coding as having more in common with 'traditional' instrumental practices which focus on an instrument and a performer on stage, engaged in real-time composition, than the live performer as a channel for a spectacle that is epitomised by the practice of Deadmau5. In particular, approaches to instrumentality found in free improvisation are present in live coding.

Deadmau5 is of interest to us given his prominent place in contemporary, popular electronic music combined with his controversial stance on the role of the musician in live performance, while Bailey's performance method and writing about free improvisation, although not electronic, match the declared intentions of many live coders and therefore provide a background for understanding improvisation and its relationship to the "live" in live coding. We see Bailey as an authority on what a certain 'pure' idea of liveness and improvisation, which provides a valuable contrast with Deadmau5's idea of live performance. Deadmau5 realises different notions of live electronic music practices that combine an almost Wagnerian spectacle with a dismissal of many traditional notions of virtuosity and instrumentality that might be construed as unnecessary in electronic music. In a sense, Bailey and Deadmau5 strive for opposite aims: Deadmau5 wants complete predictability and repeatability, a play button that always plays the same perfectly composed thing, whereas for Bailey each performance is a real-time composition seeking freedom from idiom and even the whole of musical history, and his practice rallies against that hypothetical play button. Neither of these aims is ever achievable, but they provide useful extremes within which to understand the practice of the live coding laptop musician.

These two views of a live performance form poles on a continuum of attitudes and ideologies of liveness, performance and musical instruments, upon which we can place and analyse live coding. Drawing on this, we suggest the following. Live coding situates the laptop as an instrument and locus of real-time composition bearing many similarities to 'traditional' instruments, and with the explorations of these instruments practised in relatively free improvisation. Live coding typically bears few similarities to the practices of Deadmau5, who treats the computer in a performance as a reproduction tool rather than a compositional one, despite possible aesthetic similarities (and frequently a relationship to dance music and dance music culture) that the two share.

2. Liveness

By examining this aforementioned continuum, it can be seen that while a designation of "live" in either case indicates performer activity as an event is underway, the type of activity in which the performer engages – composition or facilitation of spectacle – varies tremendously. The events we discuss can all said to be "live" yet in very different ways, and thus a worthwhile discussion of "liveness" can take place. Auslander has written an important contemporary text on liveness, which draws on Baudrillard, using the word "mediatized" in a fashion he describes as:

> loosely... to indicate that a particular cultural object is a product of the mass media or media technology. "Mediatized performance" is performance that is circulated on television, as audio or video recordings, and in other forms based on in technologies of reproduction. Baudrillard's own definition is more expansive: "What is mediatized is not what comes off the daily press, out of the tube, or on the radio: it is what is reinterpreted by the sign form, articulated into models, and administered by the code". (Auslander 2008, 5)

This supports the view that rather than being a polar opposite to "mediatized" (as in Auslander (Auslander 2008; Auslander 2000; Auslander 2006)), designations of "live" are used to declare that a performer is active in some way in front of an audience, with the following consequences:

> 1. Liveness can be based on the prior perception of performer activity or decision-making. 2. Liveness and mediatization can co-occur. Live laptop music involves the performance of the mediatized. Mediatization may in fact amplify perceptions of liveness. From this viewpoint, audiences call something 'live' when they feel aware of performer decisions, typically but not always manifest in explicit physical activity in the moment of the performance... (Bown, Bell, and Parkinson 2014, 18)

While older music technologies like violins might be construed as not being mediatised (even though they may be used to repeat and mediate a score), digital music technology is almost completely entangled with the technology of reproduction. Live coding is nearly always mediatised, explicitly so according to Baudrillard's definition, in so far as the laptop and code are commonly conceived and used as technologies of seamless reproduction. Software abstractions are precisely tools for the reproduction of certain behaviours in the computer. In the typical case, it uses the media of code, samples, synthesisers, and projection, yet its practitioners and audience have little doubt about it being characterised as "live". Considering liveness then from the view of activity, it remains to determine what type of activity is taking place. The liveness in live coding is fulfilled through a performer's activity in generating the sound, rather than a performer's presence as a figurehead in a spectacle.

3. Deadmau5 and Liveness in Contemporary Electronic Dance Music

Deadmau5, despite the name, is neither dead nor a rodent, but a very successful DJ and producer, known for his large-scale immersive live shows, over which he presides wearing a huge mouse head. For many, Deadmau5 may be seen as the epitome of everything that is wrong with Electronic Dance Music, and the embodiment of a particularly North American strand of 'stadium' dance music that is far away from the evasive "underground" spirit that many value. However, whilst practitioners involved might claim there is a great stylistic and artistic gulf between Deadmau5 and live coding, there are also many similarities. Both an algorave live coding event and a Deadmau5 concert are events held at night, often in clubs, where one of the performer's main intentions will be to make an audience dance, through the production and manipulation of electronic sounds triggered or synthesised by a laptop (Collins and McLean 2014). Regardless of any value judgements about the aesthetics or practice of Deadmau5, we believe that he cannot be dismissed and in his practice and blog posts we find a very interesting idea about liveness in contemporary electronic music.

On his tumblr (Zimmerman 2013), Deadmau5 famously declared "We all hit play", going on to say, "its no secret. when it comes to "live" performance of EDM... that's about the most it seems you can do anyway." He explains the mechanics of his live set-up, informing us that he can do some 'tweaking', but structurally most of his set is predetermined, premixed and tied in with queues for lights and video:

> heres how it works.... Somewhere in that mess is a computer, running ableton live... and its spewing out premixed (to a degree) stems of my original producitons, and then a SMPTE feed to front of house (so tell the light / video systems) where im at in the performance... so that all the visuals line up nicely and all the light cues are on and stuff. Now, while thats all goin on... theres a good chunk of Midi data spitting out

> as well to a handful of synths and crap that are / were used in the actual produciton… which i can tweak *live* and whatnot… but doesnt give me alot of "lookit me im jimi hendrix check out this solo" stuff, because im constrained to work on a set timeline because of the SMPTE. Its a super redundant system, and more importantly its reliable as FUCK! […] so thats my "live" show. and thats as "live" as i can comfortably get it (for now anyway)[1]

He trades off spontaneity for reliability: the best laid plans of Deadmau5 rarely go awry. In doing so, he diminishes the possibility to demonstrate his own personal virtuosity, but interestingly to us this seems to be of little concern to him, and he even goes on to be actively dismissive of claims and displays of virtuosic live electronic performance;

> Im just so sick of hearing the "NO!!! IM NOT JUST DOING THIS, I HAVE 6 TABLES UP THERE AND I DO THIS THIS AND THIS" like… honestly. who gives a fuck? i dont have any shame in admitting that for "unhooked" sets.. i just roll up with a laptop and a midi controller and "select" tracks n hit a spacebar.

For Deadmau5, what is important is not the ability to virtuosically do these things live and in real time, but the artistic processes involved in the composition and production of these tracks, and the immersive live show involving all the spectacles of lights and videos:

> my "skills" and other PRODUCERS skills shine where it needs to shine… in the goddamned studio, and on the fucking releases. thats what counts.[…] you know what makes the EDM show the crazy amazing show that it is? you guys do, the fans, the people who came to appreciate the music, the lights, all the other people who came, we just facilitate the means and the pretty lights and the draw of more awesome people like you by our studio productions. which is exactly what it is.

We could link this to the electroacoustic "tape music" tradition, where a "live" performance traditionally involves the playback of fixed media with some live spatialisation for the event; we could also link it to rock or even opera through the focus on the spectacle. Deadmau5 declares:

> "im not going to let it go thinking that people assume theres a guy on a laptop up there producing new original tracks on the fly. becausje none of the "top dj's in the world" to my knowledge have. myself included."

Deadmau5's approach is met with suspicion by many, especially those who want to see a musician actively doing something - effectively labouring - on stage, and for whom "liveness" might be signified by perceived performer activity in the area of composition. This does beg the question: at precisely what historical point did we expect our composers to be adept performers of their music anyway? The divisions between composer and instrumentalist are still alive in well in the world of classical music, and whilst there may be many problematic value systems and divisions of labour that accompany this, at least the role of composition is valued.[2]

Deadmau5's live sets blur the already murky distinctions between a "live" set and a "DJ" set (distinctions which we cannot fully explore here), and Deadmau5 suggests that most large scale electronic dance music performances operate along similar lines. The artist plays their original compositions, perhaps mixing and modifying parts, but doesn't compose new material in real time, and whilst the well established and traditional musical spectacle is alive and well, traditional ideas of instrumental virtuosity are discarded.

4. Derek Bailey and Improvisation

At the other end of the spectrum to Deadmau5's joyously unashamed play-pressing, we find the musical practice of free improvisation. The free improvisation to which we refer emerged primarily in the United Kingdom in the 1960s from a nexus of workshops and regular concerts. These include the regular sessions by the "Joseph Holbrooke" trio of Derek Bailey, Tony Oxley and Gavin Bryars above the Grapes in Sheffield, the improvisation workshops run by John Stevens in Ealing College, West London, the London Musicians Collective, the work of AMM and Cornelius Cardew, and the Spontaneous Music Ensemble sessions at the Little Theatre with John Stevens, Trevor Watts and Paul Rutherford. Free

[1] We have not corrected the spelling or grammar used in the blog post.

[2] The issue of labour is treated further in the last section of this paper.

improvisation drew on diverse musical sources including but not limited to American free jazz (in particular Ornette Coleman), the American avant-garde of Cage and La Monte Young along with the European avant-garde of Stockhausen. Histories of the influences on free improvisation, its early days and its position in Western music history are traced in Sansom (Sansom 2001) and Prevost (Prévost 2001).

Free improvisation is clearly not fully represented by the ideas and practice of the sole figure of Derek Bailey, but as a practitioner, and one who wrote about and spoke articulately about his practice and his lifelong relationship with a musical instrument (the guitar), he remains a pioneer and a useful figurehead for the purposes of our discussion. Bailey's book Improvisation: its Nature and Practice in Music was first published in 1980 and remains a canonical and widely referred to text on the subject of improvisation. It is still one of relatively few texts discussing the ideas and ideals of "free" improvisation (as opposed to jazz improvisation, though the scope of Bailey's book does also go beyond said "free" improvisation).

Bailey introduces improvisation in the following way:

> Defined in any one of a series of catchphrases ranging from 'making it up as he goes along' to 'instant composition', improvisation is generally viewed as a musical conjuring trick, a doubtful expedient, or even a vulgar habit (Bailey 1993).

He goes on to say of "free" improvisation that: "It has no stylistic or idiomatic commitment. It has no prescribed idiomatic sound. The characteristics of freely improvised music are established only by the sonic musical identity of the person or persons playing it." (Bailey 1993, 83). He calls it the oldest musical practice - for the earliest attempts at sound producing by humans presumably had no idiom or structure to be constrained by - and describes it as a recent movement emerging from a certain questioning of musical rules.

Whilst in the early days of free improvisation a lot of work was undertaken to escape musical idioms and convention - experimenting with escaping meter or regular tonal structures, for instance - we are drawing on free improvisation because it is a practice which foregrounds process and an immediacy with instruments and musical materials. Real time interactions and an explorations of the sonic possibilities of whatever musical materials are at hand take importance over the strict following of a score and the ability to reproduce seamless, identical versions of a piece of music that has already been written. We will see ways in which a similar focus on process, interaction and exploration of materials is is also brought to the fore in live coding.

In free improvisation, the instrument is re-imagined as a pure expressive tool, capable of generating a whole range of sounds unbounded by notions of what music should or shouldn't be, and how the instrument should or should not be used. This might be achieved through practice leading to general mastery and the honing of advanced techniques such as circular breathing amongst woodwind players, but it might also be achieved by an amateur who is lucky enough not to be conditioned by years of training to think that, for example, a guitar shouldn't be played with a toothbrush. Bailey speaks of "the instrumental impulse", which is quite separate from the act of playing in terms of score-following that many musicians might do. Discussing instrument-playing amongst non-improvisers, he notes, "[creating] music is a separate study totally divorced from playing that instrument" (Bailey 1993, 98). Improvisation explores the musical affordances of an instrument and begins with the instrument, not the score. For Bailey, the instrument "is not only a means to an end, it is a source of material, and technique for the improvisor is often an exploitation of the natural resources of the instrument (Bailey 1993, 99)". When Bailey says "[the] accidental can be exploited through the amount of control exercised over the instrument, from complete - producing exactly what the player dictates - to none at all - *letting the instrument have its say*" (italics our own) (Bailey 1993, 100), he could quite easily be talking about live coding, the use of generative algorithms, and the exploration of what a laptop and coding environment has to 'say' musically. These are the "natural resources" of this novel instrument.

5. Improvising with Computers

Free improvisation and the practice of Derek Bailey may seem an unusual bedfellow for computer music. The polemical biography of Derek Bailey uses the computer as an example of the antithesis of the permanent revolution of Bailey's playing. We are told that "what he plays is more consistently surprising than anything constructed on a computer: it happens in real time." (Watson 2004, 1) and that "nothing is quite as dull and boring and dead as knowing precisely what is going to happen - whether that is listening to computerised electronica…"(Watson 2004, 2). Elsewhere it is clear that the author has a more nuanced view of computers and creativity, but still, we see the computer being situated against the freedom and unpredictability of Derek Bailey. As we will argue, live coding makes specific efforts to reframe the computer as an instrument, as opposed to a machine of mediation, construction and predictability, and there is a whole history of using computers as unpredictable musical tools and even improvising agents.

George Lewis has been working on improvising with computers since the 1980s (Lewis 2000; Lewis 1999). Lewis's work places particular emphasis on the computer as improviser, moving more into the realms of artificial intelligence and researching "independent computer agency" (Lewis 1999, p.108) than many live coders. To an extent, his work explores whether we can treat the computer as a musician rather than as musical instrument. As he notes, in his most well known computer music piece Voyager "the computer system is not an instrument, and therefore cannot be controlled by a performer. Rather, the system is a multi-instrumental player with its own instrument," (Lewis 1999, p.103). Nonetheless, he is a pioneer of bringing computers into the sphere of improvised music and offering an alternative view of the musical role of the computer to that of the predictable sequencer. Other examples of improvising with computers which predate the live coding on which we focus in this paper would include the work of "The Hub" (Gresham-Lancaster 1998), Robert Rowe's interactive systems (Rowe 1999) and the work of Joel Chadabe (Chadabe 1984), to name but a few. Whilst these are all important in a historical framing of laptop improvisation and live coding, we will not discuss them at any length here as we wish to remain focused on the specifics of live coding: namely, real-time implementation and writing of code, and the projection of this code so the audience can see it, for it is here that the unique qualities of the practice emerge and we see the way in which the instrumental capacity of the laptop is explored.

The computer-as-instrument, and ways for expressively improvising with computers, is also widely explored in what we refer to as the New Interfaces for Musical Expression (NIME) field, where attempts are often made to extend the computer allowing for gestural control through accelerometers, biosensors and other means. One survey of this is found in (Miranda and Wanderley 2006). This is a field to which whole conferences (such as the New Interfaces for Expression Conference, the proceedings of which are another reference for any further investigations) are dedicated, and live coding often finds a comfortable place within this field. Again, it is outside the scope of our writing here to engage with much of this research, and we focus on live coding where very little attempts are made to extend or improve the input capabilities of the computer to make them more expressive: instead, the inherent, unadulterated affordances of keyboard, mouse and code are harnessed. Audience legibility is explored less through gestural legibility that through the projection of the performer's screen.

6. Live Coding

We will now draw on the practices of Deadmau5 and Bailey to examine some aspects of the practice of live coding. We define live coding as "the interactive control of algorithmic processes through programming activity as a performance". This practice is not homogenous, and live coding realistically refers to a range of practices and approaches with differing degrees of liveness, explored, for instance, by Church et al (Church, Nash, and Blackwell 2010). However, these practices are united by principles found in the draft manifesto (http://toplap.org/wiki/ManifestoDraft) (if we are to take Toplap as being representative of the movement, and for the sake of this paper, we will).

Live coders uses many of the same tools as Deadmau5 - that is, at the heart of the musical production set up there lies a laptop - but these tools are used in quite different ways, even though we might draw connections between the sonic outcomes. However, as we have seen, Deadmau5 does not believe that any of the "top djs in the world" use this tool for "producing new original tracks on the fly". Either Deadmau5 had not encountered live coding, or perhaps he was not considering live coders amongst these world class DJs, for it is exactly the idea of "producing new original tracks on the fly" that motivates much of if not all live coding. Peter Kirn, in one of the slew of aftermath articles that followed Deadmau5's blog post, brings attention to the multiplicity of practices in creating live electronic music that are "more live" than Deadmau5. (Kirn 2012) In this sense, live coding begins to have more in common with the liveness of composition in Bailey than the liveness of execution in Deadmau5.

Live coding nearly always involves - and more often than not actively encourages - improvisation and unexpected musical occurrences. Live coder Thor Magnusson declares, "Live coding emphasizes improvisation" and "[the] default mode of live coding performance is improvisation." (Magnusson 2014). There are multiple ways in which live coding remains an unpredictable musical environment to operate in. The frequent inclusion of randomness or algorithms with levels of complexity to generate musical materials which prevent the coder from being able to fully anticipate the musical output of their coding. Beyond the unpredictability of a program's output, live coders often execute their code in real-time in response to the preceding stream of sound, the sound of other players, the response of an audience, and so on. At the extreme end of the liveness in live coding is live coding "from scratch", which could be seen as an impossible ideal due to the underlying layers of abstraction on which a performance relies. The "from scratch" appellation signifies the coder's intention to perform in the moment or with their immediate presence, precisely the concept of "live" identified in Bown et al. (Bown, Bell, and Parkinson 2014) and in many senses an epitome of non-idiomatic free improvisation.

However, live coding is, in other respects, not the "free" improvisation as described by Derek Bailey. It is often working decidedly within or at least in a conversation with idioms, and trying to obey musical rules - often in situations where obeying of musical rules, such as the rhythmic structures of house and techno, is frowned upon (for instance, at ICMC).

Early free improvisation groups, such as Bailey's Joseph Holbrooke trio, were attempting to escape "the dogma of the beat", whereas live coding (for those generating dance music, at least) could be seen as trying to embrace the dogma of the beat. Nonetheless, free improvisation serves as a crucial reference point because it encapsulates an approach to instruments which explores their sonic affordances, such as we see in live coding. It also embodies the uncertainty which live coders cultivate, the uncertainty that makes it 'live' coding, and the uncertainty which distances it from the aspirations of total control that we find in Deadmau5. The approach to the laptop in live coding envisages it as a musical instrument quite different to the studio tool of Deadmau5.

7. The Instrument and the Stage in Live Coding

The manifesto on Toplap declares, "Show us your screens." Through projecting the performer's screen during performance, coding environments and the laptop itself are presented as an instrument, drawing on a traditional conception of instruments that might be closer to thinking about a guitar or piano than the way in which Deadmau5 and his audience imagine his laptop. Toplap notes that "[it] is not necessary for a lay audience to understand the code to appreciate it, much as it is not necessary to know how to play guitar in order to appreciate watching a guitar performance".

The showing of screens potentially allows for the demonstrations of instrumental virtuosity - what Deadmau5 mockingly describes as the "lookit me im jimi hendrix check out this solo" elements of performance - that are precisely what is absent in the performances of Deadmau5 and precisely what Deadmau5 is not interested in. This virtuosity might be perceived differently to the virtuosity of a guitarist or other traditional instruments which might seem more connected to the limits of the performer's body. Even to one unfamiliar with the playing of guitar, the sheer speed at which a performer can play can be impressive and indicative of virtuosity. The shared experience of a human body between performer and listener and an awareness of the physical limitations of that body can provide a common ground for understanding of virtuosity. Similarly, a long sustained note on a woodwind or brass instrument, perhaps achieved through circular breathing, might be perceived as virtuosic through its direct connection to the perceived or known limits of a body. The issues of speed and physical limitations in live coding is discussed by Sorensen and Brown, referring to their duo aa-cell:

> One problem that all live programmers must deal with is how to physically type the required code within a reasonable period of time; reasonable for both the audience but, probably, more importantly, to assist the performer to more rapidly realise ideas (Sorensen and Brown 2007)

It might be that the virtuosity in live coding is not anchored to the performing body in the same way that it might be for a guitarist and that touch typing will never be as impressive as Van Halen. Virtuosity on the part of a live coder might instead manifest itself through the use of difficult or less-common functions, complexity of code written live, or elegance of expression in coding. Within live coding, this transparency of the projected activity is interrogated: Green writes about the potential lack of transparency in code with low role-expressiveness (among other problems of notation), and work is being done to explore legibility in programming languages (Green 1989; McLean et al. 2010).

In the framing of laptop as instrument, with a focus on the legibility of that instrument, live coding could be seen as re-asserting elements of a concert hall tradition in electronic music. Describing the concert hall tradition of musicians on stage, electronic musician Francisco Lopez notes:

> From my perspective, electronic music doesn't need this. Of course it can have it, it can develop its own versions of it (as indeed it does). But it's not inherent to it, it's not a natural consequence of the practices and essential manners of the operations of electronic music, but rather a symbolic acceptance of a tradition of a very different nature (in this regard, probably an opposite nature). What is more important, I believe, is that by blindly following this tradition it wastes the potential for strengthening a most important breakthrough in music of perhaps historical proportions (López 2004).

Lopez discusses "the visible intricacy of instrument playing" that rock and classical traditions share, and that we might see live coding beginning to share, too. He notes:

> "While in the previous tradition of instrumental music each kind of sound corresponds to a certain gesture and to a specific physical instrument, in electronic music every possible sound is produced with the same click of a mouse, pushing of a button or turning of a knob. I don't find anything interesting in showing/contemplating these actions (if they are visible at all)." (López 2004)

Live coding has found a way around this: the interesting interaction happens not in the clicking of a mouse, but in the interactions on the screen and through code. "Show us your screens" becomes a way of either re-asserting an instrumental tradition in computer music or of increasing the transparency of the production. In the projection, electronic music performance returns to the stage. Live coding contrasts with the view of Lopez in that it finds value in the means of production, particularly its relation to the present.

It should be said that the highly visual nature of a Deadmau5 performance also differs from a Lopez performance which takes place in a completely darkened room, and there is strong link to operatic and rock spectacles in Deadmau5's performances Still, in some senses Deadmau5 might be construed as being closer to the radical abandonment of the stage that Lopez dreams of than the instrumentality of live coding in that a Deadmau5 performance, like a Lopez performance, discards the presentation of instrumental activity.

However, live coding ultimately avoids "blindly following" the tradition that Lopez rejects: if we are suggesting that live coding frames the laptop as an instrument in quite a traditional sense - more like the piano or the guitar than the studio tool of Deadmau5 - it should be noted that this is done in quite a radical way. It is a vital experiment that draws on the explorations of instruments in free improvisation, using the unique affordances of code and computer to create a live instrument in which ideas of score and composition are blurred, a set of extended techniques for drawing unexpected modes of expression out of a sophisticated adding machine. This serves to open up expressive possibilities in human-machine interaction, rather than closing them down through smuggling outdated musical value systems in through the back door. When the growing market for gestural controllers makes it all the more easy for a laptop performer to express virtuosity through physical gestures, drumming on pads, and navigating arrays of blinking lights, live coding is interrogating new languages of expression on stage whilst continually adapting its own instruments. Coding languages are developed and extended through new libraries, for instance, and the laptop-as-instrument develops, becoming a different machine in the hands of every programmer-performer.

7.1. Compositional and Instrumental Strategies in Live Coding

We have proposed that live coding quite explicitly treats the computer as an instrument, but it is a unique instrument begot by the very specific affordances of laptops and coding environments. A live coding performance will combine composition and improvisation, and the temporal nature of typing and evaluating code adds to the blurring between improvisation and composition within a live coding.

Composition in live coding ranges from the direct authoring of lists of durations and notes to meta-composition in which processes are defined which then determine the final sonic details. It blurs the distinction between instrument and score (A. Blackwell and Collins 2005, 3). Its instrument contains the score, and the score is manipulated in a performance to produce the media stream. The improvisational and compositional activity typically takes place in this range.

In the pre-performance activities of a live coder we see similarities to the studio composition of Deadmau5, and also the practicing, exploring and honing of new techniques before a concert by an improviser such as Bailey. Like Deadmau5's studio work, the live coder might make libraries in advance of a performance that might not be altered during a performance. On the other hand, like Bailey the live coder builds chops which are stored not in the muscle but in text files. The typically modular nature of these code tools and their spontaneous usage in a performance allows them to be used improvisationally rather than in inflexible arrangements like a pre-arranged Deadmau5 set. In other words, they are used live for compositional purposes rather than perfunctory execution.

The laptop-as-instrument in live coding is a novel instrument that in many instances is being continually refined and developed by practitioners in the field. Different musical challenges are met and, on occasion, tackled, a tactic we find in free improvisation where "a large part of most improvising techniques is developed to meet particular situations encountered in performance." (Bailey 1993, 99)

For instance, there is often a 'conversational' element of improvisation, as players will 'exchange' sounds, and to play in this manner one must be able to quickly move between sounds. For a variety of reasons, moving quickly between sounds is not always easy when performing with a computer. The difficulties of sudden changes in sound are discussed in Sorenson and Brown; Hession and McLean (Hession and McLean 2014) discuss attempts on McLean's part to develop his coding language to go beyond fixed tempos and time signatures and explore changes in performance, enabling a more 'conversational' improvisation.

The musical exploration of laptop and code is similar to the unconventional interrogations of instruments through 'extended technique' and other strategies that are common in free improvisation. Keith Rowe started playing his guitar flat on a table, Rhodri Davies plays his harp with a fan, and Sachiko M plays a sampler without loading sounds into its memory. The hidden sonic affordances of an instrument are revealed through unconventional uses. A paper from the early days of musical live coding describes a similar strategy at play:

> "With commercial tools for laptop performance like Ableton Live and Radial now readily available, those suspicious of the fixed interfaces and design decisions of such software turn to the customisable computer language. Why not explore the interfacing of the language itself as a novel performance tool, in stark contrast to pretty but conventional user interfaces?" (Collins et al. 2003)

Live coding frames the laptop as a real time musical instrument, different to the laptop as studio tool or composition device, which draws on some of the more traditional ideas of instruments and in some senses places it closer to acoustic instruments and traditions of stages and concert halls. However, this is a new instrument, a sonic exploration of a space of possibilities created by laptops and coding environments, and it is in a state of flux and development.

8. Labour, Composition, and Live Coding

Rather than debate the value of the performance ethics of Deadmau5, Bailey or live coders, a caveat is mentioned in order to show that given a continuum, it is possible to imagine performances which exist at different points along that continuum. Having argued this far, it now remains for a live coder to make a spectacle from some kind of perfect reproduction of programming. This could be a real-time presentation of the symbols of code without any compositional activity actually taking place, like executing pre-written code according to a script. It could be said that recent live performances of Ryoji Ikeda and Robert Henke performing as Monolake do something in this direction. Ikeda could be said to fetishise data or revel in its code-like appearance. The recent Monolake spectacle "Lumière" features the text "love" and "code" in a tightly synced live AV show (author 2014).

A live coder executing pre-written blocks of code without editing them in a pre-sequenced order could also be said to bear significant similarity to Deadmau5. It might also be argued that writing a "for" loop or even re-implementing a well-known algorithm in a performance puts the performer closer to the Deadmau5 pole if it can be said that there is little that is compositional in those two activities.

This brings into focus the issue of liveness and "labour". It seems some may want to see a performer working or engaging in physical activity, precisely the definition of "liveness" argued for here. This may stem from pressure of 'concert hall' and other older performance practices as mentioned in the discussion of Lopez above. Whether there is value in the non-compositional labour involved in writing out for loops or dealing with the requirements of the syntax of a language remains to be argued in a future paper. The visible screen of the live coder at least assures the audience that the labour being undertaken is appropriate to the task in hand and the performers fee (if they were lucky enough to get one), and they are less likely to be checking their Facebook, filing their tax return, replying to emails or submitting conference papers.

In a sense, Deadmau5 epitomises a new model for labour-free "live" computer music that earlier theorists have anticipated, one that embraces the very unique affordances of the computer as a servile musical tool. On the other hand, the physical spectacle of Deadmau5 simultaneously places him quite firmly in rock or operatic traditions. In contrast, live coding reveals the compositional labour to the audience.

9. Conclusion

Through looking at the very different practices of Derek Bailey and Deadmau5, we have described a continuum. They provide opposing poles on this continuum of liveness with regards to composition and thinking about instruments.

The aspirations towards transparency of activity in live coding contrast with the opaqueness of Deadmau5's activity in terms of its effect on the sound. Live coding values the performer's compositional activity in the performance whereas Deadmau5 does not, placing the emphasis on the live activity of the lights and the audience as co-performers and in the non-live compositional activity in the studio prior to the concert.

Live coding uses the laptop and software as a musical instrument. In some ways, this is quite traditional compared to Deadmau5, but perhaps more importantly we find an instrumental practice that relates to free improvisation, especially through the unbounded exploration of the musical potential of materials.

References

Auslander, Philip. 2000. "Liveness, Mediation and Intermedial Performance." *Degrés: Revue de Synthèse À Orientation Sémiologique* (101).

———. 2006. “Music as Performance: Living in the Immaterial World.” *Theatre Survey* 47 (02): 261–269.

———. 2008. *Liveness: Performance in a Mediatized Culture*. Routledge.

author, unknown. 2014. “Robert Henke: Lumière.” Product site. *Ableton.Com*. https://www.ableton.com/en/blog/robert-henke-lumiere-lasers-interview/.

Bailey, Derek. 1993. *Improvisation: Its Nature And Practice In Music*. Da Capo Press. http://www.worldcat.org/isbn/0306805286.

Blackwell, Alan, and Nick Collins. 2005. “The Programming Language as a Musical Instrument.” *Proceedings of PPIG05 (Psychology of Programming Interest Group)* 3: 284–289.

Bown, Oliver, Renick Bell, and Adam Parkinson. 2014. “Examining the Perception of Liveness and Activity in Laptop Music: Listeners’ Inference About What the Performer Is Doing from the Audio Alone.” In *Proceedings of the International Conference on New Interfaces for Musical Expression*. London, UK.

Chadabe, Joel. 1984. “Interactive Composing: An Overview.” *Computer Music Journal*: 22–27.

Church, Luke, Chris Nash, and Alan F Blackwell. 2010. “Liveness in Notation Use: From Music to Programming.” *Proceedings of PPIG 2010*: 2–11.

Collins, Nick, and Alex McLean. 2014. “Algorave: A Survey of the History, Aesthetics and Technology of Live Performance of Algorithmic Electronic Dance Music.” In *Proceedings of the International Conference on New Interfaces for Musical Expression*.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. “Live Coding in Laptop Performance.” *Organised Sound* 8 (03): 321–330. doi:10.1017/S135577180300030X. http://dx.doi.org/10.1017/S135577180300030X.

Green, Thomas RG. 1989. “Cognitive Dimensions of Notations.” *A. Sutcliffe and Macaulay, Editors, People and Computers V*: 443–460.

Gresham-Lancaster, Scot. 1998. “The Aesthetics and History of the Hub: The Effects of Changing Technology on Network Computer Music.” *Leonardo Music Journal* 8: 39. doi:10.2307/1513398. http://www.jstor.org/discover/10.2307/1513398?uid=3738328&uid=2134&uid=4578002247&uid=2&uid=70&uid=3&uid=4578002237&uid=60&sid=21102875061697.

Hession, Paul, and Alex McLean. 2014. “Extending Instruments with Live Algorithms in a Percussion/Code Duo.” In *Proceedings of the 50th Anniversary Convention of the AISB: Live Algorithms*.

Kirn, Peter. 2012. “Deadmau5, Honest About His Own Press-Play Sets, Misses Out On ‘Scene’.” http://createdigitalmusic.com/2012/06/deadmau5-honest-about-his-own-press-play-sets-misses-out-on-scene/.

Lewis, George E. 1999. “Interacting with Latter-Day Musical Automata.” *Contemporary Music Review* 18 (3): 99–112.

———. 2000. “Too Many Notes: Computers, Complexity and Culture in Voyager.” *Leonardo Music Journal* 10: 33–39.

López, Francisco. 2004. “Against the Stage.” *Unpublished Article. Retrieved May* 8: 2009.

Magnusson, Thor. 2014. “Improvising with the Threnoscope: Integrating Code, Hardware, GUI, Network, and Graphic Scores.” In NIME.

McLean, Alex, Dave Griffiths, Nick Collins, and Geraint Wiggins. 2010. “Visualisation of Live Code.” *Proceedings of Electronic Visualisation and the Arts 2010*.

Miranda, Eduardo Reck, and Marcelo M Wanderley. 2006. *New Digital Musical Instruments: Control and Interaction Beyond the Keyboard*. Vol. 21. AR Editions, Inc.

Prévost, Eddie. 2001. “The Arrival of a New Musical Aesthetic: Extracts from a Half-Buried Diary.” *Leonardo Music Journal* 11: 25–28.

Rowe, Robert. 1999. “The Aesthetics of Interactive Music Systems.” *Contemporary Music Review* 18 (3): 83–87.

Sansom, Matthew. 2001. “Imaging Music: Abstract Expressionism and Free Improvisation.” *Leonardo Music Journal* 11: 29–34.

Sorensen, Andrew, and Andrew R. Brown. 2007. “Aa-Cell in Practice: An Approach to Musical Live Coding.” In *Proceedings of the International Computer Music Conference*.

Watson, Ben. 2004. *Derek Bailey and the Story of Free Improvisation*. Verso.

Zimmerman, Joel. 2013. “We All Hit Play.” http://deadmau5.tumblr.com/post/25690507284/we-all-hit-play.

Sharing Time and Code in a Browser-Based Live Coding Environment

Charlie Roberts
University of California at Santa Barbara
charlie@charlie-roberts.com

Karl Yerkes
University of California at Santa Barbara
yerkes@mat.ucsb.edu

Danny Bazo
University of California at Santa Barbara
dannybazo@mat.ucsb.edu

Matthew Wright
University of California at Santa Barbara
matt@create.ucsb.edu

JoAnn Kuchera-Morin
University of California at Santa Barbara
jkm@create.ucsb.edu

ABSTRACT

We describe research extending the live coding environment Gibber with affordances for ensemble, networked, live coding performances. These include shared editing of code documents, remote code execution, an extensible chat system, shared state, and clock synchronization via proportional-integral control. We discuss these features using the framework provided by Lee and Essl in their 2014 paper *Models and Opportunities for Networked Live Coding*.

1. Introduction

Collaborative editing of documents has been a staple of browser-based authoring tools for many years, with operational transforms (Ellis and Gibbs 1989) enabling writers to freely edit documents with other authors concurrently. Concurrent editing of shared documents is especially promising for ensemble live-coding performance, as it offers performers the ability to view and edit the work of their fellow performers, but it also creates a variety of challenges. Having implemented concurrent editing of code in two different environments for live coding performance, we share lessons learned and outline feedback mechanisms to help inform ensemble members when code is executed during performance and who executed it. In addition, we describe research that automatically synchronizes Gibber instances to a master rhythmic clock via PI control structures, even with the inherent jitter imposed on communications by the TCP-based WebSocket protocol.

Our research enables a group of performers to sit in a room with their laptops, join a network, execute a single line of code, and immediately enter an ensemble performance where each performer's code is freely available for editing by every member and all clocks are synchronized. We call this system *Gabber*; it works inside of the browser-based live coding environment *Gibber* (Roberts and Kuchera-Morin 2012).

2. Strategies for Networked Ensemble Performance

Lee and Essl survey strategies enabling networked live coding performances in (Lee and Essl 2014a). Gibber incorporates many of these strategies; this section builds off their work by describing Gibber's networked ensemble affordances using their terminology.

2.1. Time Sharing

The first strategy discussed by Lee and Essl is *time sharing*, which concerns synchronizing events between performers. Although Gibber initially had no methods for sharing time, we have addressed this problem by phase-syncing all schedulers

in ensemble performances as described in Section 3. Despite the research outlined here on temporal synchronization, we note that there is a rich terrain of performance possibilities that can be explored without synchronization; three authors on this paper have successfully given a number of such performances (Roberts et al. 2014).

2.2. Code Sharing

Gibber implements code sharing at a number of different levels. For asynchronous collaboration, Gibber provides a centralized database enabling users to save and retrieve the programs (aka giblets) they create. This database is searchable, and users can provide a variety of metadata about giblets to aid in the search process. Giblets can be easily iterated and a versioning system enables users to examine changes throughout a giblet's development. Most importantly, every giblet can be accessed via a unique dedicated link, providing an easy mechanism for users to distribute their work to others.

Gibber's initial synchronous affordances enable any registered Gibber user to start collaboratively editing a code document with a remote user (who can be located anywhere with internet access) by simply clicking their name in one of Gibber's chatrooms; this feature also enables users to remotely execute code across the network if permission is granted. In this instance, users can either find each other through happenstance by joining the same chat room, or users who know each other can agree to meet in a particular chat room to participate in a shared editing session together. We envision the second instance being particularly useful for classes taught in computer labs or via distance learning; a teacher can have all students meet her/him in a particular chat room and then assist students as needed via the collaborative editing features.

Although performances consisting of a single, shared code editor initially seemed promising, in practice they proved problematic (Wakefield et al. 2014), as program fragments frequently shifted in screen location during performances and disturbed the coding of performers. A better strategy is to provide a dedicated shared editor for every member of the performance; this strategy was adopted in the *Lich.js* live coding environment (McKinney 2014) with read-only code editors. In other words, with N players there are N collaboratively editable documents, along with a social convention that each document uniquely "belongs to" or is somehow primarily associated with a single person. Our implementation of Gabber provides this capability, enabling everyone to code in their own personal editor and also read and edit the code of other users as necessary.

Although Gibber's original code-sharing model allowed this, quadratic growth of managing code sharing permissions made it too cumbersome for $N > 2$. We subsequently implemented a much simpler method for beginning networked ensemble performances in Gibber. After performers agree upon a unique name for a Gabber session, they begin the session with a single line of code:

```
Gabber.init( 'uniqueSessionName' )
```

Executing this either creates (if necessary) or joins (if already existing) a named networked live coding session on the worldwide Gibber server, and triggers the local browser to:

1. Open a Gibber chatroom named identically to the Gabber session that contains all the session participants.
2. Create a new user interface column containing everybody else's shared code editors (and updating dynamically as people join or leave).
3. Share the code editor where the call to `Gabber.init` was executed with all members of the session.

After these actions are carried out, the user interface of the Gibber environment will resemble Figure 1. Clicking on a tab in the shared editing column reveals the associated user's code editor.

2.3. State Sharing

There are two modes of state sharing in a Gabber session. The default *local* mode is designed for physically co-located performances: each user explicitly selects code (from anybody, as described in Section 2, although performers will commonly execute code they authored) and triggers execution on their own computer to generate a unique audio stream which renders only their personal actions. Users can also explicitly execute code on the machines of other ensemble members using specific keystroke macros. If the executed code is contained in personal editor of the executing performer then it is broadcast and run on all machines on the network. If instead the code is contained in the shared editor

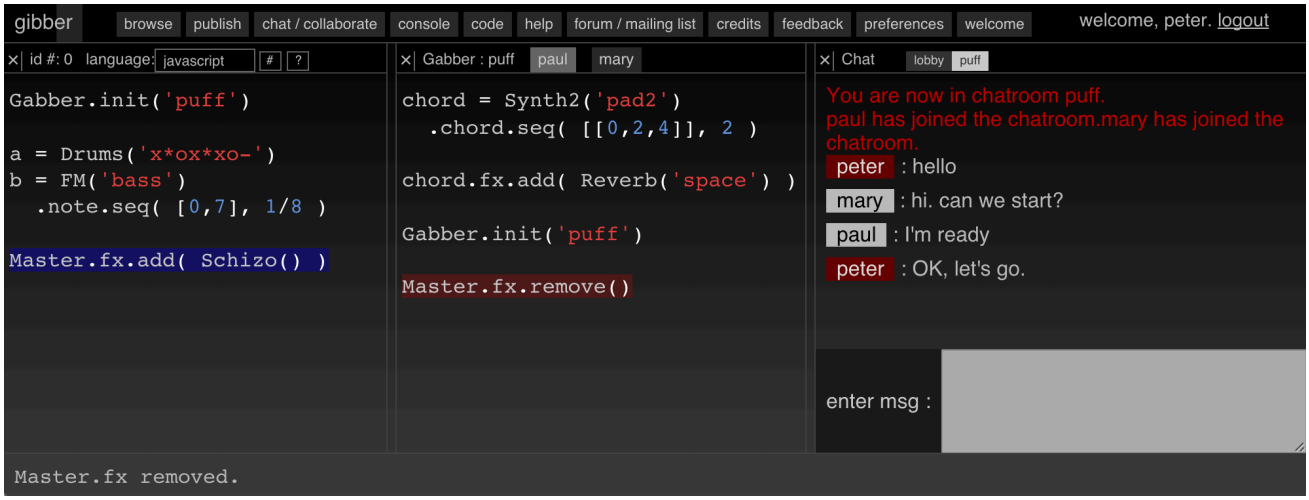


Figure 1: *A screenshot of the Gabber interface in a networked performance with three performers: Peter, Paul and Mary. The screenshot shows what Paul is seeing. Note the two highlighted code fragments; the blue highlight indicates code that the performer (Peter) has sent over the network to be executed. The red highlight indicates code that another member (Paul) of the ensemble has remotely executed on Peter's machine.*

of another performer it is only executed on the associated performer's computer, enabling, for example, a user to lower the amplitude of a particular ugen they find to be too loud.

In contrast, *remote* mode assumes performers are in physically disparate locations; this in turn means that each computer must locally render the entire performance. Executing any code in remote mode automatically also sends it across the network for execution by all ensemble members. Although prior research has streamed audio between ensemble members (Wang et al. 2005), remotely executing code offers a much lower bandwidth footprint, but comes with potentially higher CPU usage for each computer and the inability to share stochastic or interactive components of a performance.

In order to mitigate this, Gabber provides an easy mechanism for sharing arbitrary JavaScript data, drawing inspiration from the tuple approach adopted by *Impromptu* (Sorensen 2010) and the shared namespace used in *urMus* (Lee and Essl 2014b). The main `Gabber` object contains a `shared` data object which is distributed among all members of a Gabber performance. Defining a shared variable consists of a single method call; after executing this method any changes to the variable are distributed to all ensemble members. An example of defining a shared variable and changing it is given below:

```
Gabber.shared.add( 'pattern', [ 'c4','d4','g4' ] ) // define & distribute the initial 'pattern' variable
Gabber.shared.pattern = [ 'd4','d#4','g#4' ]       // change the pattern and distribute changes
```

Importantly, properties of the PI controller used to synchronize the clocks of Gibber instances (discussed in Section 4) are all properties of this shared object, enabling users to easily tune the controller and automatically propagate changes to all participants.

2.4. Access Control and Catching Them Red Handed

Although various strategies for limiting code execution on the machines of other ensemble members have been explored (Lee and Essl 2014b), Gibber uses a simpler, fully permissive scheme similar to the *Republic* quark for SuperCollider (Rohrhuber et al. 2007). Remote code execution, while presenting powerful opportunities for spatialized code execution, also has great potential for misuse and mischief. In performances by the CREATE Ensemble at UC Santa Barbara, one notorious user named `killbot` would routinely start algorithms running on remote machines that would gradually kill off running unit generators. As a result, designing killbot-proof algorithms became a game for the ensemble. A call to `Gabber.block( 'userNameToBlock' )` blocks remote execution by a particular performer, removing the challenge of blocking killbot but perhaps making for less stressful ensemble dynamics.

Even assuming that ensemble members have each others' best interests at heart, there is a need for performers to understand when (and by whom) code is remotely executed on their computers, e.g., so performers understand sudden changes in the sounds they are generating. For this reason we have added a number of simple visual annotations to Gibber to

indicate remote code execution in *local* mode, as shown in Figure 2. When code is executed and sent to all members of a performance, it is highlighted in red in all editors. When code is remotely executed on the computer of a single performer it is highlighted in blue.

2.5. Communication Facilitation

Each Gabber session has an associated chatroom for coordinating performance structure, discussing code, and banter. There are also programmatic hooks enabling users to detect when chat messages arrive and take action based on their content and/or author. These hooks have been used in summer camps for high school students to explore generative music-making in a program run by the Experimental Music and Digital Media program at Louisiana State University, and in classes at UC Santa Barbara.

3. Centralized and Decentralized Network Structures

Lee and Essl define a *centralized* performance network as one where clients submit code to a server for rendering. The first ensemble Gibber performance, performed in April of 2012, adopted this model (Roberts and Kuchera-Morin 2012; Roberts et al. 2014): a single server computer drove both the audio and visual projection systems, with scrolling displays of all code sent from clients and of the performers' chatroom. As Lee and Essl describe, this model automatically synchronizes time and state according to when the server executes each received code block. One additional benefit is the potential to preview code changes before submitting them to the server. Gibber's different keystroke macros to execute code locally vs. remotely enable performers to audition code for execution on headphones before sending it to the server for projection through the house speakers. This style of previewing only works in performances where each ensemble member is submitting code that does not rely on the state of the master computer; accordingly, in our first Gibber performance each member had particular ugens that they were responsible for creating and manipulating. Gabber can easily mimic these early performances. Ensemble members can use headphones on their personal laptops while a single, central, computer is connected to the audiovisual projection systems. Users target their personal computer (to audition) or the central computer (to present to the audience) for code execution as described in Section 2.3

Research presented in this paper extends Gibber to support the other two network models discussed by Lee and Essl. The *remote* mode of performance, described in Section 2.3, supports the *decentralized* model, where every machine runs all elements of the performance concurrently. The *local* mode of performance supports *decentralized programming with central timing*, enabling users to have personal control over state on their computers while providing temporal synchronization.

4. Network time synchronization

Many network time synchronization solutions exist (e.g., GPS, NTP, PTP), but most are implemented at the system level, requiring software installation and careful configuration of each machine in the network. Furthermore, these common solutions only maintain synchronization of system clocks (which Gibber does not use), not audio clocks (which Gibber does use). By implementing network time synchronization entirely within Gibber, we free users from the need to run external software to achieve synchronization. In effect, setting up a shared performance requires nothing more than an internet connection enabling users to connect to the worldwide Gibber server. The Gibber server is freely available on GitHub if users need to run their own local instance; this could be useful in a performance where an internet connection is unavailable or unreliable.

4.1. Synchronization with a Hybrid Proportional Controller

The Gabber server broadcasts its time (in samples) to all Gibber instances (clients) on the network via TCP/WebSockets once per audio buffer (typically 1024 samples). Upon receiving the server's time (now out-of-date because of network latency), each client subtracts its local time. Controller code within each client uses this difference (aka error) to decide how to correct the client's local time such that the client's timer converges to synchronize with that of the master.

We use a hybrid controller (Lygeros and Sastry 1999) with two modes: coarse and fine. The coarse mode operates when the client is out of sync beyond a particular threshold, and immediately sets the client's time to the most recently received master's time, disregarding any discontinuities this causes. For example, the first time a Gibber instance receives the server's time (when someone joins a performance already in progress), the coarse mode takes effect immediately, getting

the new performer roughly in sync with the rest of the group. Audio dropouts caused by spikes in CPU usage in the client can also cause this type of correction.

The fine mode, which operates when the error is below the threshold, uses a proportional controller to make small, smooth adjustments to the client's time. The default proportional constant was chosen empirically following experimentation and observation but can be freely edited by users; as described in Section 2.3 such changes to the controller are immediately propagated to all members of a performance. The controller acts on a low-passed version of the error signal due to the relatively low rate at which the error calculation and control is performed (typically 44100/1024 Hz, once per audio block callback). Changing the length of the running mean involves a tradeoff between fast response (short mean length) and smooth operation but long convergence time (long mean length).

Our nominal value of the threshold for switching control modes was also determined empirically through consideration of the minimum amount of time that a client would have to be out of sync for the coarse correction to be more aesthetically pleasing (approximately 100 ms for 120bpm 44100Hz sample rate group performance, varying according to musical features such as tempo). The threshold used for coarse correction is exposed to the user via Gabber's shared data object.

The code below is our controller implementation in JavaScript pseudocode:

```
K_p = .05
coarseThreshold = .1 * sampleRate
runningMean = RunningMean( 50 )
localPhase = 0

function onMessageFromServer( masterPhase ) {
  error = masterPhase - localPhase
  if( abs( error ) > coarseThreshold ) {
    // coarse controller
    localPhase = masterPhase
    Gibber.setPhase( localPhase )
    runningMean.reset()
  }else{
    // fine controller
    var phaseCorrection = K_p * runningMean( error )
    localPhase += phaseCorrection
    Gibber.adjustPhaseBy( phaseCorrection )
  }
}
```

(Brandt and Dannenberg 1999) presents a method of "Forward-synchronous" audio clock synchronization using PI-control. Their implementation accounts for noisy error measurement using a second PI controller that takes advantage of local, low-jitter system clocks on clients. We use signal conditioning (moving average) instead. Their system makes control adjustments much less often than ours.

(Ogborn 2012) describes EspGrid, a protocol/system for sharing time and code, as well as audio and video among players in networked laptop orchestras. EspGrid uses Cristian's algorithm (Cristian 1989) to present a synchronized system clock to each user so that s/he might schedule the playing of shared beat structures. We implemented Cristian's algorithm in Gibber, but found the results unsatisfying compared to our current solution. EspGrid has the benefit of using UDP while Gibber is limited to TCP/WebSockets (and the jitter associated with its error correction) for network messages. This could be one reason for our comparatively subpar results with Cristian's algorithm.

4.2. Results

In tests, clock synchronization in Gibber has produced aesthetically satisfying results. Sound events feel like they happen "in beat" even on a WiFi network, with users experiencing quickly fixed audio glitches. However in unfavorable WiFi conditions (i.e., heavy traffic, many overlapping networks, interference) the controllers in Gibber instances may exhibit thrashing and/or non-convergent correction. By adjusting the proportional control constant, coarse mode threshold time, and error signal low-pass length, the system can be made more tolerant of various differing conditions, but as a general rule it is best to use a wired LAN in performances where wireless conditions are less than ideal.

When a Gibber instance becomes "out of beat" for whatever reason (i.e. network problems, jitter, high CPU load, etc), it typically heals within the coarse mode threshold time. In our tests, measurements of the steady-state error for clients

on a WiFi network had a mean of 13.5 samples with a standard deviation of 39.2 samples. For comparison, the measured network jitter was approximately 2ms (~88 samples).

5. Conclusions and Future Work

We have described an extension to Gibber, named Gabber, that significantly eases setup of networked live coding performances. Joining a Gabber performance requires only a single line of code to be executed, and provides clock synchronization, code and state sharing, remote execution of code, and an extensible chat system for networked performances.

In the future, we are particularly interested in additional visual indicators documenting the actions of individual performers to their fellow ensemble members. For example, given that a performer is responsible for creating and manipulating a particular set of ugens, a waveform and/or spectral visualization of their summed output could be displayed in the associated tab of their shared editor on remote machines. This would enable users to, at a glance, see the overall spectrum and amplitude contributions of each member, potentially giving them an idea of who is responsible for each instrument.

Further future work will be informed by performances given using Gabber. Although the CREATE Ensemble has successfully performed a number of networked pieces using Gibber (as has the Laptop Orchestra of Louisiana) we have yet to perform with Gabber in its current incarnation. We are excited about the introduction of a shared clock for ensemble performances, and the resulting possibilities for networked algorave participation.

6. Acknowledgments

We'd gratefully acknowledge the Robert W. Deutsch foundation, who made this research possible.

References

Brandt, Eli, and Roger B Dannenberg. 1999. "Time in Distributed Real-Time Systems."

Cristian, Flaviu. 1989. "Probabilistic Clock Synchronization." *Distributed Computing* 3 (3): 146–158.

Ellis, Clarence A, and Simon J Gibbs. 1989. "Concurrency Control in Groupware Systems." In *ACM SIGMOD Record*, 18:399–407. 2. ACM.

Lee, Sang Won, and Georg Essl. 2014a. "Models and Opportunities for Networked Live Coding." In *Proceedings of the Live Coding and Collaboration Symposium*.

———. 2014b. "Communication, Control, and State Sharing in Networked Collaborative Live Coding." In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*, 263–268.

Lygeros, J., and S. Sastry. 1999. "Hybrid Systems: Modeling, Analysis and Control." UCB/ERL M99/34. Electronic Research Laboratory, University of California, Berkeley, CA.

McKinney, Chad. 2014. "Quick Live Coding Collaboration in the Web Browser." In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*, 379–382.

Ogborn, David. 2012. "EspGrid: a Protocol for Participatory Electronic Ensemble Performance." In *Audio Engineering Society Convention 133*. Audio Engineering Society.

Roberts, C., and J.A. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference*.

Roberts, Charles, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. 2014. "Gibber: Abstractions for Creative Multimedia Programming." In *Proceedings of the ACM International Conference on Multimedia*, 67–76. ACM.

Rohrhuber, Julian, Alberto de Campo, Renate Wieser, Jan-Kees van Kampen, Echo Ho, and Hannes Hölzl. 2007. "Purloined Letters and Distributed Persons." In *Music in the Global Village Conference (Budapest)*.

Sorensen, Andrew C. 2010. "A Distributed Memory for Networked Livecoding Performance." In *Proceedings of the ICMC2010 International Computer Music Conference*, 530–533.

Wakefield, Graham, Charles Roberts, Matthew Wright, Timothy Wood, and Karl Yerkes. 2014. "Collaborative Live-Coding Virtual Worlds with an Immersive Instrument." In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*.

Wang, Ge, Ananya Misra, Philip Davidson, and Perry R Cook. 2005. “CoAudicle: a Collaborative Audio Programming Space.” In *In Proceedings of the International Computer Music Conference*. Citeseer.

def Gribber = (Grace + Gibber)

Timothy Jones
Victoria University of Wellington, NZ
tim@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington, NZ
kjx@ecs.vuw.ac.nz

ABSTRACT

Grace is a new object-oriented education programming language that we are designing. One of the Grace implementations, Hopper, is an interpreter that runs on top of JavaScript. Gibber is series of libraries that support real-time audio processing, and also a simple livecoding interactive development environment, also on top of JavaScript. In this demonstration, we will present Gribber, a web-based IDE that incorporates the Hopper interpreter into the Gibber IDE, and so gives Grace live access to the Gibber audio libraries. Because of Hopper's continuation-passing design, Gribber programs can appear multi-threaded, apparently running one or more loops in parallel, each pseudo-thread generating sound and graphics in simultaneously. We will demonstrate how Gribber can be used for *BALSA*-style algorithm animation, and *Sorting Out Sorting* style algorithm races.

1. Introduction to Grace

We are engaged in the design of Grace, a new object-oriented open source programming language aimed at instructors and students in introductory programming courses (Black et al. 2012; Black et al. 2013; Homer et al. 2014). Grace aims to include features that have been found useful in software practice, while allowing multiple different teaching approaches without requiring that concepts be introduced to students before they are ready.

While many aspects of Grace's design will be familiar to most object-oriented programmers — we hope Grace's syntax, declarations, control structures, and method semantics appear relatively conventional — there are some aspects of Grace's design that are unique. In particular, Grace tries to separate the concepts of *object*, *class*, and *type*. In Grace, objects can be created without classes (like JavaScript or Self), or with classes (like most other object-oriented languages). Classes in turn can be built *by hand* out of objects (like Emerald, if anyone's counting). Grace types are optional (gradual): types in Grace programs may be left out all together (like Python or JavaScript), always included (like Java or C♯), or used in some mixture (like Dart or Dylan).

To give the flavour of the language, we show the Grace code for an imperative Insertion Sort, as may be found in many introductory programming course sequences. Hopefully this example is comprehensible to readers from a wide range of programming backgrounds:

```
method insertionSort(array) {
  var i := 2
  while {i <= array.size} do {
    var j := i
    while {(j > 2) && (array.at(j-1) > array.at(j))} do {
       array.swap(j) and(j-1)
       j := j - 1
       }
    i := i + 1
  }
  print "done insertion"
}
```

The first Grace implementation, Minigrace (Homer 2014) was a self-hosted prototype compiler to C and JavaScript. More recently, Jones has built Hopper (Jones 2015), a continuation-passing style interpreter written in JavaScript.

2. Livecoding in Grace

As well as being artistically and philosophically interesting in its own right (Blackwell et al. 2014), livecoding promises to increase engagement and learning in Computer Science pedagogy — indeed pedagogy has motivated much of the early work in live programming of many kinds (Papert 1980; Goldberg and Robson 1983). More recently, projects such as SonicPi (Aaron 2015), for example, use livecoding both to increase student engagement in programming, and also to support learning; indeed there is some evidence that livecoding can increase both engagement and learning even when it replaces static slide shows of the same content (Rubin 2013).

Unfortunately, our existing first Grace programming environment prototypes did not support livecoding. While Minigrace could self-host in JavaScript in a browser, it ran only in batch mode: code could not be modified on the fly, and loops (e.g. sort methods) would lock up the browser until they completed (Homer 2014). This is a problem for introductory programming courses where sorting is a common topic.

3. Grace in Gibber

We were able to solve these problems by combining our second-generation Hopper interpreter (Jones 2015) with the Gibber livecoding environment (Roberts and Kuchera-Morin 2012; Roberts et al. 2014). Gibber provides powerful audio and video libraries, and an IDE particularly suited to livecoding that can be easily extended to support different languages. JavaScript livecoding in Gibber unfortunately suffers from the problem that loops in programs will effectively delay the browser until they complete — this is a problem of JavaScript's single-threaded event-based programming model: each event must run to completion before the next event can begin execution. For e.g. livecoding and visualising a sorting algorithm, we need to visualise each interesting event in the algorithm as it happens, rather that just flick from an unsorted to a sorted array in a single time step.

The Hopper interpreter solves this problem by running Grace programs in a continuation-passing style. Each Grace subexpression (*method request* in Grace terminology) is executed one step, and then control is returned to the main loop, with the next step reified as a continuation (Abelson and Sussman 1985). After a timeout to allow other events, browser (or Gibber tasks), or indeed other Grace expressions to run, the delayed continuation will be picked up, again executed one step, again reified as a continuation and control yielded again. The effect is quite similar to SonicPi's use of Ruby's threading (Aaron and Blackwell 2014) except implemented solely in an interpreter written in JavaScript: the continuation passing and time slicing is not visible to the Grace programs. This allows Gribber to support *BALSA*-style algorithm animation (Brown 1988; Brown and Hershberger 1991), and *Sorting Out Sorting* style algorithm races (Baecker and Sherman 1981), which cannot be supported in Gibber in naked JavaScript. Figure 1 shows a Grace insertion sort running under Gribber in the Gibber IDE.

4. Conclusion

We have combined the Grace Hopper interpreter with the Gibber livecoding environment to produce Gribber — a livecoding environment for the Grace educational programming language. We hope Gribber will help communicate the advantages of livecoding to the educational programming languages community, and the advantages of a clean educational programming language design to the livecoding community.

5. Acknowledgements

We thank the reviewers for their comments. This work is supported in part by the Royal Society of New Zealand Marsden Fund and James Cook Fellowship.

References

Aaron, Sam. 2015. "π))): Sonic Pi." `http://sonic-pi.net`.

Aaron, Sam, and Alan Blackwell. 2014. "Temporal Semantics for a Live Coding Language." In *FARM*.

Abelson, Harold, and Gerald Jay Sussman. 1985. *Structure and Interpretation of Computer Programs.* MIT Press; McGraw-Hill.

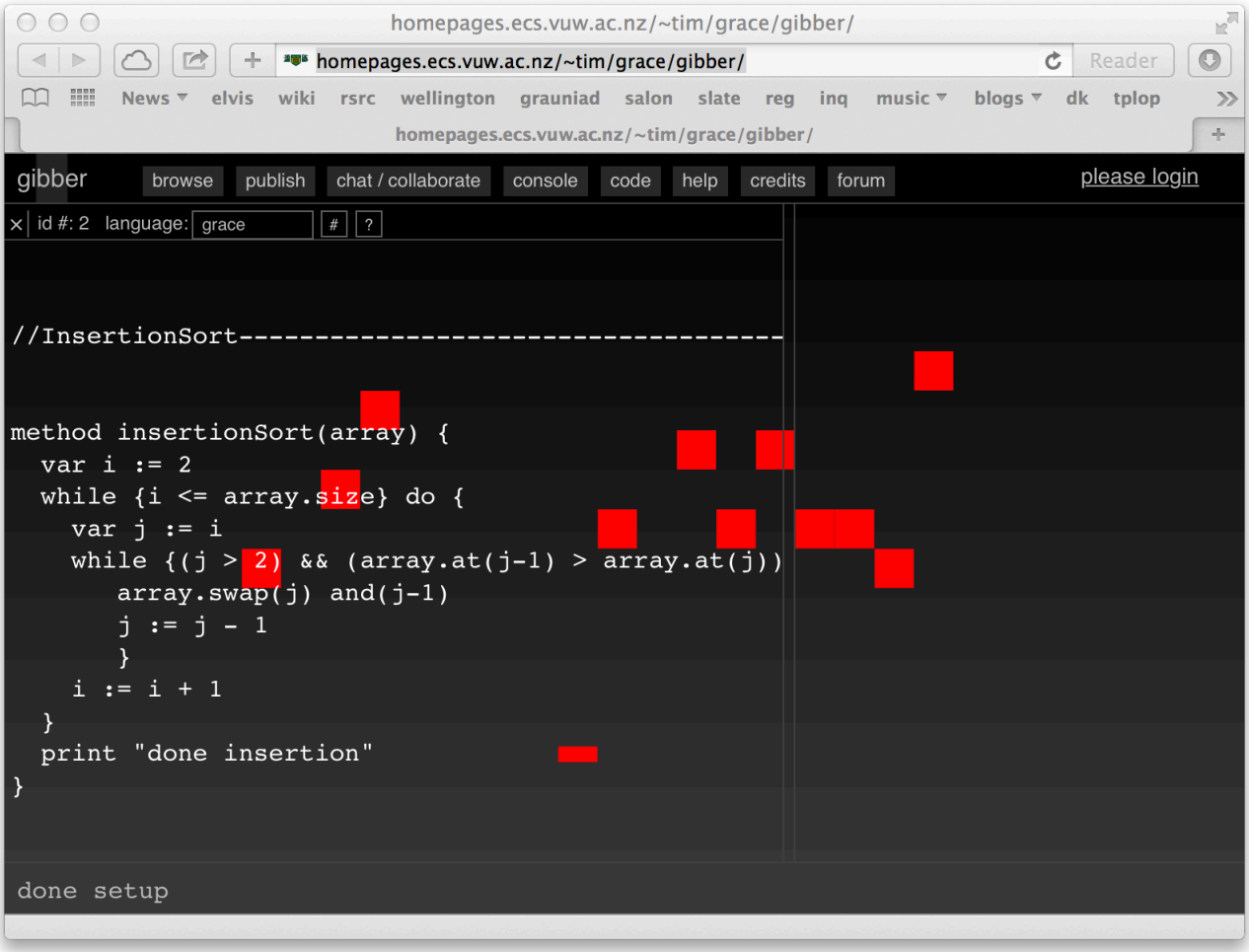


Figure 1: *A Grace insertion sort running under Gribber in the Gibber IDE. Note the translucent squares visualising element values: these are updated as the algorithm runs.*

Baecker, Ronald M., and David Sherman. 1981. “Sorting Out Sorting.” 16 mm colour sound film.

Black, Andrew P., Kim B. Bruce, Michael Homer, and James Noble. 2012. “Grace: the Absence of (Inessential) Difficulty.” In *Onward!*, 85–98.

Black, Andrew P., Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. “Seeking Grace: a New Object-Oriented Language for Novices.” In *SIGCSE*, 129–134.

Blackwell, Alan, Alex McLean, James Noble, and Julian Rohrhuber. 2014. “Collaboration and Learning Through Live Coding (Dagstuhl Seminar 13382).” *Dagstuhl Reports* 3 (9): 130–168.

Brown, Marc H. 1988. *Algorithm Animation.* ACM Distinguished Dissertation. MIT Press.

Brown, Marc H., and John Hershberger. 1991. “Color and Sound in Algorithm Animation.” *IEEE Computer* 25 (12) (December).

Goldberg, Adele, and David Robson. 1983. *Smalltalk-80: the Language and Its Implementation.* aw.

Homer, Michael. 2014. “Graceful Language Extensions and Interfaces.” PhD thesis, Victoria University of Wellington.

Homer, Michael, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. “Graceful Dialects.” In *ECOOP*, 131–156.

Jones, Timothy. 2015. “Hop, Skip, Jump: Implementing a Concurrent Interpreter with Promises.” Presented at `linux-conf.au`.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, Inc.

Roberts, C., and J. Kuchera-Morin. 2012. “Gibber: Live Coding Audio in the Browser.” In *ICMC*.

Roberts, C., M. Wright, J. Kuchera-Morin, and T Höllerer. 2014. “Gibber: Abstractions for Creative Multimedia Programming.” In *ACM Multimedia*.

Rubin, Marc J. 2013. “The Effectiveness of Live-Coding to Teach Introductory Programming.” In *SIGCSE*, 651–656.

From Live Coding to Virtual Being

Nikolai Suslov
Fund for Supporting
Development of Russian Technology
Vologda, Russia
SuslovNV@krestianstvo.org

Tatiana Soshenina
Moscow Institute of Architecture
(State Academy)
Moscow, Russia
TVSoshenina@gmail.com

ABSTRACT

The self-explorative, collaborative environments and virtual worlds are setting up the new standards in software engineering for today. In this, live coding is also required in reviewing as for programmers and as for artists too. The most popular live coding frameworks, even being built by using highly dynamic, reflective languages, still suffer on tight bindings to single-node or client-server architecture, language or platform dependence and third-party tools. That leads to inability nor to develop nor scale to the Internet of things the new created works using live coding. In the paper we introduce the prototype of integration of object-oriented language for pattern matching OMeta onto Virtual World Framework on JavaScript. That integration will allow using the live coding in virtual worlds with user-defined languages. Also we explore the possibilities of a conformal scaling of live coding in the case of augmented reality systems and Internet of things. In summary, the paper describes the efforts being done for involving virtual worlds architecture in live coding process. All prototypes that are described in the paper are available for experimenting with on Krestianstvo SDK open source project: http://www.krestianstvo.org

1. FROM CODER TOOLS TO SELF EXPLORATIVE, COLLABORATIVE ENVIRONMENTS

Fundamentally, live coding is about highly dynamic programming languages, reflectivity, homoiconicity and Meta properties. For example, families of functional languages like Lisp, Scheme, Clojure or pure object-oriented Smalltalk, etc. could be considered as "ready" for live coding. And adding to that languages some rich library of functions for working with network, multimedia (audio, video) and integrated program development environment (IDE) based on the debugger, makes them a complete environment for live coding. The most popular live coding frameworks still share this tools-centric approach, but the current developments in virtual & augmented reality, Internet of Things, robotics etc. shows, that this approach requires some review. The revival and emergence of the self explorative environments and virtual worlds, like Self language environment, Etoys, Scratch, OpenCroquet, LivelyKernel, Virtual World Framework is observed now. This will change the role of the programmer and artist in the process of live coding and also will change the process of live coding itself. See Figure 1 for the prototype of controlling several instances of SuperCollider using self explorative, collaborative Open Croquet based virtual world: Krestianstvo SDK.

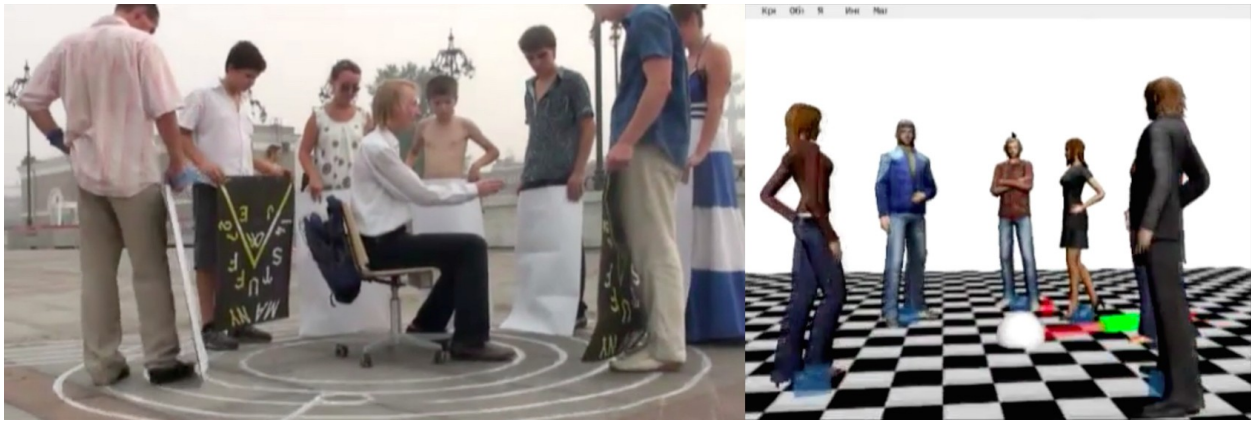


Figure 1. Installation Man'J (Soshenina 2010)

2. LIVE CODING IN VIRTUAL WORLDS WITH USER DEFINED LANGUAGES

We want to introduce the prototype of integration of object-oriented language for pattern matching OMeta on to Virtual World Framework. The integration will allow to define on any VWF component it's own language grammar and replicate it through the application instances, then have a running scripts based on that shared grammar for that component. For example, one could have all the languages down from Logo (Turtle graphics) to Lisp, Smalltalk, even SuperCollider, available for scripting the virtual world just in the Web browser in real-time, see Figure 2.

The Virtual World Framework (VWF) provides a synchronized collaborative 3D environment for the web browser. Continuing the Open Croquet research effort, VWF allows easy application creation, and provides a simple interface to allow multiple users to interact with the state of the application that is synchronized across clients, using the notion of virtual time. A VWF application is made up of prototype components, which are programmed in JavaScript, which allows a shared code and behaviors used in distributed computation, to be modified at runtime (Virtual World Framework 2015). OMeta is a new object-oriented language for pattern matching. It is based on a variant of Parsing Expression Grammars (PEGs), which have been extended to handle arbitrary data types. OMeta's general-purpose pattern matching facilities provide a natural and convenient way for programmers to implement tokenizers, parsers, visitors, and tree transformers (Warth 2007). We use ADL Sandbox project as a real-world application, which is built on top of VWF.

Figure 2. Integration of OMeta onto Virtual World Framework

To realize that integration we implemented the OMeta driver for VWF. The drivers in VWF define the autonomic actions that happen within a system, dividing responsibility and delegating work for each action of the system. These actions include things such as creating or deleting a node, getting or setting a property, calling methods, and firing events. To load the driver we were needed to include it in the files, that respond for VWF booting process.

As an example, see Figure 3 for a simple L-system parser-generator and Turtle-Sphere for drawing it (Suslov 2014). The virtual world holds two objects, one for drawing and another for parsing and generating L-system strings. For that purposes, the last one has two grammars, one for parsing user-defined input string

and another for parsing the generated structure. The rule of the generator is defined recursively in OMeta (Warth 2009). We used an enhanced Script editor inside the virtual world framework for editing the source of the grammar.

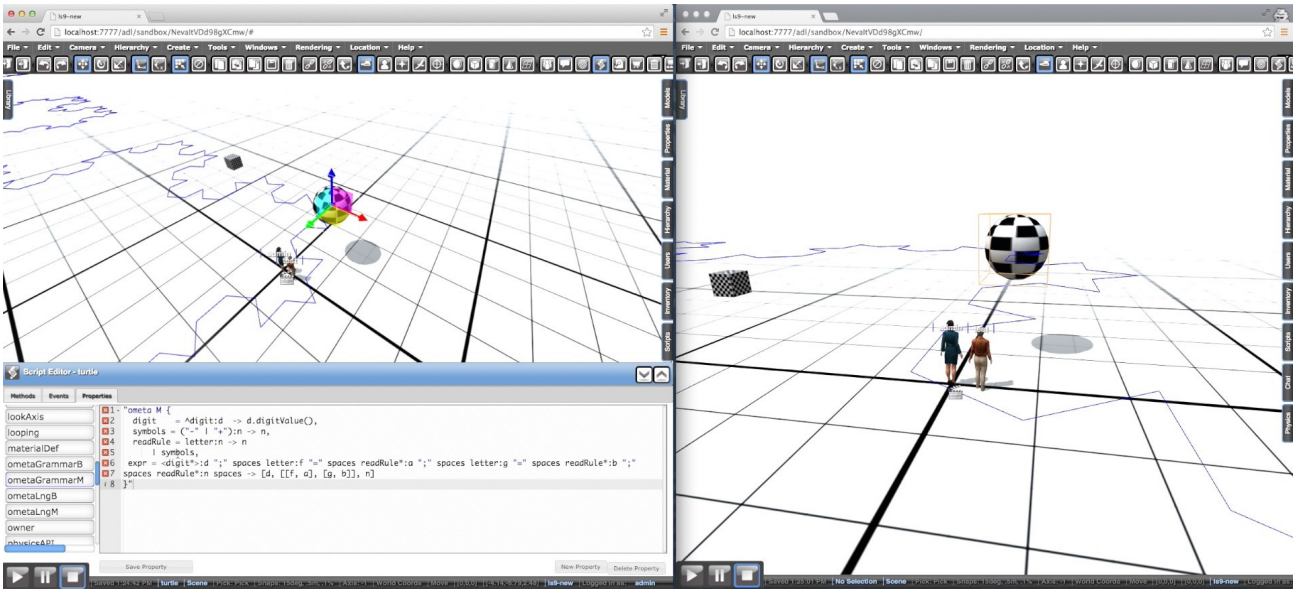


Figure 3. Editing L-system grammar in virtual world

That Script editor is working as Smalltalk browser in self-exploratory environment Squeak (Ingalls 1997). It allows to inspect the properties, methods, and events of the selected VWF component visually and to modify it.

Here is the source code of OMeta parsers of L-System in VWF and sample expression, which is defined by one of the live coding session participant in real time:

```
ometa ParseString {
  digit    = ^digit:d  -> d.digitValue(),
  symbols = ("-" | "+"):n -> n,
  readRule = letter:n -> n
      | symbols,
 expr = <digit*>:d ";" spaces letter:f "=" spaces readRule*:a ";" spaces letter:g "=" spaces readRule*:b ";"
spaces readRule*:n spaces -> [d, [[f, a], [g, b]], n]
}
ometa ParseStructure {
  symbols = ("-" | "+"):n -> n,
  line = (letter | symbols)*,
  rules = [letter [line]]*,
  takeRule:g = [takeRuleFor(g)*:m] -> (vwf.callMethod(nodeID,"flatCollection",[m])),
  takeRuleFor:g = letter:k -> (vwf.callMethod(nodeID,"selectRule",[g, k])) | symbols:n -> [[n]],
  gen 0 [rules] [line]:t -> t,
  gen :n :g takeRule(g):k = gen(n-1, g, k):m -> m
}
tree = LSystem.matchAll('5; F=F-F++F-F; G=; F++F++F', 'expr')
LSystemGen.matchAll(tree,'gen')
```

VWF could contain a lot of simulations running in it. These simulations could be based on data, generated by software, but also the data, that is gotten from any external hardware, like sensors. OMeta language will be a natural solution for parsing that incoming data in user-defined way.

3. CONFORMAL SCALING OF LIVE CODING IN THE CASE OF INTERNET OF THINGS AND AUGMENTED REALITY SYSTEMS

In general, a lot of existed live coding programming languages and their tools are based on single-node or client-server architectures. And these nodes are extended with a huge amount of communication protocol libraries, like Open Sound Control, MIDI, raw packets, user-defined API etc. for working with controllers and external nodes. So, the artist works in form of a "puppeteer", who actually has a quite different approaches in live coding of itself and connected to him nodes. In other words, very often the scaling mechanism from one to many nodes lies at the API and libraries level, instead of the programming language itself. The virtual worlds, which are based on the model of distributed computation, like Open Croquet and Virtual World Framework, solve that problem by generalizing the model of communication, omitting it up to the virtual machine level of the programming language. So, that we could observe a conformal scaling from one node to many, while never using live coding techniques in terms of communication protocols.

Figure 4. Photo of CAVE environment, running by OpenQwaq/Krestianstvo virtual world

We built the prototype of CAVE (cave automatic virtual environment), which is made of six nodes (personal computers), see Figure 4. CAVE application is based on our modified version of the OpenQwaq/Open Croquet virtual world development kit: Krestianstvo SDK. Four nodes are used for projecting virtual world onto the cube walls, one node is used for controlling movements in virtual world and the last one is used for setting the parameters of the CAVE. Open Croquet architecture was built for creating replicated virtual worlds, which are distributed over the network. Several running virtual machines and their corresponded images with source code, define the replicated state of the whole computation – Island (Smith 2003). Live coding could be done from any node, as all nodes are equal to each other. So, the coder is working in Smalltalk browser, which could be opened in virtual world also. Any modification in source code is replicated immediately to all instances of one shared Island. While coding, an artist uses well-known

Smalltalk language constructions. Scaling the CAVE with the new nodes is done just by starting the new ones and connecting them to the running virtual world. Using an avatar an artist adjusts the properties of the CAVE system, see Figure 5. The early prototype was used for installation in The State Tretyakov Gallery in Moscow and The National Pushkin Museum in Saint Petersburg for turning one of the exhibition halls into CAVE: "A. Ivanov Biblical sketches" installation.

So the artist gets the full featured connected virtual/real space to manipulate with. For example: it is easy to model augmented lightning for the physical room. The artist could freely experiment with the transitions and effects of a light's source, move it in the virtual space, while observing the visual effect on the real room's walls. More over, one could experiment with geometry transformations, like dividing, adding new virtual content, while projecting his distributed virtual model just onto physical space in real-time (Suslov 2012). One could use Microsoft Kinect, DMX controllers, TUIO markers, etc. connected right into virtual world.

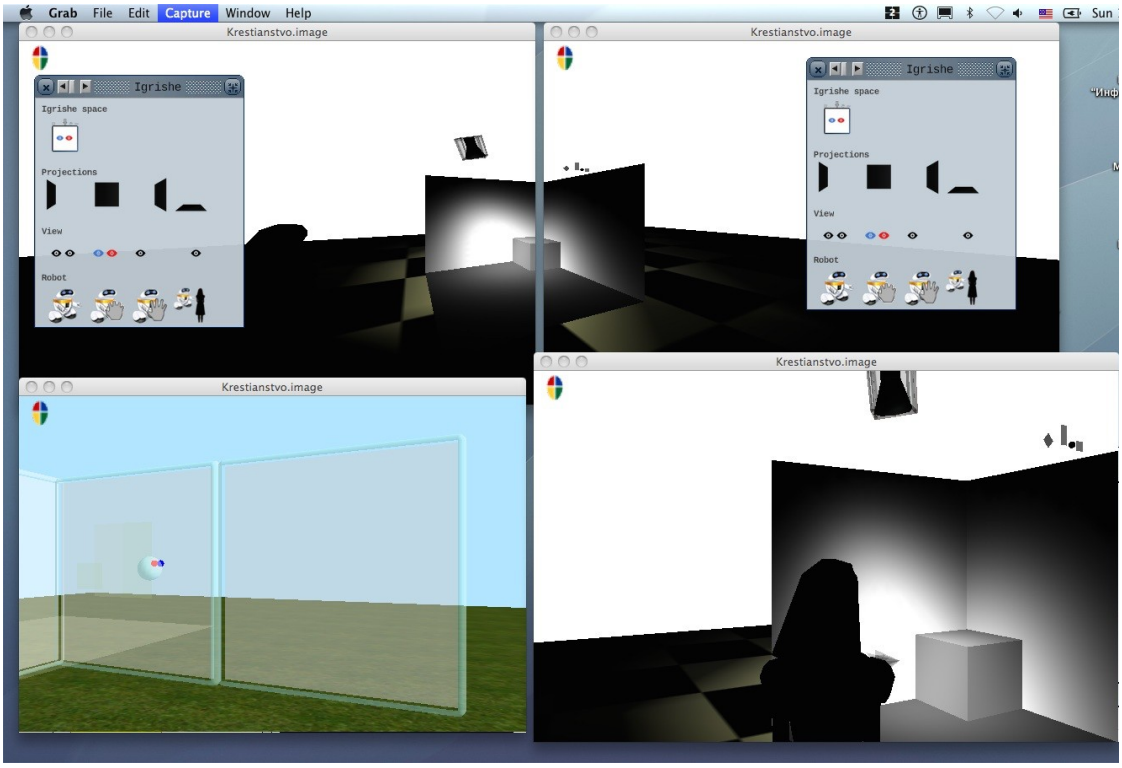


Figure 5. CAVE control virtual world based GUI

4. CONCLUSIONS

The novel VR, gesture, motion detection interfaces and robotics stuff allow the artists to build/run real-time performances/VJ sessions in Theatre, Art gallery and Learning labs, where they get the full featured connected virtual/real space to manipulate with (augmented reality). But, live coding in such cases become more and more complex task, due to heterogeneous nature of Internet of things. Existing live coding environments are continuing building up to the set of extensions and tools for artists-programmers. On the contrary, the virtual worlds have already started to provide the ability for code to be delivered in a lightweight manner and provide easy synchronization for multiple users to interact with common objects and environments. The Virtual World Framework is an example, which will help to interface different multimedia content, simulations, objects, users and locations; which will extend and expand the scope of live coding process and move it towards Virtual Being.

The integration of Virtual World Framework and OMeta makes possible a collaborative live coding on distributed objects with user-defined grammars. These objects could exist alongside each other in the same replicated virtual world, being programmed on quite different languages, but holding the same simulation. Also the integration allows researchers and programmers to experiment with language design ideas for

distributed objects in collaborative way. Due to Internet nature of VWF, OMeta being in it is extremely suitable for parsing an online content from hypertext to rest services and open data.

Acknowledgments

We would like to express thanks for the valuable insights that Victor Suslov, Sergey Serkov, Jonathan Edwards, Richard Gabriel, the participants and reviewers of the Future Programming Workshop at SPLASH 2014 have given to us. And to all others, who have helped in the realization of the projects, described in this paper.

REFERENCES

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. SIGPLAN Notices, 32(10):318–326.

Smith, D. A., Kay, A., Raab, A., and Reed, D. P. 2003. Croquet — A Collaboration System Architecture. In Proceedings of the First Conference on Creating, Connecting, and Collaborating through Computing (C5' 03), pages 2–9. IEEE CS.

Soshenina, Tatiana. 2010. Man'J, http://architectan.blogspot.ru/2010/08/manj.html, as of 07 March 2015

Suslov, Nikolai. 2012. "Krestianstvo SDK Towards End-user Mobile 3D Virtual Learning Environment". In Proceedings of the Conference on Creating, Connecting and Collaborating through Computing (C5) 2012, Institute for Creative Technologies, University of Southern California, Playa Vista, California, USA, IEEE, Page(s): 9 - 14

Suslov, Nikolai. 2014 "Virtual World Framework & OMeta: collaborative programming of distributed objects with user defined languages", The Future Programming Workshop at SPLASH 2014, Portland, Oregon, USA, video demo screencast http://vimeo.com/97713576, as of 07 March 2015

Suslov, Nikolai. 2012. CAVE in Krestianstvo SDK 2.0, https://vimeo.com/35907303, as of 07 March 2015

Virtual World Framework. 2015. https://virtual.wf/documentation.html, as of 07 March 2015

Warth, A. and Piumarta, I. 2007. OMeta: an Object-Oriented Language for Pattern-Matching. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, New York, NY, USA. ACM Press.

Warth, A. 2009. Experimenting with Programming Languages. PhD dissertation, University of California, Los Angeles.

Physical Livecoding with littleBits

James Noble
Victoria University of Wellington, NZ
kjx@ecs.vuw.ac.nz

ABSTRACT

littleBits (littleBits.cc) is an open-source hardware library of pre-assembled analogue components that can be easily assembled into circuits, disassembled, reassembled, and re-used. In this demonstration, we will show how littleBits — and the KORG littleBits SynthKit in particular — can be considered a physically-embodied domain specific programming language, and thus how assembling or improvising music with littleBits circuits is a tangible form of livecoding.

1. Introduction

littleBits (littleBits.cc) is an open-source hardware library of pre-assembled analogue components that can be easily assembled into circuits, disassembled, reassembled, and re-used (Bdeir 2009). Designed to inspire and teach basic electrics and electronics to school-aged children (and adults without a technical background) littleBits modules clip directly onto each other. littleBits users can build a wide range circuits and devices with "*no programming, no wiring, no soldering*" (Bdeir 2013) — even extending to a "Cloud Module" offering a connection to the internet, under the slogan "*yup. no programming here either* [sic]" (littleBits 2014).

The littleBits system comes packaged as a number of kits: "Base", "Premium", and "Deluxe" kits with 10, 14, and 18 modules respectively; and a series of booster kits containing lights, triggers, touch sensors, and wireless transceivers. littleBits have recently introduced special purpose kits in conjunction with third party organisations, notably a "Space Kit" designed in conjunction with NASA, and a "Synth Kit" designed in conjunction with KORG that contains the key components of an analogue modular music synthesizer.

In spite of littleBits' marketing slogans, we have argued that littleBits —- and the littleBits Synth Kit in particular (Noble and Jones 2014a; Noble and Jones 2014b) — as a live physically-embodied domain specific programming language. If building littleBits circuits is programming, then performing music with the littleBits Synth Kit (configuring modules to construct an analogue music synthesizer, and then producing sounds with that synthesizer) can be considered as a music performance by live coding (McLean, Rohrhuber, and Collins 2014) — especially as the circuit construction typically occurs simultaneously with sound production.

The demonstration draws directly on previous demonstrations given at the FARM workshop in Uppsala (Noble and Jones 2014a) and the VISSOFT conference in Victoria (Noble and Jones 2014b).

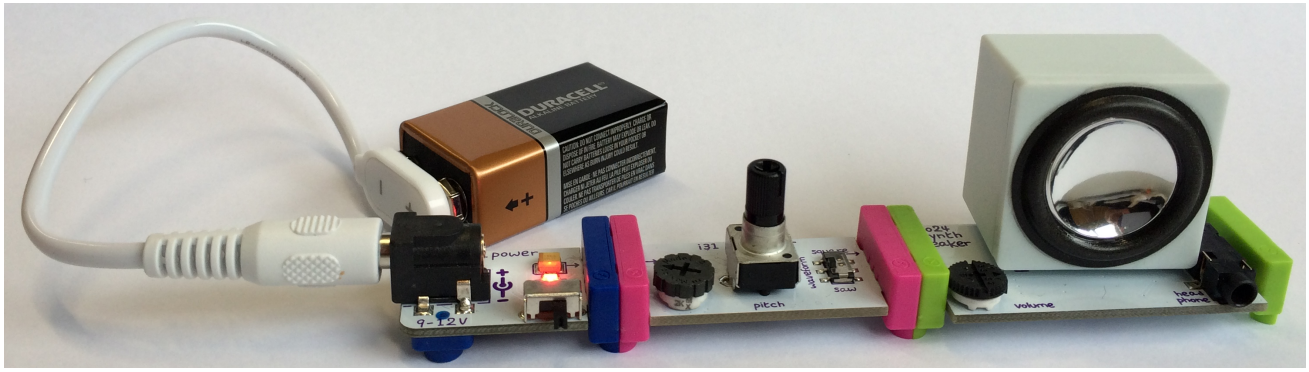


Figure 1: *A simple littleBits Synth Kit circuit. From left to right, the three modules are a power source, an oscillator, and a speaker.*

2. The littleBits SynthKit

Figure 1 shows the one of the simplest circuits in the littleBits synth kit — indeed, the simplest littleBits circuit that can actually make any sound. This circuit is composed of three simple modules — a power module on the left, an oscillator module in the centre, and a speaker module on the right. The power module accepts power from a nine volt battery (or a 9V guitar pedal mains adapter) and provides that power to "downstream" modules — as seen in the figure, littleBits circuits flow in a particular direction, and all modules are oriented so that this flow is left to right.

3. Livecoding with littleBits

If building and configuring circuits with littleBits can be considered as a form of embodied, tangible, programming, then performing music "live" with littleBits can be considered as a form of livecoding — performance programming to produce music (McLean, Rohrhuber, and Collins 2014; Blackwell et al. 2014; Magnusson 2014) — or in this case both *building* and *playing* synthesizers as a live performance. In this section we describe our practice livecoding littleBits, and compare and contrast with typically textual livecoding (inasmuch as typical livecoding practice can be supposed to exist).

This demonstration (Noble and Jones 2014a; Noble and Jones 2014b) draws on the author's experience livecoding/performing littleBits with "Selective Yellow", an experimental improvisation duo of indeterminate orthography drawing on New Zealand's heritage of experimental music practice (Russell 2012; McKinnon 2011), yet seeking to recreate (electronically) all the worst excesses of free jazz with all the enthusiasm of antisocial teenagers meeting their first MOS6851, while maximising the time required to set up equipment.

Selective Yellow performances typically employ a number of different synthesizers or sound generators as well as littleBits, ranging from digital toys (Kaosscilators, Buddhamachines) to semi-modular analogue and MIDI digital synthesizers, played with a variety of controllers (wind controllers, monome grids, knob boxes etc) — while eschewing a primary role for digital audio workstation software and computer-based virtual instruments. Selective Yellow is still a relatively young project, probably only Grade 2 as evaluated by Nilson (Nilson 2007).

Livecoding with littleBits involves two main activities that are tightly interleaved in a performance, first building the circuits by clipping modules together, and second "playing" the resulting synthesizer by turning the shafts, thumbwheels, switches, the "keys" on the keyboard module to actually generate sound. Generally a performance — or rather the portion of the performance improvised upon littleBits — starts with the smallest possible sound-generating circuit, typically the single unmodulated oscillator in figure 1. Once the littleBits are assembled (and the speaker module's output patched into the sound system) we can manipulate the oscillator's pitch and output waveform. Depending on the context of the improvisation, the possibilities of such a straightforward sound generator will be more or less quickly exhausted, at which point the performer will disassemble the circuit, insert one or more additional modules (a second oscillator, a filter, or perhaps a keyboard or sequencer module) and then continue playing the resulting circuit. In this overall pattern, littleBits livecoding is similar to some textual livecoding, where performers typically start with a single texture and then build a more complex improvisation over time.

While the circuit building and playing are conceptually separate activities, an advantage of the physical nature (and careful design) of the littleBits components is that the two activities can be very tightly interleaved. Indeed, with more complex circuits (or more than one Synth Kit) its is quite possible to keep part of a circuit playing and producing sound (such as a sequencer driving an oscillator) while building/editing another branch of the same circuit — adding in a second oscillator controlled by the keyboard module with an independent output route, perhaps, or adding in a modulation path to a filter that is already making sound in the main circuit. Again, this overall dynamic is also found in some textual livecoding performances (see e.g. the SuperCollider jitlib (Collins et al. 2003)). Of course, because of the underlying simplicity of the analogue synthesizer embodied within the littleBits modules, the sounds produced by littleBits Synth Kit are much less complex than the sounds that can be produced by a general-purpose laptop running a range of digital synthesis or sampling (virtual) instruments, although, being purely analogue, they have a piercing tone all of their own.

In the same way that programmatic live coders generally display the code on their laptop screens to the audience of the performance (Collins et al. 2003), Selective Yellow projects an image of the desk or floor space where the little bits circuits are being assembled. The projected image not only seeks to dispel "dangerous obscurantism" (Ward et al. 2004) but also to illustrate how the source is being generated - especially as some modules include LEDs as feedback to the performer. The sequencer module, for example, lights an LED to indicate the current sequencer step, and other littleBits modules can also be used to provide more visual feedback on circuits where that is necessary.

This projected display seems particularly useful for audiences when the performer is "debugging" their circuit (live). Here again the physicality of the littleBits modules comes to the fore, so there is something for the audience to see: the easiest way to debug a littleBits circuit is just to pull it apart, and insert a speaker module after each module in the circuit in turn,

listening to the sound (if any) being output by each module. Often this lets the performer understand (and the audience to notice) that there is no sound output from a littleBits circuit, allowing the performer either to readjust the module parameters, or to re-assemble the circuit in a different design, if not producing the desired sound, at least producing something.

4. Acknowledgements

Thanks to Chris Wilson, for being the other half of Selective Yellow. Thanks to a wide range of anonymous reviewers for their comments. This work is partially supported by the Royal Society of New Zealand (Marsden Fund and James Cook Fellowship).

5. References

Bdeir, Ayah. 2009. "Electronics as Material: littleBits." In *Proc. Tangible and Embedded Interaction (TEI)*, 397–400.

———. 2013. "LittleBits, Big Ambitions!" `http://littlebits.cc/littlebits-big-ambitions`.

Blackwell, Alan, Alex McLean, James Noble, and Julian Rohrhuber. 2014. "Collaboration and Learning Through Live Coding (Dagstuhl Seminar 13382)." *Dagstuhl Reports* 3 (9): 130–168.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (3) (December): 321–330.

littleBits. 2014. "Sneak Peek: the Cloud Block." `http://littlebits.cc/cloud`.

Magnusson, Thor. 2014. "Herding Cats: Observing Live Coding in the Wild." *Computer Music Journal* 38 (1): 8–16.

McKinnon, Dugal. 2011. "Centripetal, Centrifugal: Electroacoustic Music." In *HOME, LAND and SEA: Situating Music in Aotearoa New Zealand*, edited by Glenda Keam and Tony Mitchell, 234–244. Pearson.

McLean, Alex, Julian Rohrhuber, and Nick Collins. 2014. "Special Issue on Live Coding." *Computer Music Journal* 38 (1).

Nilson, Click. 2007. "Live Coding Practice." In *New Interfaces for Musical Expression (NIME)*.

Noble, James, and Timothy Jones. 2014a. "[Demo Abstract] LittleBits Synth Kit as a Physically-Embodied, Domain Specific Functional Programming Language." In *FARM*.

———. 2014b. "Livecoding the SynthKit: littleBits as an Embodied Programming Language." In *VISSOFT*.

Russell, Bruce, ed. 2012. *Erewhon Calling: Experimental Sound in New Zealand*. The Audio Foundation; CMR.

Ward, Adrian, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. 2004. "Live Algorithm Programming and a Temporary Organisation for Its Promotion." In *READ_ME — Software Art and Cultures*, edited by Olga Goriunova and Alexei Shulgin.

TextAlive Online: Live Programming of Kinetic Typography Videos with Online Music

Jun Kato, Tomoyasu Nakano, Masataka Goto
National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba, Japan
{jun.kato, t.nakano, m.goto}@aist.go.jp

ABSTRACT

This paper introduces a web-based integrated design environment named "TextAlive Online" that supports creating Kinetic Typography videos synchronized with songs available online. It is the hybrid of a content authoring tool and a live programming environment. Through its development, we investigate the interaction design that most benefits from the interactive user interfaces used by designers and programmers, as well as the collaborative nature of the open-source culture on the web. This system is accessible at textalive.jp and the interactive "live" version of this paper is available at textalive.jp/paper.

1. Introduction

The Internet has allowed many people to express their creativity in the connected world. The open-source culture has altered the way of content authoring. For instance, the Creative Commons license made it easy for creators to author derivative content from existing one. Many songs and their derivative content are uploaded onto websites like SoundCloud and YouTube. However, they are designed as a platform for distributing content, not for authoring it.

Programmers live in a far more advanced world in terms of content authoring. They can easily collaborate with others by forking repositories, sending pull requests, pushing back changes, etc. Recent Web-based Integrated Development Environments (WIDEs) integrate such online activities into the IDEs, supporting the users' collaborations. Live Programming is another important technique that has recently attracted attention and been implemented in various IDEs, helping the programmers' content authoring. It provides a fluid programming experience by continuously informing the user of the state of the programs being developed.

We foresee that this programmers' way of content authoring is powerful yet general enough to be applied to various types of content authoring. At the same time, the typical workflow of today's programmer is not only focused on designing algorithms but also involves more data organization and manipulation. There is prior research on development environments with support for such workflow. Representative examples include Gestalt for machine learning (Patel et al. 2010), DejaVu for interactive camera applications (Kato, McDirmid, and Cao 2012), VisionSketch for image processing applications (Kato and Igarashi 2014), Unity for game applications, and web-based live coding environments for music and video improvisation such as Gibber (Roberts et al. 2014) and LiveCodeLab (Della Casa and John 2014). Content authoring and programming environments are colliding with each other. Therefore, it is important to design a content authoring environment that supports both data manipulation and live programming.

This paper introduces TextAlive Online, which allows a user to create Kinetic Typography videos interactively with live programming on a standard web browser. Kinetic Typography is a technique for animating text, which is known to be a more effective way to convey emotional messages compared to static typography. Our system supports the creation of Kinetic Typography videos that show the lyrics of a song synchronized with its audio content accessible on the web. While many live coding environments focus on improvisation of music, our system focuses on providing a platform on which the user can elaborate on creating videos.

Thanks to its web-based nature, the system can retrieve the specified audio content, analyze and estimate vocalized timing of all characters in the lyric text. Then, the estimation results can be cached and corrected collaboratively by several different people. A JavaScript code editor is embedded in the system along with the general graphical user interfaces (GUIs) for video authoring. JavaScript programmers can create, edit, and share algorithms for text animation, which are called "templates," with others. Please note that many designers are now casual JavaScript programmers as well.

In this paper, the core concept behind TextAlive Online is explained. Before we developed TextAlive Online, we had prototyped a Java-based desktop application named TextAlive Desktop that follows the concept and runs on a local machine. The next section introduces the interaction design shared between TextAlive Desktop and Online. Then, the unique feature of TextAlive Desktop with the results of the preliminary user study are briefly explained. This experience led us to believe in the potential impact of making the system web-based. Although rewriting a standalone desktop application to run on a web browser supported by a server-side backend required tremendous engineering effort, TextAlive Online opens up new content authoring possibilities for the web on the web. The following section explains its unique user experience. Then, related work is introduced, followed by a discussion and our conclusion.

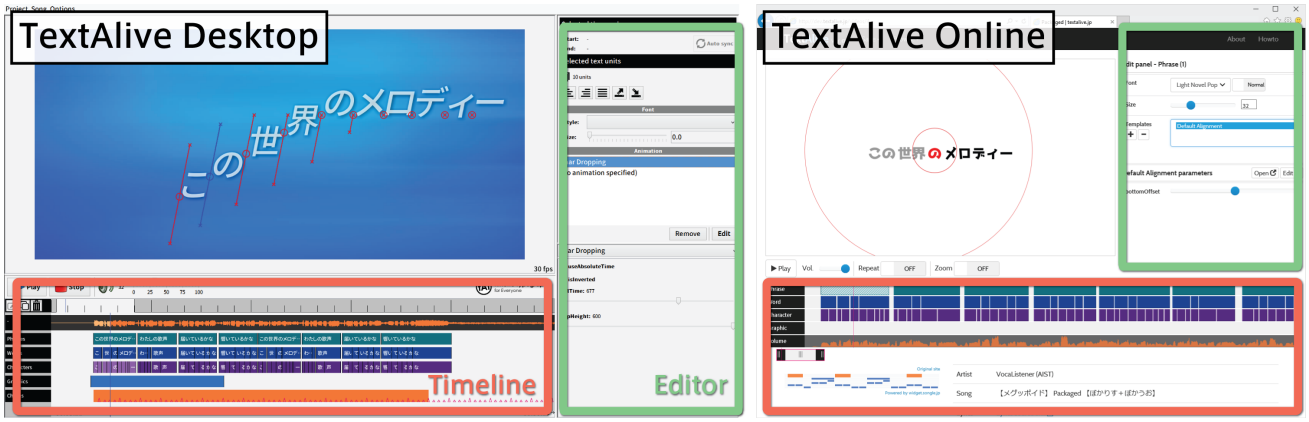


Figure 1: *TextAlive Desktop and TextAlive Online are both live programming environments for creating Kinetic Typography videos that share most of their GUIs. TextAlive Online enables a more collaborative way of content authoring, requires a server-side backend, and raises interesting research questions.*

2. Videos as Pure Functions of Time

Digital videos are usually created with content authoring tools such as Adobe After Effects, Adobe Premier and Windows MovieMaker. A content authoring tool saves information of a video as a project file that has references to external files including images, videos, and audio. When a video project is exported as a rendered video file, the tool generates tens of images per second with a particular resolution by calling rendering routines for every frame.

While ordinary video files have particular resolution in time and space, there are videos that potentially have an infinite resolution in time and space; object positions can be acquired by calculating interpolation between key frames and object shapes can be specified as vector graphics. In other words, a certain kind of videos can be represented by a pure function without any side effects that takes the current time as its argument and returns the rendering result. The demoscene and megademo are prior attempts at creating fascinating videos from code, but their code usually has heavy side effects over time and cannot jump to arbitrary time. Various live coding environments allow the programmer to define a function of time that outputs a visual scene, but most of them focus on improvisation. As a result, the scene is often ephemeral and transformed along time through live coding.

In TextAlive, a kinetic typography video is represented by a collection of pure functions of time with several parameters that can be customized through interactive GUIs. Since lyric text contains repetitions and a video contains multiple parts with similar effects, it is desired that a function for a text unit can be reused for another text unit. Therefore, instead of defining one huge function that renders everything on the video, we introduce a structure of functions – one function per each text unit (a phrase, word, or character) of lyric text. The functions are represented by instances of classes, which we call "templates," with particular parameters. As a concrete example of how these information is stored, please see textalive.jp/videos/20?format=json, which contains links to external resources, including songs, lyrics, and versioned template definitions and their parameters. For each timing, the video image is rendered by calling a method for all of the template instances. Most of the instance methods immediately return without doing anything since they are assigned to text units that are hidden at the specified timing. The rest of the methods modify the rendering parameters of text units to place and reshape them. An example of template definition can be found at textalive.jp/templates/KaraokeColor, which changes the color of assigned characters. If a method call takes too long time, the corresponding template is disabled and will be re-enabled when the programmer updates the template definition.

3. Interaction Design

The user first inputs a song and lyric text to the system. Then, the system estimates the vocalized timing (the starting and ending of the vocalization) of every character in the lyric text. With this information, a playable kinetic typography video is automatically composed and opened in the graphical video editor. Thus, this system provides the user with this concrete example to start working on at the beginning instead of the empty canvas provided by the general video authoring tools.

As shown in Figure 1, the main window of TextAlive consists of 1) a Stage interface that shows the editing video, 2) a Timeline interface specifically designed for the intuitive timing correction of the given text units (phrases, words, and characters), and 3) an Editor interface that allows for choosing and tweaking the templates that define the motion of the selected text units. Casual users and designers can create videos in their style using these GUIs. Though, choosing the text animation from the existing templates limits the flexibility of the resultant videos to a certain extent. TextAlive provides a text-based code editor for live programming of the template definitions to overcome such a limitation.

The text-based code editor is tightly coupled with the aforementioned graphical interfaces. The user can select text units using the Timeline interface, click on one of the assigned templates, edit its motion algorithm, and update the video without needing to pause the music playback. Moreover, part of the Editor interface is dynamically generated from the results of static code analysis of the source code. For instance, when the user annotates a member variable with the comment `@ui Slider(0, 100)`, a slider is populated. It can dynamically change the field values of the template instances assigned to the selected text units. Such GUI components make it easy for the programmer to develop and debug templates iteratively. Moreover, it also helps other people who use the template, including designers and casual users who cannot or do not want to read nor write code.

4. TextAlive Desktop

TextAlive Desktop is the first system that follows the core concept and implements the interaction design described in the previous section. As a preliminary user study, we provided TextAlive Desktop to seven participants with different backgrounds (designers, programmers, and casual users). Then, we asked them to create Kinetic Typography videos using their favorite songs. While the prior publication introduces the system and the study in detail (Kato, Nakano, and Goto 2015), this section summarizes the lessons learned that motivated us to develop the Online version.

4.1. Programming Needs Structured Data

The user needs to wait for the completion of the timing estimation for the song and lyrics pair before they can start creating a new video. If the pair has been previously analyzed, it is possible to reuse the cached data. When the analysis is complete, the estimated result typically contains errors that need to be corrected manually. Such timing correction is important for the Kinetic Typography videos that should be well synchronized with the songs.

The data resources used in the programming should always be clean, structured, and easy to manipulate from the programmer's perspective. The synchronization of a song and its lyrics is considered data manipulation that creates structured data from the raw materials, which is referenced by the template source code. In addition, creating high-quality videos in TextAlive requires an iterative cycle of the data manipulation, programming, and playback. While live programming techniques can reduce the programming burden, this kind of data manipulation might remain as the bottleneck in the iterative cycle. Therefore, it is considered important to make *integrated design environments* in which the user interfaces are tightly coupled for the data manipulation and live programming.

4.2. Designers Want to Design, Programmers Want to Code

TextAlive Desktop is the hybrid of an interactive graphical tool and a live programming environment. We assumed that many designers are now capable of writing code, and thus, can benefit from the live programming part of TextAlive. However, during the user study, we observed a clear tendency for the designer participants (and, of course, casual users without prior knowledge of programming) to hesitate opening up the code editor. They wanted more predefined and ready-for-use templates. In addition, combining multiple templates for a single text unit seem to be difficult for the users. It is because the results are often unexpected and there is no hint to look for visually pleasing combinations.

On the other hand, the programmers could write various algorithms for text animation but were not always interested in creating a complete video. While all the participants welcomed the capability of the integrated environment, collaborations between people with different backgrounds are found to be the key to making the most use of it.

5. TextAlive Online

Given the lessons learned from the deployment of TextAlive Desktop, we built TextAlive Online, a web-based application with which people can collaboratively author kinetic typography videos. The resulting system is available at textalive.jp. While the main interface consists of almost identical user interface components to the desktop version (Figure 2), there are online-specific features and improvements as explained in this section.

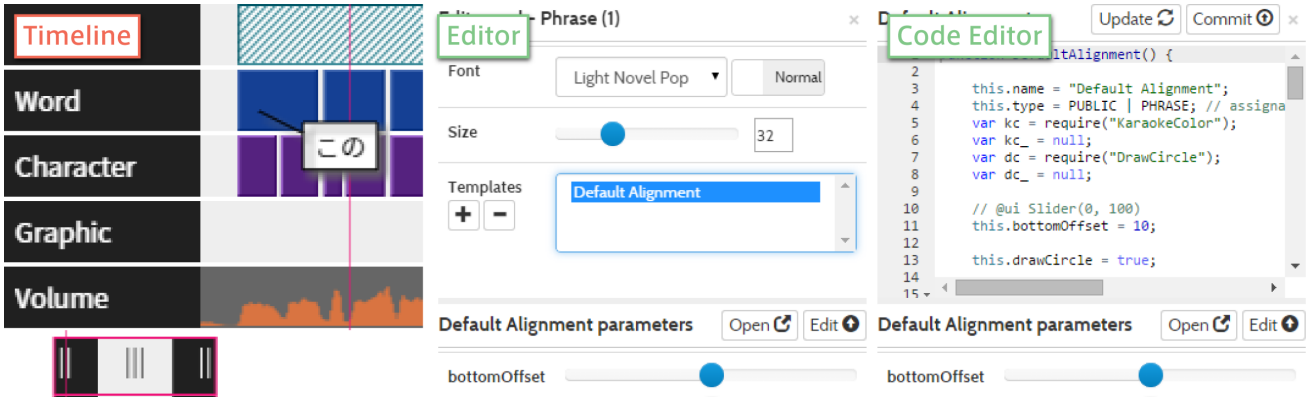


Figure 2: *TextAlive Online provides three user interfaces for editing Kinetic Typography videos.*

5.1. Everything is Shared and Version-Controlled

With TextAlive Online, the user just needs to search for and choose a song from one of the music streaming services at textalive.jp/songs (currently supporting SoundCloud, YouTube, a MP3 file on the web, and other several music streaming services) and provide its lyrics by pointing to a URL (currently supporting a TXT file and some websites with specific HTML formats) to start creating a Kinetic Typography video. If the song was not previously analyzed, the user is told to wait for several minutes. Otherwise, the user can instantly see an automatically-generated Kinetic Typography video.

Timing information about the vocalization of phrases, words, and characters can be corrected at any time, even during video playback. Then, the user can commit the timing information to share it with other users. In this way, TextAlive Online supports creating structured data collaboratively that is used for programming text animation.

A Kinetic Typography video on TextAlive Online consists of the following information: 1) URLs to the song and lyrics, 2) the structure of the lyrics, which is a tree of the text units (phrases, words, and characters), and 3) a list of assigned templates and their parameter values for each text unit. These set of information is represented by a JSON object and is version-controlled on the website. In fact, not only the videos but also the estimated timing information, correction history, and template definitions are all version-controlled, enabling to fork derivative work and to rollback flawed edits. Saving a new template definition is as easy as clicking the "Commit" button beside the update button that locally updates the definition (Figure 2, Code Editor).

5.2. Some Templates Make Use of Other Templates

In the Desktop version, the user is allowed to assign multiple templates to a text unit for flexible design of kinetic typography. For instance, a template for fading in and out text and a template for moving text can be assigned to the same phrase to combine both effects. However, it seemed difficult for the user to find a good combination of multiple templates and the feature was not used frequently. To simplify the interface, TextAlive Online currently limits the number of templates that can be assigned to a text unit to one. In addition, to maintain the reusability of templates, it allows the programmer to import and use other templates in a template definition.

textalive.jp/templates/DefaultAlignment is one example that calls a method of another template. The `KaraokeColor` is dynamically loaded with the `require(templateName)` function (which is similar to the popular RequireJS library), instantiated, and its `highlight` method is called on for coloring each character.

6. Related Work

This section mainly covers the prior work in the literature for human-computer interaction and live coding.

6.1. Tools for Content Authoring

Various tools have already been proposed for creating Kinetic Typography videos, such as toolkits for programmers (J. C. Lee, Forlizzi, and Hudson 2002) and GUI applications for end-users (Forlizzi, Lee, and Hudson 2003). However, most of them aim at augmenting text-based communication, and none of them provide support for synchronizing text with audio. Therefore, they rarely have the timeline interface necessary to seek the time and edit timings precisely.

More general video authoring tools are equipped with a timeline interface such as Adobe AfterEffects. They typically contain scripting engines in which the user can write code (called "Expression" in AfterEffects) to control the property changes over time. Although, it is just one way of specifying the visual properties, and thus, do not output video as a clean, structured object. There are also tools for interactive VJing, which is creating videos in real time synchronized with music. For instance, Quartz Composer and vvvv have been used for such purposes. Other examples are introduced in the next subsection. While the current implementation of TextAlive focuses on authoring videos and does not support improvisation of visual content, it is an interesting future work.

Live coding environments for music typically focus on improvising music and the resulting performance can only be recorded as static audio or video. On the other hand, Threnoscope (Magnusson 2014) is a live programming environment to describe musical scores. It is similar to TextAlive in that both help the user to create structured data (score or video) which can be modified later, either by the original author or other people.

6.2. Live Programming of Immediate Mode GUIs

TextAlive renders each frame of the video by repeatedly calling on the `animate(now)` method of all template instances. The programmer's code is responsible for building the scene from scratch for every frame. This type of user interface rendering is called immediate mode GUIs (related discussion at Lambda the Ultimate), while the frameworks for retained GUIs such as Java Swing expose APIs that essentially manipulate a scene graph composed of GUI widgets with hidden state information inside.

Typical GUI applications adopt the retained GUIs since they are considered convenient, but they have certain downsides such as difficulty in tracing the state transitions. This is particularly problematic for live programming. As a result, many of the existing live programming environments only support immediate GUIs. Notable examples that run on a standard web browser include Gibber (Roberts et al. 2014), LiveCodeLab (Della Casa and John 2014) and TouchDevelop (Burckhardt et al. 2013).

6.3. Live Programming with Parameter Tuning Widgets

Editing a text-based source code is often cumbersome since the programmer needs to type the parameter values many times to find the best value in order to make the full use of live programming. Providing sliders for this purpose is a common practice in this field as seen in recent Processing extension.

Impromptu (Sorensen and Gardner 2010), Overtone (Aaron and Blackwell 2013), and Threnoscope (Magnusson 2014) are live coding environments for music which can bind parameter values not only to GUI sliders but also to sliders on a physical control board. Unity is a game engine that is capable of instantiating widgets without killing the entire application. VisionSketch (Kato and Igarashi 2014) takes a similar approach and allows the programmer to draw shapes on the input images to interactively change the parameters passed to the computer vision algorithms.

7. Conclusion

We propose TextAlive Online, an online content authoring environment with support for live programming. It is the hybrid of a content authoring tool and a live programming environment, each of which has been separately evolved and is good at interactive data manipulation and intuitive programming, respectively. It is the successor of TextAlive Desktop, which is a desktop application and whose user study revealed the need for a web-based backend that features cooperative data manipulation, live programming, and video authoring.

The current implementation allows for embedding a created video into a web page, as demonstrated in the top page. Kinetic Typography is applicable not only for songs and lyrics but also for more general text information. It might be interesting to allow external developers to import motion algorithms from TextAlive Online and use them for animating text on their web pages. In addition, our future work includes improvisation of kinetic typography videos that enables a "text jockey" who creates text animation on-the-fly similar to a disc jockey and visual jockey.

While there have been proposed many live coding environments that output a visual scene as well as audio, existing environments typically put emphasis on improvising the scene from scratch in front of the audiences. Such environments are optimized for real-time live performance that is ephemeral and can only be recorded as videos with a static resolution in time and space. On the other hand, TextAlive Online is a platform on which the user can create and play videos that are essentially a collection of pure functions of time and potentially have an infinite resolution in time and space. The videos are always rendered on-the-fly by calling the functions. Since their information is kept structured, anybody can create derivative work from them, either by GUIs or by writing code. We believe that there is much potential in live coding techniques not only for live performance but also for collaboratively elaborating on creating various kinds of media content.

8. Acknowledgements

This work was supported in part by JST, CREST. TextAlive Online uses Songle Widget API for audio playback, whose main developer is Takahiro Inoue. It also relies on Songle web service for retrieving vocalized timing of lyric text and other information regarding the audio track, whose main developer is Yuta Kawasaki.

References

Aaron, Samuel, and Alan F. Blackwell. 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 & Design*, 35–46. FARM '13. New York, NY, USA: ACM. doi:10.1145/2505341.2505346. http://doi.acm.org/10.1145/2505341.2505346.

Burckhardt, Sebastian, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. "It's Alive! Continuous Feedback in UI Programming." In *Proc. of PLDI 2013*, 95–104. PLDI '13. Seattle, Washington, USA. doi:10.1145/2491956.2462170. http://doi.acm.org/10.1145/2491956.2462170.

Della Casa, Davide, and Guy John. 2014. "LiveCodeLab 2.0 and Its Language LiveCodeLang." In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, 1–8. FARM '14. New York, NY, USA: ACM. doi:10.1145/2633638.2633650. http://doi.acm.org/10.1145/2633638.2633650.

Forlizzi, Jodi, Johnny Lee, and Scott Hudson. 2003. "The Kinedit System: Affective Messages Using Dynamic Texts." In *Proc. of CHI 2003*, 377–384. Ft. Lauderdale, Florida, USA. doi:10.1145/642611.642677. http://doi.acm.org/10.1145/642611.642677.

Kato, Jun, and Takeo Igarashi. 2014. "VisionSketch: Integrated Support for Example-Centric Programming of Image Processing Applications." In *Proceedings of the 2014 Graphics Interface Conference*, 115–122. GI '14. Toronto, Ont., Canada, Canada: Canadian Information Processing Society. http://dl.acm.org/citation.cfm?id=2619648.2619668.

Kato, Jun, Sean McDirmid, and Xiang Cao. 2012. "DejaVu: Integrated Support for Developing Interactive Camera-Based Programs." In *Proc. of UIST 2012*, 189–196. Cambridge, Massachusetts, USA. doi:10.1145/2380116.2380142. http://doi.acm.org/10.1145/2380116.2380142.

Kato, Jun, Tomoyasu Nakano, and Masataka Goto. 2015. "TextAlive: Integrated Design Environment for Kinetic Typography." In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 3403–3412. CHI '15. New York, NY, USA: ACM. doi:10.1145/2702123.2702140. http://doi.acm.org/10.1145/2702123.2702140.

Lee, Johnny C., Jodi Forlizzi, and Scott E. Hudson. 2002. "The Kinetic Typography Engine: an Extensible System for Animating Expressive Text." In *Proc. of UIST 2002*, 81–90. Paris, France. doi:10.1145/571985.571997. http://doi.acm.org/10.1145/571985.571997.

Magnusson, Thor. 2014. "Improvising with the Threnoscope: Integrating Code, Hardware, GUI, Network, and Graphic Scores." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, edited by Baptiste Caramiaux, Koray Tahiroglu, Rebecca Fiebrink, and Atau Tanaka, 19–22. London, United Kingdom: Goldsmiths, University of London. http://www.nime.org/proceedings/2014/nime2014_276.pdf.

Patel, Kayur, Naomi Bancroft, Steven M. Drucker, James Fogarty, Andrew J. Ko, and James Landay. 2010. "Gestalt: Integrated Support for Implementation and Analysis in Machine Learning." In *Proc. of UIST 2010*, 37–46. doi:10.1145/1866029.1866038. http://doi.acm.org/10.1145/1866029.1866038.

Roberts, Charles, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. 2014. "Gibber: Abstractions for Creative Multimedia Programming." In *Proceedings of the ACM International Conference on Multimedia*, 67–76. MM '14. New York, NY, USA: ACM. doi:10.1145/2647868.2654949. http://doi.acm.org/10.1145/2647868.2654949.

Sorensen, Andrew, and Henry Gardner. 2010. "Programming with Time: Cyber-Physical Programming with Impromptu." In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 822–834. OOPSLA '10. New York, NY, USA: ACM. doi:10.1145/1869459.1869526. http://doi.acm.org/10.1145/1869459.1869526.

Performances

Performance abstracts

1. Concert A, Monday

CePRA special session with instrumentalists Anne Veinberg and Greta Eacott.

Pianist Anne Veinberg is active as soloist, improvisor and ensemble player in Europe and Australia, and will take part in four collaborations in the opening performance session of ICLC. With her keen interest in contemporary music, Anne regularly collaborates with composers and has premiered numerous works. She is particularly interested in exploring works for piano and live electronics/live coding and the more theatrical contemporary piano repertoire. As a collaborative pianist, Anne is a member of Duo Kolthof/Veinberg, Duo Neshome, Duo H|A, Ensemble SCALA (microtonal music) and Off<>zz (livecode/piano).

Classically trained, Anne studied at the Sydney Conservatorium High School, obtained her Bachelor in Music from the University of Melbourne, and completed her Master of Music at the Conservatory of Amsterdam. In September 2015 she will begin her PhD research; "Keyboards Unite: Exploring chamber music possibilities between a pianist and live coder" as part of the DocArtes program in Ghent.

Greta Eacott's biography is not available for the compilation of the preliminary proceedings, please refer to the website for details.

1.1. Gopher - Marcel Wierckx and Anne Veinberg

Gopher is the second of a series of compositions for Disklavier that are to be performed as a duet between pianist and live coder. In this composition the system is used to generate notes on the piano which are performed as harmonics by the pianist. Pitch and rhythm are treated as separate entities, enabling the live coder to generate complex shifting patterns using relatively simple commands. Gopher is inspired by the fairground game in which the player hammers the heads of plastic gophers as they pop up out of holes in a random sequence. The pianist plays harmonics on the strings of the piano in much the same way, and in that sense the piece can be seen as a game between the pianist and live coder.

Marcel Wierckx is a sound and video artist who creates music and image for concert, film, theatre and dance. He recently created the music and sound design for Chunky Move's critically acclaimed dance production An Act of Now which was awarded the Age Critic's Award in 2012. His work is performed at major festivals and conferences around the world, including the International Computer Music Conference, Live.Code.Festival, ISCM World Music Days, and STRP. Wierckx is currently lecturer in Live Electronic Music at the Amsterdam Conservatory of Music, and Composition and Software Design at the Utrecht School of Music and Technology. http://www.LowNorth.nl

1.2. Improvisation for Pianist (Disklavier) and Live Coder - Juan A. Romero and Anne Veinberg

The improvisation consists of mutual feedback using the Disklavier as an acoustic and mechanical interface between the pianist and the live coder. Phrases and notes will be recorded and algortihmically transformed and played back throught the same device interfering and interacting with the player. Some sound synthesis will be also live coded to expand and adapt the sound of the piano and the playing style of the perfomer.

Juan A. Romero was born 1982 in Medellín, Colombia. He is member of the live coding band Benoît and the Mandelbrots and the audiovisual performance duo Ganzfeld. At the moment he is a doctoral candidate and research assistant at the IMWI (HfM Karlsruhe).

1.3. Encoding the Marimbist - Thor Magnusson and and Greta Eacott

In this performance the marimbist Greta Eacott will perform real-time generated musical notation in the form of code. The coding language called CMU (Code Music Notation) is a notational language for human interpreters and thus different from traditional CUI's (Code User Interfaces) written for machine interpreters. CMU is an object oriented programming language with a C-family syntax and dot notation, also supporting functional approaches, such as first class functions and recursion.

Thor Magnusson is a live coder and has created the ixi lang and Threnoscope live coding systems. He lectures in Music and convenes the Music Technology programme at the University of Sussex, Brighton.

Greta Eacott is a composer, percussionist and marimba player living in Copenhagen.

1.4. Type a personality - Nick Collins and Anne Veinberg

Type a Personality is a score for pianist with interwoven typing as well as live coded synthesis engine electronics part. The control sequence for the electronics part is a direct result of Anne's scripted typing; the inevitable errors in typing certain keys under time pressure, and the inter-key time intervals, directly set the state and memory values of the audio engine. The exact text to type out is different for each performance, but watch out for a possible extract from the TOPLAP manifesto.

The work is a da capo aria, where the second play through of the material involves additional embellishment. During Type a Personality's performance, audio is gradually collected for the concatenative synthesis, new audio in turn triggering output from that engine based on the instruction code program typed out. Indeed, multiple synthesis/processing state engines can be driven from the combination of audio, typed letters and typing timing collected, providing a complex mixture of acoustic and electronic, machine listening and non-programmer live coding.

NICK COLLINS is Reader in Composition at Durham University. His research interests include live computer music, musical artificial intelligence, and computational musicology, and he has performed internationally as composer-programmer-pianist and codiscian, from algoraves to electronic chamber music. As a composer, he investigates new possibilities in autonomous interactive music systems performing with acoustic musicians, pure computer works such as microtonal generative music, and even the occasional piano sonata (http://composerprogrammer.com/music.html)

1.5. Off<>zz Live - Off<>zz

Off<>zz is a laptop and piano/toy piano duo featuring Felipe Ignacio Noriega and Anne Veinberg respectively. All our performances are improvised and our style is guided by our search for bridging the gap between instrumental and live coding music making. Felipe codes from scratch in supercollider and this in part guides the speed of musical development in our improvisations but is altered due to the presence of an acoustic instrument, namely the piano. We move between blended piano/computer soundscapes, to vibrant grooves, contrary expressions which eventually morph together and musical explosions.

Off<>zz is unique in its contribution to the live coding scene as an ongoing project which explores the possibilities between live coding and acoustic instrumental performance. There are many differences, advantages and disadvantages between music making via live coding and music making via piano. We aim to utilize the good of each and mask the weaknesses through supporting each others musical actions. Our classical background leads to a style that moves between avantgarde, minimal, post modernist and electro groove music.

Felipe and Anne have been collaborating for several years, from more conventional contemporary music works to Off<>zz, a piano and live-coding duo. They have performed at various festivals including Smaakt naar Muziek (Amsterdam), La Escucha Errante at ZAWP (Bilbao) and undertook a residency at STEIM in November 2013.

Felipe Ignacio Noriega is a composer and laptop artist born in Mexico City and now living in Amsterdam, The Netherlands. He collaborates for various settings where a common research is the importance of movement, theatricality, and visual fantasy achieved through the aid of cheap, domestic technologies. He is co-founder of Amsterdam Chamber Opera, Bo

is Burning, Panda Zooicide and DTMF signaling - collectives that explore music, opera, theater and new methodologies of interdisciplinary collaboration and inter-communication.

Felipe Ignacio received the 1st prize of the composition competition from the Tromp Percussion Festival Eindhoven in 2010 and the Publieksprijs of the 2scoreMusicTheater competition in 2012 in Enschede, for his first opera LAKE. He graduated Cum Laude from the Masters in composition at the Conservatorium van Amsterdam in 2013 and is compostition winner of the Young Artist Fund Amsterdam 2015. www.felipeignacio.info

2. Concert B, Monday

2.1. BEER - Birmingham Ensemble for Electroacoustic Research

We propose to perform two live-coded works, Swarm 1, by Norah Lorway, and SwitchBlade, by Scott Wilson + BEER. The former piece involves improvising through a loosely pre-determined structure using supplied templates and materials, which are then modified and extended by the ensemble. In the latter each musician live codes 3 layers of different types, which are chosen and combined using a central control algorithm, itself live-coded.

BEER, the Birmingham Ensemble for Electroacoustic Research, was founded by Scott Wilson in 2011 as a project to explore aspects of realtime electroacoustic music making. Particular interests include networked music performance over ad hoc wi-fi systems, and live coding (programming music in real time using algorithms that can be altered while they are running). In keeping with post-free jazz developments in improvisation (e.g. Zorn, Braxton), we create structures in software that impose limitations and formal articulations on the musical flow (with networked software systems serving as intervention mechanism / arbiter / structural provocateur par excellence). Musical influences run the gamut from Xenakis to Journey. Past and current members have include Konstantinos Vasilakos, Norah Lorway, Tim Moyers, Martin Ozvold, Luca Danieli, Winston Yeung, and Scott Wilson.

2.2. The Augmented Live Coding Orchestra - Fabrice Mogini

This is a performance in 'live composing', using live coding and live algorithms to:

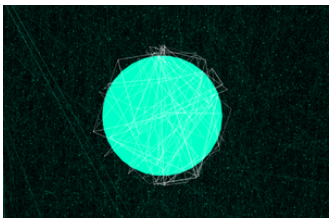
- control a library of orchestral samples (the Live Coding Orchestra).
- control and process pre-written audio which was created during previous coding session with a similar setup.

The aim is to start with an original note series, based on the letters of a composer's name; then develop and orchestrate this material into a live composition, taking into account the audio played at the same time, which will be re-harmonised, arranged and orchestrated, thus augmenting the audio with the live coding orchestra and the other way round, augmenting the orchestra with the audio.

Fabrice was born in Cannes, France. First attracted to Classical, Jazz and progressive Music, he encountered generative music and algorithmic composition in 1998, while studying on the London Sonic Arts course where he learnt SuperCollider. Fabrice has been involved in Live Coding since 2002, originally as a means to have better control over algorithmic processes and as an attempt to link improvisation and computer-aided composition. Fabrice is a lecturer in Music based in London and also works as a freelance Music composer and producer specialising in music for film and television.

3. Club A, Monday

3.1. Warp Hole Sounds - Warp Hole Sounds, AV Torres



"Warp Hole Sounds" is an algorithmic exploration and an audio visual experience of the non-trivial phenomena, which in theory is possible inside these unseen and unexperienced corners of our universe. It pursuits a more embodied understanding of the behaviour of the physics inside or close to these bizarre spaces.

Through computer algorithms, Andres simulates strong gravitational fields which evolve in time by being nourished by particles (light, matter, energy and sound). The illusive infinite freedom from algorithms allow to simulate and to tweak several conditions from these theoretical places, provoking behaviours that some times cannot be computed or previously estimated. A series of sonic and visual experiences arise from the never ending evolution of these simulated wormholes.

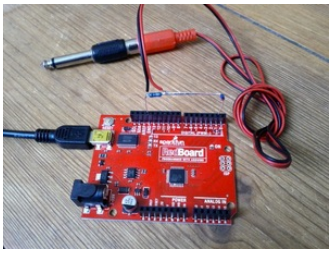
Born in Mexico City in 1985. Andres Villa Torres has a Design background and a long relationship with Music. He likes to explore the borders between the digital, the material and the "real" serving himself from diverse interactive and non-interactive media, technology and algorithms.

3.2. Feedforward -an electric guitar and live code performance - Alexandra Cárdenas

Feedforward is a system, a cyber instrument composed by performer, electric guitar and laptop. The guitar sounds trigger previously written code and will make the laptop live code itself. The performer reacts to the choices of the computer. The initial settings of the system are a composition that is open to improvisation for both the performer and the laptop. Synths created in SuperCollider will analyse the sound of the guitar and trigger the autocode function of IXI lang. This will be an interaction of preexistent code and new born code, of preexistent sounds and new born sounds. Synthesis and transformations of the electric guitar sounds will make a piece that emerges from the electroacoustic tradition merging it with the noise tradition through an automatic live coding generative environment.

Composer and improviser of music ranging from writing orchestral works to improvising with electric guitar and laptop. Using mainly the software SuperCollider to turn her computer into another musical instrument, her work has focused recently in creating pieces with Live Electronics and exploring the musicality of coding and the algorithmic behaviour in the music. An important part of this exploration is the practice of live coding. Currently she lives in Berlin, Germany and studies the masters Sound Studies at the University of the Arts. www.tiemposdelruido.net

3.3. chain reaction - Fredrik Olofsson

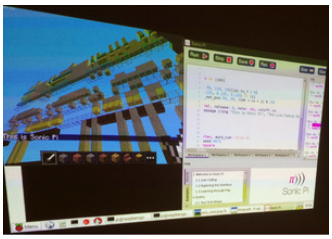


In chain reaction I write small programs for Arduino boards and make them send useless serial data directly to the speakers. Depending on the type of data being sent, at what baud-rate and how the improvised little C programs flow, I get different rhythms, melodies and a variety of noises.

Fredrik Olofsson is educated in music composition at the Royal Music Academy in Stockholm, Sweden. Ever since his graduation 15 years ago, he has worked with developing software and electronics for interactive installations. He has also travelled the world performing electronic music and live video - both solo and in different constellations. Currently, alongside commissions, residencies, collaborations, workshops and performances, he is doing contract work for the project rhyme.no in Oslo, Norway and teaches computational art at Universität der Künste in Berlin, Germany. http://www.fredrikolofsson.com

3.4. Sonic Miner - Sam Aaron and Alan Blackwell

Sonic Pi has become well-established as an educational platform in which live-coded music can be used an element of early computer science education in schools. The Sonic Pi Live and Coding project has also demonstrated the potential of live coding, using Sonic Pi, as a contribution to the school music curriculum. Recent releases of Sonic Pi have included support for controlling the external API of Minecraft: Pi Edition - a version of the popular Minecraft open world game which runs on the low cost Raspberry Pi computer, and also provides interfaces for code-based scripting. Although Minecraft: Pi Edition was originally designed to be controlled from a Python interpreter, the same API can now be live-coded from within Sonic Pi.



This performance is an initial exploration of the potential for Minecraft to be used in an algorave context. Sam Aaron, developer of Sonic Pi, maintains a practice-led research discipline in which the educational tools that he uses in classroom teaching are the same tools that he uses in public performance. The latest releases of Sonic Pi (also available for Windows and Macintosh) therefore include the latest updates that have been developed to support his own performance practice. This philosophy is consistent with music and arts education practices, in which it is considered beneficial to provide professional quality materials even to students, wherever possible. In keeping with this philosophy, we are therefore exploring the extent to which Minecraft itself can become a performance instrument for algoraves. Sam will be coding algorave music, while Alan Blackwell attempts to produce dance visuals through algorithmic manipulation of the Minecraft world. We hope that we can achieve networked synchronisation between music and the Minecraft world, to a degree that offers an immersive dance experience. Failing that, we will offer the audience direct access to the Minecraft view and/or avatar controls via wireless controllers, allowing them to engage as co-performers in virtual space.

Sam Aaron is a live coder, computer science educator, and developer of the popular Sonic Pi live coding environment. In addition to his online audience and performances with groups including Meta-eX, Poly_core and others, he has extensive previous experience as a computer science researcher, and as developer (with Jeff Rose) of the Overtone live coding language. His current research and participation in ICLC is supported by a generous donation from the Raspberry Pi Foundation. Alan Blackwell is a researcher in human-computer interaction, with a focus on languages for visualisation, digital media and end-user development. Everything he knows about Minecraft was learned from his daughter Elizabeth, but he has managed to write several academic papers on the topic! His participation in ICLC is supported by a grant from the Boeing Corporation.

3.5. Auto - Canute

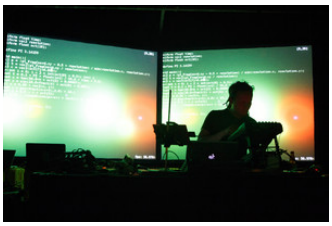
Yee-King plays a set of digital v-drums which trigger percussion sounds and longer, synthesized sounds. McLean livecodes in the Tidal language, generating polyrhythmic patterns, melodic synthesizer lines and some sampled vocal patterns. They cover a range of rave inspired styles including industrial techno, drill and bass, dubstep and so forth, with occasional bursts of noise and free improv. The performance will be fully improvised, and will experiment with introducing autocoding into collaboration between percussionist and coder.

Canute is a performance collaboration between Matthew Yee-King and Alex McLean, exploring improvisatory practice through programming languages and percussion. Their performances bridge techno, drill n bass, and experimental free jazz improv.

Matthew Yee-King is a computer music composer and performer, and postdoctoral research fellow at Goldsmiths College, where he works on a project to develop a social network for music learners. He has worked in a range of musics from the use of agent based live improvisers through more straight ahead electronic music to free improv with jazz players. He has performed live internationally and has recorded many sessions for BBC Radio. In the past his solo music has been released on electronic music imprints such as Warp Records and Richard James' Rephlex Records. Past collaborators include Jamie Lidell, Tom Jenkinson (Squarepusher), Finn Peters and Max de Wardener. http://www.yeeking.net/ http://www.gold.ac.uk/computing/staff/myeeking/

Alex McLean is a musician and researcher based in Yorkshire, UK. Alex is Research Fellow in Human/Technology Interface and Deputy Director within ICSRiM, University of Leeds. He created Tidal, the mini-language for live coding pattern. He performs internationally as a live coder, including as one third of the long-lived band Slub with Dave Griffiths and Adrian Ward. He coordinates promoters Algorave and algorithmic record label ChordPunch with Nick Collins, and has co-organised around 80 Dorkbot electronic art events in Sheffield and London. Alex completed his PhD thesis "Artist-Programmers and Programming Languages for the Arts" at Goldsmiths in 2011. He recently released Peak Cut EP on Computer Club, in the form of a bootable USB key. http://yaxu.org/ http://music.leeds.ac.uk/people/alexmclean/

3.6. Live Coded Controllerism - Luuma



The piece will be a performative design of a new modular instrument. It will start with a blank code page, and a diverse set of controllers. Over the course of the piece, I will live-code the sound and controller mappings for an Algorave style improvisation, bringing in new controllers from the collection as the piece develops. As the set progresses, an instrument will take form, and the performance will move from coding to controllerism. The piece harnesses the flexibility of livecoding to allow the design of an instrument to evolve with the music. The piece will highlight the interplay between instrument, mapping, sound and algorithm design.

Chris Kiefer is a computer-musician and researcher in musician-computer interaction. He's interested in designing digital music instruments using multiparametric sensing techniques and in improvising electronic music. Chris performs as Luuma, and has recently been playing at Algoraves with custom made instruments including malleable foam interfaces and touch screen software. His music has been released on algorithmic music label Chordpunch.

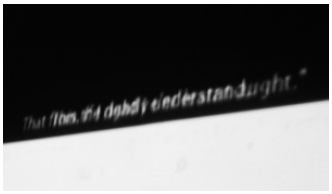
4. Concert C, Tuesday

4.1. living sound - Dragica Kahlina

The performance is a combination between live coding and instrumental music. The sound journey is improvised, layered and has a strong focus on the timbral aspect of music. Live coding works within a prepared, but dynamic framework that gives the performer the freedom to change all aspects of the music spontaneously. The instrument used is an Eigenharp Alpha an electronic controller with 132 buttons that act as 3-axes joysticks, a breath controller and 2 pressure stripes. The OSC (open sound control) data from the instrument is used to communicate with the live coding environment on the laptop. The live coding happens in Overtone, a music library that interfaces Clojure with a Supercollider server. But the environment is more than that, it builds a living game world behind the scenes. In this world sounds are living creatures with an AI-mind of their own. The musician doesn't necessary play notes anymore, but seeds the sound creatures and interacts with them. Playing an instrument becomes playing with reactive AI-organisms in a sound world. The environment is built, maintained and changed by live coding and influences and changes the type of sound creatures that can be created and the music the audience hears. Granular synthesis fits into this really well and is used for most of the sound.

Dragica is a sound artist interested in the intersection between programming, music and science. She uses mostly Clojure with Overtone, Supercollider or Max/MSP played with a Eigenharp to create her improvised electronic sound journeys. She composes music and designs sounds for games. Dragica has worked as a professional game programmer specialized on AI. Her background is in theoretical physics.

4.2. To code a dadaist poem - Sean Cotterill



This performance is an evolution of a concept I explored for Rodrigo Velasco's event 'on-the-fly codepoetry' held in January 2015 (http://cargocollective.com/onfcopoe/otfCp00). For the performance I will be using live coding techniques to create cut-up poetry and sound on the fly, reminiscent of the process described by Tristan Tzara in his work 'To make a dadaist poem'.

I will be accessing, splitting up and re-combining on the fly a large bank of poetry in the public domain using SuperCollider (and displayed using Processing). The lines and snippets of poetry will be sequenced using random and probabilistic programming techniques to form an evolving, shifting new poem, which will develop throughout the duration of the performance. During the performance I will also improvise live-coded sound using SuperCollider derived from the evolving poem, teasing out semantic and mimetic relationships between sound and text, and in turn adapting the sequencing and usage of poems according to the development of the music.

The performance has no set structure, I will begin sequencing and sound from scratch, and improvisation is the key methodology underpinning the performance. I will, however, have a Processing sketch prepared as a 'blank canvas' to receive sequenced poetry as strings via OSC in a flexible and dynamic manner, as well as displaying the sequenced poetry in an aesthetically discerning manner.

I will also assemble banks of poetry and accompanying OSC addressing in advance of the performance SuperCollider and use the modular SCVim editor and environment with completion and snippet addons in order to facilitate fast and efficient improvisation during the performance, developing an improvised text and music performance ecology from the ground-up in real time, as quickly as possible.

Sean Cotterill is a musician, digital artist and live coder from Newcastle, UK, working in and around fields of experimental sound, dance music, light, hardware & software building, interactivity and inter-disciplinary practice.

Sean is a recent graduate of Newcastle University's International Centre for Music Studies, where he studied improvisation, popular music, sound art and electronic music performance and computational music techniques. He is involved in a number inter-disciplinary and musical projects such as coï¿¥i¾¡pt, Mutual Process, RE/CEPTOR, State of Grace and The Midnight Doctors as well as delivering digital arts commissions both in the UK and internationally for organisations such as ZENDEH and ISIS Arts.

4.3. Wezen - Gewording - Marije Baalman

Wezen - translation:be, character, creature, essence, gist, nature, being, intelligence, also orphans (2012-4; work in progress)

Wezen is a solo music-theatre work where I try to explore the relationship between theatrical gesture and musical instrument, between the body and the sonic world.

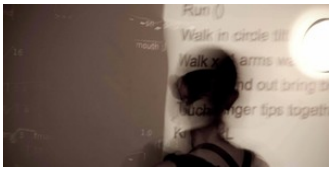
Gewording (Becoming) is the first performance version where the link between physical and sonic gesture is explored during a live performance, combining movement of the body and live coding.

Marije has a diverse background in physics, acoustics, electronic music and sound art, and performance. In her artistic work Marije moves between live performance, livecoding, and interactive installations involving light and sound. Her focus is on composing of behaviours and interaction modalities, creating processes driven by sensor data, rather than fixed sound tracks. She has collaborated with various people, amongst which Alberto de Campo, Chris Salter, Michael Schumacher, Attakkalari Dance Company and Workspace Unlimited.

5. Concert D, Tuesday

5.1. HTB2.0 - Kate Sicchio

This dance piece is an exploration of live coded electronics and improvisational movement. A dancer wears a custom garment of haptic actuators. These actuators are programmed real-time via OSC to 'buzz' on the right and left sides of the dancer to indicate which side of the body the dancer will move. The score is being live coded by choreographer while the dancer is responding to the haptic feedback. This piece explores live coding of bodies and movement as output rather than a sonic or visual output as found in many live coding performances.



Dr. Kate Sicchio is a choreographer, media artist and performer whose work explores the interface between choreography and technology. Her work includes performances, installations, web and video projects and has been shown in Philadelphia, New York City, Canada, Germany, Australia, Belgium, and the UK at venues such as Banff New Media Institute (Canada), WAX Brooklyn (New York), FoAM (Brussels) and Arnolfini Arts Centre (UK).

5.2. End of buffer - Repl Electric

My performance is a mix of improvisation and composed music and visuals. The primary programming language used is Clojure a LISP based language running on the JVM. The visuals are written with the OpenGL Shading Language and the music with Overtone a client to the SuperCollider sound engine and finally a library I created myself called MUD optimizing immediacy of expression. All of this is bound together using the Emacs editor, taking a common tool used widely for programming and emphasizing its power and feedback in coding live.

There is a deep connection between sound and visuals, experimenting with different ways of visualizing frequencies, beats and time. My visual style is focused on simple 2d geometric shapes, playing with peoples assumption around the solidity of these, progressively deforming with the flow of the music.

Abstraction is used in the code to create not just a useful domain specific language but plays with themes that inspired the synths and visuals. My musical style is ambient with a focus on simple handcrafted synths and multi voice progressions. Taking the culture of open source to music all the performance code, and libraries used are open for all to see and change: https://github.com/repl-electric/cassiopeia

Joseph Wilk performs as Repl Electric. Binding light, sound and poetry to the fingertips. Taking the culture of open source, the code is open for all to see and play: http://github.com/repl-electric

6. Concert E, Wednesday

6.1. Slamming Street 01100110 - Afrodita Nikolova, Sam Aaron and Alan Blackwell

Slamming Street 01100110 is an experimental performance collaboration involving poetry and live coding. Our goal is to explore the borderlands between computation and human experience; between media and algorithms; and between structures and interpretations as a creative intercultural encounter. Using Linton Kwesi Johnson's dub poem 'Street 66' as a shared starting point, Sam Aaron, Afrodita Nikolova and Alan Blackwell have worked together to understand their experiences as improvisers and boundary-crossers. Encounters with authority, as framed by national cultures on the international stage, suggest resonances between 1970s Brixton, 1990s New Zealand, and the Macedonia of today. In the live coding context, the introduction of poetry performance (Nikolova is a Macedonian poetry slam champion) raises challenging questions about the nature of text, and the experience of voice, when juxtaposed with sonic structures, sampled sound and processed media imagery.

Technical set-up is as follows: Sam performs with Sonic Pi, the live coding environment that he has developed with support from Raspberry Pi Foundation, in a recent release augmented with support for live audio input that is used here for processing Afrodita's voice. Alan performs with Palimpsest, a visual language that he has created for live algorithmic image transformation. Sam and Alan's previous collaboration as "The Humming Wires" was originally formed to investigate the ways in which live coding undermines considerations of copyright in mashup and homage, through improvised re-codings of Red Right Hand by Nick Cave and the Bad Seeds. Subsequently used as the basis for networked algorave performance, use of the same systems (with some performance-oriented enhancements) in Slamming Street 01100110 gives us an opportunity to push genre and media boundaries through working with text and human voice.

- Afrodita Nikolova - text/voice
- Sam Aaron - sound/Sonic Pi
- Alan Blackwell - image/Palimpsest

Afrodita Nikolova is a PhD student in Education at the University of Cambridge, having previously studied English literature and arts creativity education. She is a prize-winning poet, interested in poetry slams, and is the first Macedonian student awarded the Gates Cambridge scholarship. Sam Aaron is a computer scientist and live coder, who performs constantly around the UK and internationally. He developed the popular Sonic Pi system, and is committed to helping young people understand the creative opportunities in computer science. Alan Blackwell is Reader in Interdisciplinary Design at the University of Cambridge Computer Laboratory. Alan and Afrodita's participation in ICLC has been generously funded by the Boeing Corporation, and Sam's research is funded by a donation from the Raspberry Pi Foundation.

6.2. Mind Your Own Business - Birmingham Laptop Ensemble

Mind Your Own Business (2013, MYOB) is a piece which entangles different elements and roles in an electronic sound performance amongst a group. Noone can play any sound without anyone else's activity. Each performer only has partial control over the resulting music: rhythm, pitch, timbre and manipulative effects are the different roles. Adding to the confusion the ensemble then rotates the designated roles throughout the duration of the piece. Everything is livecoded which poses yet another challenge in view of differing keyboard layouts. The music is improvised based on a starting set of three synthesized sounds but everything can be changed at anytime.

How can something musically interesting evolve in the given time slot of three minutes? Can one 'mind their own business' yet work towards a continuity from the predecessor's code while integrating the things happening on screen and in the space around? MYOB exposes the complexity of algorithms and networks for the participating performers in a live performance situation to create a state of disorientation. This represents in some ways the distance and alientation that the machine introduces for human music performance on stage: a strongly cognitive activity where one is never really in control.

The piece was built using the ‚Republic' quark of the SuperCollider (SC) programming language which enables networked livecoding of music.

Jonas Hummel (author) is a sound engineer, experimental musician and currently a PhD researcher, at Manchester Metropolitan University, England. His research interests include collaboration in network music situations, group improvisation with custom-built interfaces and instruments and the agency of technological objects in real-time computer music performance. Previous projects include ensembles of networked radios (Translocal Rundfunk Orchestra) and laptop computers (PowerBooks UnPlugged, Birmingham Laptop Ensemble, Republic111). He also worked in various documentary film projects for sound recording and design. www.jonashummel.de

BiLE (Birmingham Laptop Ensemble) are a collaborative group of composers and performers of instrumental and electroacoustic music. Live performance and improvisation are at the core of BiLE which has an open attitude towards experimentation with sound, allowing members of the group to bring their own tools and musical experience to the ensemble. This enables composers with varied backgrounds to contribute to every aspect of the performance. BiLE like to experiment with new ideas and technology and are continually developing and intergrating new aspects into their performances. The current core team of performers are: Charles Céleste Hutchins, Shelly Knotts, Holger Ballweg and Jonas Hummel. BiLE perform with live visuals by glitch artist Antonio Roberts. www.bilensemble.co.uk

6.3. vida breve - tristeTren

vida breve' is an audiovisual feedback dialog between the Ollinca's guitar processed by SuperCollider and drawings processed by Processing then live coded in Tidal, also Ollinca use her voice to establish an algorithmic - analog dialogue, the approach is to generate a live coded guitar feedback loop, playing and improvising analog sounds in real time to turn and cut these processes and loudly redirected by writing algorithms in real time.

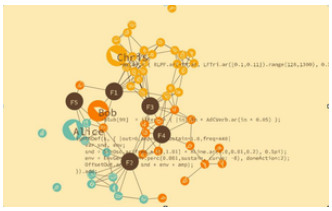
The brief history of Ollinca,

- 1988, Mexico City.
- 2008, Bachelor Degree, Visual Communication Design in the FAD of the UNAM with a major in Audiovisual and Multimedia Design.
- 2011, Diploma in Digital Video.
- 2011, Refresher course in Experimental Illustration.
- 2011, Master of Design and Visual Communication at the Posgrado de Artes y Diseño, UNAM. with the thesis project "The Illustrated Letter, a Resource for the Graphic Designer". I´m a freelance designer, and I'm permanently involved in lettering and illustration, as well in a few music projects :D

Rodrigo Velasco (1988, Mexico City)

studied Graphic Design and explored the interplay between sound and image; currently interested in writing of code-poetry, he has created yèct a space for germinated ideas; also, is involved in live coding which develops as part of his current research in the Visual Arts Master at UNAM and ICSRiM, University of Leeds.

6.4. Flock - Shelly Knotts, Holger Ballweg, Jonas Hummel



Flock (2015) for Live Coders explores flocking mechanisms in network structures as a means of managing collaboration in a live coding performance. Loosely modelling the behaviour of bureaucrats in their interactions with self-regulating political systems, the three performers engage in a live coding election battleground, hoping to win votes from an artificial population. The more votes a performer wins, the more prominent in the final mix that performer's audio will be. Performers will develop their framework code in rehearsals beforehand, allowing them to form individual musical election strategies, before making their policy proposals (in musical form) to the artificial population in performance. The voting mechanism itself is based on Rosen's work on flocking in bi-partite decentralized networks (Rosen 2010). In Flock the network is made up of 2 types of node: feature trackers (using the SCMIR library in SuperCollider); and AI agents (who have preferences and voting rights). The feature tracker nodes hold information relating to the current feature states of the input audio from the performers (1 node per feature). At regular intervals during the performance the AI population will 'vote' for the audio input which has the closest current feature set to that of their ideal profile.

Humans and agents alike become ensnared in a chaotic game of cat and mouse as the clarity of whether the human input to the system is convincing the AI society to flock to their musical proposal, or the humans are rather chasing the various preferences of the agents to win votes, becomes blurred. The humans can't predict exactly how agents will react or move within the network. In order to win votes the humans can aim for mass appeal with relatively neutral proposals or try to find a radical niche which strongly differentiates them from other performers.

Shelly is a data-musician who performs live-coded and network music internationally, collaborating with computers and other humans. She has received several commissions and is currently part of Sound and Music's 'New Voices' emerging-composer development scheme. She is currently studying for a PhD with Nick Collins and Peter Manning at Durham University. Her research interests lie in the political practices implicit in collaborative network music performance and designing systems for group improvisation that impose particular (anti)social structures. As well as performing at numerous Algoraves and Live Coding events, current collaborative projects include network laptop bands BiLE and FLO (Female Laptop Orchestra), and live coding performance [Sisesta Pealkiri] with Alo Allik. http://shellyknotts.co.uk/

Holger Ballweg is a live coder and programmer living in Newcastle upon Tyne (UK). After studying for an M.A. in Musikinformatik (Music Informatics) at Karlsruhe University of Music (Germany), he is now pursuing a PhD in sonification with Paul Vickers at Northumbria University. He is a member of the live coding band Benoît and the Mandelbrots, performing over 60 concerts in Germany and Europe. He also performed with the Birmingham Laptop Ensemble (BiLE) and with Shelly Knotts. http://uiae.de.

Jonas Hummel is a sound engineer, experimental musician and a PhD researcher, at Manchester Metropolitan University, England. His research interests include collaboration in network music situations, group improvisation with custom-built interfaces and instruments and the performativity of technological objects in real-time computer music performance. Previous projects include ensembles of networked radios (Translocal Radiophonic Orchestra) or laptop computers (PowerBooks UnPlugged, Birmingham Laptop Ensemble, Republic111). He also works in documentary film projects as sound recordist and sound editor/designer.

7. Concert F, Wednesday

7.1. Cult of graa> - Niklas Reppel

'Cult of graa>' is an umbrella term for instant composing performances, coded live and from scratch with the aid of the 'graa>' mini-language.

The 'graa>' language is a self-developed, Python-based domain specific language for the rapid creation and manipulation of directed graphs, Markov chains and other stochastic processes. Thus, it incorporates non-deterministic elements in its very foundations, and the challenge for the performer lies in juggling with a bunch of musical objects, each of which behaves unpredictable to a certain amount, without drifting into complete randomness.

While the language lends itself for a variety of musical styles, this particular performance will explore the juxtapositon of piano sounds and simple synthesized waveforms and is

inspired by the somewhat harsh piano works of Giacinto Scelsi, the player-piano pieces by Nancarrow, but also the pattern-oriented minimalism of composers like Terry Riley.

Niklas Reppel (http://www.parkellipsen.de), born on November 21, 1983 in Witten an der Ruhr, Germany, is a programmer and musician and holds a B.Sc. in computer science from TU Dortmund. He recently left the world of commerical software development to pursue a master's degree in music informatics at the IMWI at HfM Karlsruhe. Formerly rooted in the local improv- and jam-session scene as an instrumentalist, an interest for improvisative live-coding developed quite naturally. Bored by deterministic music, and inspired by the industrial/natural soundscapes of his home area, non-determinism gained increasing significance in his work.

7.2. Improvisation - very long cat (Shawn Mativetsky and David Ogborn)

very long cat are a new network music ensemble combining tabla (Shawn Mativetsky) and live coding (David Ogborn), rehearsing and performing via the Internet and employing an eclectic range of techniques and technologies. For the inaugural International Conference on Live Coding we will improvise a combination of two formats or "genres" that are complementary features of the ensemble's work: live coding signal graphs that transform the live sound of the tabla sound (live coding "live electronics"), and live coding patterns and grooves that provide a rhythmic frame for tabla improvisation.

With very long cat, we are not only bridging physical distances, but are also working to bring together two very different musical cultures, and both old and new practices of improvisation and creative music making. Shawn Mativetsky performs tabla in both traditional and new music contexts. He learned in the tradition of the Benares gharana, which is renowned for its rich tradition of solo tabla playing, drawing upon centuries old traditions of improvisation. David Ogborn's live coding practice has been strongly influenced by the live coding community around the Cybernetic Orchestra (McMaster University) and the greater Toronto and Hamilton area, and attempts to engage as directly as possible with the "theatre" of code, privileging simple and visible code that is modified quickly. For very long cat, Ogborn performs with SuperCollider, and especially with the affordances of JitLib.

Exponent of the Benares gharana, and disciple of the legendary Pandit Sharda Sahai, Shawn Mativetsky is a highly sought after tabla performer and educator. He 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 solo CD, Payton MacDonald: Works for Tabla, was released in 2007, and Cycles, his new CD of Canadian compositions for tabla, was released in the fall of 2011. http://www.shawnmativetsky.com/ David Ogborn (a.k.a. d0kt0r0) is a hacker, sound artist and improviser. At McMaster University in Hamilton, Canada he directs the live coding Cybernetic Orchestra (http://soundcloud.com/cyberneticOrchestra) , and teaches audio, code and game design in the undergraduate Multimedia and graduate New Media and Communication programs. http://www.d0kt0r0.net

8. Club B, Wednesday

8.1. Shared buffer - Alexandra Cárdenas, Ian Jarvis, Alex McLean, David Ogborn, Eldad Tsabary

Shared Buffer is a series of live coding improvisations by an ensemble of globally distributed performers (Berlin, Hamilton, Montréal, Toronto and Sheffield), all working on connected code in shared text editing interfaces. The group uses Tidal, a small live coding language that represents polyphonic sequences using terse, highly flexible and polyphonic notation, providing a range of higher order transformations.

The performers in the group are connected via the extramuros software for Internet-mediated sharing and collaboration, which was originally developed for this ongoing project. The performance is part of a series of such performances supported by the research project "Live Coding and the Challenges of Digital Society" (McMaster University Arts Research Board).

With the extramuros software, a server is run at some generally reachable location on the Internet. Performers use conventional web browsers to interact in real-time with shared text buffers provided by the server. When code evaluation is triggered in the browser window, the code in question is delivered to any number of listening clients typically at all independent locations where the performance is taking place. Independent software applications render the performance from the code at each performance site.

Alexandra Cárdenas is a Columbian composer and improviser, currently pursuing a Masters degree in Sound Studies at the University of the Arts in Berlin. Her work focuses on experimentation using live electronics, improvisation, creation of controllers and interfaces and live coding.

Ian Jarvis (a.k.a. frAncIs) is a sound artist, producer, and researcher from Toronto. His work is motivated by the implications of digital technology for creative and scholarly practices with a particular focus on live coding and the digital humanities.

Alex McLean is Research Fellow and Deputy Director of ICSRiM in the School of Music, University of Leeds, and co-founder of Algorave, TOPLAP, the AHRC Live Coding Research Network, and ChordPunch recordings.

David Ogborn (a.k.a. d0kt0r0) is a hacker, sound artist and improviser. At McMaster University in Hamilton, Canada he directs the live coding Cybernetic Orchestra (http://soundcloud.com/cyberneticOrchestra), and teaches audio, code and game design in the undergraduate Multimedia and graduate New Media and Communications programs.

Eldad Tsabary is a professor of electroacoustics at Concordia University, founder of the Concordia Laptop Orchestra (CLOrk), and current president of Canadian Electroacoustic Community—Canada's national electroacoustic organization.

8.2. Cyril vs Microscopadelica - Darren Mothersele

A participatory live experiment in combining user-generated textures with live coded 3D visuals.

The aim of this installation is to bring participatory elements into a live coding performance. The live coder co-creates the performance with the audience as they attempt to respond appropriately to the input generated by the participants.

Cyril is designed for fast prototyping of visualisations and live coding of reactive visuals. It was created (accidentally) by Darren Mothersele, a creative technologist from London, UK. http://darrenmothersele.com

8.3. Unorganised Music - Calum Gunn

Unorganised Music is a monolithic moiré pattern of rave-inflected, ever-shifting drums, stabs and sirens. A paranoid journey into the dark corners of the dancefloor.

Using a custom-built SuperCollider patch, two streams of sound are used to curate unnatural patterns on the fly, jumping between bass-heavy rhythms and hoover tunnels at will.

Calum Gunn is a Scottish programmer and musician living in London.

8.4. Tlaxcaltech - ~ON [Emilio Ocelotl + Luis Navarro]

Tlaxcaltech is an exploration of dance music and controlled improvisation through the use of SuperCollider from a mexican perspective.

~ ON

Duo formed in 2012 by Luis Navarro and Emilio Ocelotl. It focuses on audiovisual creation from programming languages and human-machine interaction, mainly using Fluxus and SuperCollider.

~ ON has performed at various events such as the International Symposium of Music and Code /*Vivo*/ (Mexico, 2012), the audiovisual concerts series Source (Mexico , 2013) , La Escucha Errante: research meetings , and electroacoustic sound creation (Bilbao, Spain 2014), Databit Festival (Arles, France, 2014) and at the Bern (Bern, Switzerland, 2014). In 2015 ~ ON organized the Permanent Seminar of the Algorithmic Party "Reggaetron".

Emilio Ocelotl: Sociologist and violist. He has taken courses in the area of computer music and composition at the Multimedia Center of the National Arts Center and the Mexican Center for Music and Sonic Arts. In 2013 he awarded a scholarship to support Young Composers in the Area of the Mexican Center for Electroacoustic Music and Sonic

Arts. His work has been presented at the Museum of Contemporary Art UNAM under the International Symposium on Music and Code / * vivo* / in the Multimedia Center of the National Arts Centre; his pieces has been presented at the International Cervantes Festival (2013) in the framework of the International Forum of New Music "Manuel Enriquez" (2014). https://soundcloud.com/emilio-oc

Luis Navarro: he studied music composition and Performance in contemporary popular music at the Academy of Music Fermatta in Mexico City (2005-2011). In 2012 he collaborated in the creation of the International Symposium of Music and Code /*VIVO*/ and lectured on the subject of "live programming" in places like the Campus Party Mexico (2011). His work, that includes image and sound, has been presented at the Multimedia Media Centre of the National Centre of Arts and at the universities UAM (2013), UNAM (2011) and University of Chapingo (2010). https://sites.google.com/site/luisnavarrodelangel

8.5. Linalab - Linalab

live coding – live patching

The laptop won't produce sound by itself, it will be sending control signals to analogic modular synthesizer

Lina Bautista, musician and sound artist, she was born in Bogotá, Colombia. She is currently living and working in Barcelona, Spain. Her work is divided in two different ways: music and live performances and soundscapes, exhibitions and other sound projects. As a musician she has participated in: Sirga Festival, Cau d'Orella, TEI international congress, Live.Coding.Festival, LEM, Eufònic and Sonar. She currently works with The Orquesta del caos, research project about sound art and experimental music, and host of the Zeppelin Festival. She is member of Sons de Barcelona, a collaborative group that work in fostering the interest in sound and technologies, and she also makes DIY workshops with Familiar.

8.6. LifeCoding - Andrew Sorensen

With over ten years of live-coding practice, Andrew is an old-hand at the live-coding game. As an institutionally trained musician, Andrew's performances place a strong emphasis on what he loosely describes as "musicianship". As part of his exploration into "musicianship" in live-coding Andrew has explored a diversity of musical styles. This diversity of styles is reflected in his performances, which often exhibit a bias towards western music theory, but are less clearly delineated by any particular genre. Andrew attempts to make his performances "understandable", even to those unfamiliar with his environment, by focusing on simple imperative algorithms built from scratch during the performance. Andrew performs using his own Extempore programming language - a general purpose language with a focus on high-performance computing and real-time systems. Extempore is a research programming language designed to explore notions of physicality and liveness in a diversity of cyber-physical domains, from sound and music through to plasma-physics and astronomy.

Andrew Sorensen is an artist-programmer whose interests lie at the intersection of computer science and creative practice. Andrew is well known for creating the programming languages that he uses in live performance to generate improvised audiovisual theatre. He has been invited to perform these contemporary audiovisual improvisations all around the world. Andrew is the author of the Impromptu and Extempore programming language environments

8.7. Gibberings and Mutterings - Charlie Roberts

Gibberings and Mutterings will be performed using Gibber, a live-coding environment for the browser created by the performer. One of the goals of the performance is to expose aspects of state and algorithmic processes to the audience; to support this we have added a number of novel visual annotations to Gibber.

These annotations primarily focus on revealing both the timing and output of sequencers controlling audiovisual properties. We additionally show algorithmic manipulations of

musical patterns over time and enable font characteristics to display continuous audio properties.

The piece is composed at the macro-level, but provides for a variety of micro-level variations and improvisations.

Charlie Roberts explores human-centered computing in the context of creative coding. He is the author of Gibber, a browser-based live-coding environment, and a variety of open-source software packages augmenting end-user interaction. He is currently a Robert W. Deutsch Postdoctoral fellow with the AlloSphere Research Group at the University of California, Santa Barbara; in the fall he will join the Rochester Institute of Technology as an Assistant Professor of New Media.

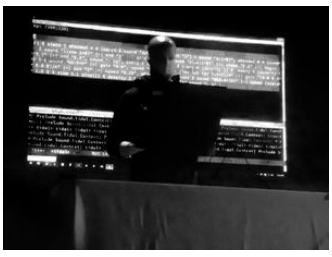
8.8. Heuristic ALgorithmic Interactive Controllers - H.AL.I.C

H.Al.I.C. is a live-coding duo searching for an interactive combination of audio and image generation to create music and visuals in real-time, originating from the same data. H.Al.I.C. is using Clojure, a dynamic programming language that targets the Java Virtual Machine. Both audio and visuals are using open source environments : Overtone and Quil. All sound of the music , generated by a laptop in this performance, is synthesized in Supercollider using Overtone, without the use of any samples, only synthesizers are involved. The visuals are generated in Quil, combining Clojure with Processing. The data used to generate graphics will be send to the synthesizers through a bi-directional communication pipeline to manipulate the sound, and at the same time, modulation data of the synthesizers can be send to provide parameters to generate images. In addition , Sondervan's DIY electronic drum-kit will be used to interact with the coded autonomic composing and improvisation tool, making use of Axoloti, an open source, digital audio platform for makers.

Dagobert Sondervan is a schooled jazz-drummer who started to work as a professional musician in 1987, still in his teens, and used computers to generate music sinds 1990. He started programming while writing algorithms in Symbolic Composer on the Atari platform. Also he has produced records with the results using the alter-egos Anton Price and Bohrbug. As Dj Dago he introduced Jungle and Drum & Bass in the Belgian club scene as resident DJ of the famous Belgian club FUSE. He has his own jazz and electronic bands as a leader, and works as a session musician for many bands and projects. Dago is co-organiser of λ-sonic, Belgium's first algorave festival.

Kasper Jordaens is schooled as an engineer-architect and has been approaching problems using technology and aesthetics. For a while he's been doing visuals for DJs while also exploring data visualisation. After building custom controllers (both hardware and software) to act closer to the music the next step was livecoding, to try and visualise music on the fly. Kasper is co-organiser of λ-sonic, Belgium's first algorave festival.

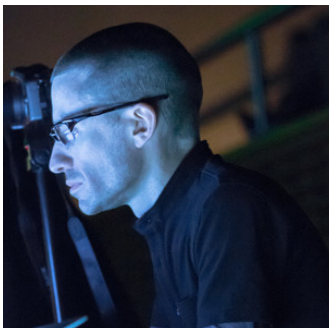
8.9. humanly-organized sound - Kindohm



In this performance I will use Tidal to create algorithmic and broken electronic music, influenced by and fusing classic styles such as Drum & Bass, breakcore, glitch, dub, and minimal industrial. I will use Tidal to control MIDI synthesizers as well as to create patterns from original samples. The performance will demonstrate a balance between pure improvisation and composed, recognizable, repeatable motifs.

Mike Hodnick (a.k.a. Kindohm) is an independent programmer and sound artist from Minneapolis, USA. In 2014 he produced the 365TidalPatterns project, regularly performed live-coded music sets, released "I Am a Computer" on Xylem records - an album produced exclusively with Tidal, and received the Minnesota Emerging Composer Award from the American Composer's Forum. Mike continues to perform regularly and produce music with code.

8.10. Improvisation - Renick Bell



This performance differs from my performances prior to 2015 in that additional agent processes which change system parameters (conductor agents) run alongside sample-playing agent processes (instrumentalist agents). These conductor agents are the result of my recent research into how autonomous processes can complement live coding activity. These conductors stop and start instrumentalists, as well as change the other

parameters used by the instrumentalists for sample-triggering, such as which sample to play and which rhythm pattern to follow. The live coding involves not only the patterns for rhythms and samples but also the algorithms which the conductors use during the performance.

By manipulating both instrumentalist and conductor agents, a rapidly changing stream of algorithmic bass, percussion, noise, and tones is improvised according to a rough sketch of the overall performance structure. This improvisation crosses the boundaries of bass music, noise, and free improvisation.

I use my own Haskell library for live coding, Conductive. The sample player was built with hsc3, a Haskell client for SuperCollider by Rohan Drape. Interaction with the system takes place in a combination of the Glasgow Haskell Compiler Interpreter (ghci), the vim text editor, the xmonad window manager, and the tmux terminal multiplexer.

Renick Bell is a musician, programmer, and teacher based in Tokyo, Japan. He is researching live coding, improvisation, and algorithmic composition. He is the author of Conductive, a library for live coding in the Haskell programming language. He has a PhD in art from Tama Art University in Tokyo, Japan, a masters in music technology from Indiana University, and a bachelors in electronic music, art, and philosophy from Texas Tech University. He has performed across Asia and Europe and in Australia and the United States. Though from West Texas, he has lived in Tokyo since 2006 and Asia since 2001.

Workshops

Workshop abstracts

1. Extramuros: collaborative live coding and distributed JavaScript visualizations

- David Ogborn, McMaster University
- Eldad Tsabary, Concordia University
- Ian Jarvis, McMaster University
- Alexandra Cárdenas, University of the Arts in Berlin

The extramuros software was developed to explore live coding and network music, bringing live coding musicians together around shared, text buffers. A half-day workshop at ICLC 2015 around extramuros will be divided into two sections. The first half of the workshop introduces the software and leads to a collective jam session in the Tidal language. The second half of the workshop is a hack-in around new possibilities for distributed, live-coded JavaScript visualizations of collective live coding activity.

1.1. Details

extramuros was developed to explore live coding and network music, bringing live coding musicians together around shared text buffers. In it's basic operation, a server is run on one computer. Some number of performers use a web browser to connect to the server and edit shared buffers of text code, with each person's editing visible to the others. At any computer where the sounding result of the code is desired (for example, the individual machines in a globally distributed ensemble), a client application is run that receives evaluated text from the server and pipes it to the audio programming language of choice. The software is released on an ongoing basis through github.com with a GPL version 3 license.

At ICLC 2015, we will conduct a half-day workshop around the extramuros software. In the first part, we will introduce the software, assist with troubleshooting, and perform a networked jam session in the Tidal language. In the second part, we will have a "hack-in" on new distributed visualization facilities. These will allow Open Sound Control messages to the server to arrive back in the connected web browsers as calls to a JavaScript stub function, which could be used to give feedback about syntax errors, other aspects of system state (such as audio levels), or more sophisticated forms of visualization of live coding activity. The intent for the hack-in is to collectively produce some strong examples of JavaScript visualization of live coding activity, which could potentially be incorporated into the extramuros codebase as examples for further development.

Participants are invited to bring their own laptops with WiFi. No software installation is required as everything to be done in the workshop can be done through a web browser. Previous exposure to Tidal and/or JavaScript, while beneficial, is not required and we will demonstrate basic strategies applicable to both halves of the workshop.

Software: [https://github.com/d0kt0r0/extramuros]
Example of extramuros in action: [https://www.youtube.com/watch?v=zLR02FQDqOM]

2. Live Coding the OpenGL Fragment Shader with Audio Responsiveness

- Shawn Lawson, Rensselaer Polytechnic Institute

2.1. Purpose

The intent of this workshop is to create a larger community of graphics-based live coders and introduce a performance IDE, *The Force*. *The Force* is open source, https://github.com/shawnlawson/The_Force ; run it here, http://shawnlawson.github.io/The_Force/.

2.2. Curriculum

All portions are hands-on, total length is approximately 3 hours. Additional detailed material can easily be included should 4 hours be expected. For self-starters, the IDE has Open Sound Control communication, should anyone which to connect to external applications.

- 15 min - Introductions and getting everyone up and running
- 10 min - Explanation of the tool, how it's constructed, and how it works: lecture/demonstration
- 25 min - Some basic functionalities of the OpenGL Fragment Shader: color, texture, uniforms, coordinate spaces
- 10 min - Break/breather
- 50 min - Some simple coding examples: basic math, shapes, lines, patterns, animation
- 10 min - Break/breather
- 60 min - Integrate audio source with some examples, go live coding crazy: audio input, OSC for the adventurous

3. Opusmodus: Live Coding and Music Composition Environment Workshop

- Stephane Boussuge, Composer, Sound-designer, Associate Composer at Opusmodus Ltd.

"Composing music today is a fascinating business. As the technology gets more complex, I feel more like a computer programmer than a musician. But I take the sounds in my head and in the world around me and bend the technology into the world I want to create." - Robert Rich

Algorithmic software is revolutionizing creative disciplines. 3D design, film animation and CGI effects, textiles, ceramics, fine art practice… We are now also seeing extraordinary inventions in signature buildings that have been designed by applying parametric software (Rhino and Grasshopper).

3.1. Why Opusmodus?

Opusmodus software operates by the same principles. Opusmodus is a new Live Coding and Music Composition environment with hundreds of functions generating and processing musical events combined with features enabling also an output in the form of a score notation.

With design factor points to realize and render thoughts and ideas in standard staff notation, Opusmodus is particularly appropriate for creating music for human performance combining instruments and voices. Let's not assume that it cannot be useful in other areas of music, but we have realized that there is a gap in the production of concert and media music (including games) that Opusmodus can fill effectively.

At the heart of Opusmodus is OMN (Opus Modus Notation), a special notation describing every possible score notation output as a script. The multiple functions in Opusmodus can generate and manipulate OMN notation, for Live Coding or Music Score composition, giving a very elegant, powerful and innovative way to compose music.

This workshop will introduce you to the Opusmodus Environment and teach the basics of this powerful music creation software.

3.2. What are you going to get from this workshop?

1. New possibility for music composition exploration and experimentation
2. New toolbox understanding with hundreds of tools
3. Access to the Opusmodus user's community
4. Good knowledge of score generation and creation with Opusmodus tools
5. Global view of all the software features: 5.1 Fast score generation 5.2 Export to MusicXml 5.3 Midi import capabilities 5.4 Powerful music description language

3.3. What are the requirements?

A computer, a desire to learn, a willingness to share your passion for music composition with your peers around a truly innovative composition tool designed to leverage your creativity!

3.4. Opusmodus Workshop or how you can embrace your creativity with a powerful user centric tool designed by composers for composers!

The workshop will start with the presentation of the Opusmodus environment…

How the composer will use the Opusmodus interface with its many features will always be related to her/his experience and personal choice. But one of the objectives around the design of Opusmodus is to respond to the various approaches composers have to make in particular projects under certain circumstances.

No creative task is ever quite the same, so it would be foolish to expect one workspace design to 'catch all' desires. Nevertheless, the Opusmodus interface is multi-faceted and has already proved it can deal with very exacting musical situations from the highly conceptual and experimental to the pragmatic needs of a commissioned work.

… **followed by a presentation of Opus Modus Notation key features,**

As music notation moves inextricably from printed pages to backlit digital displays, OMN language fully responds to the future of music presentation. With computers replacing pianos as a composer's helpmate, so have the conditions changed surrounding music creation.

New music technology has focused largely on production and presentation, whereas the conceptualisation and origination of new music requires a very different paradigm. Opusmodus takes a step towards that paradigm, being a 3rd way forward and driven by its own notation script OMN.

Working directly in OMN is perfect for those 'on the fly' experiments (test and learn!) that all composers make when they are starting out on a project.

It is almost like having a piano close by to lay down new creative thoughts, but with one that always plays what's written quite flawlessly and captures your creativity at a particular time, anywhere, everywhere.

… **with a review of some typical approach to music composition using Opusmodus toolbox key features** (System Functions)

With such a powerful application, there are just so many things that are not just useful but necessary. You need to dive into it. But the advantage of a digital workspace for a composer is that it can bring together in a single location many, different, and essential things we need to make ever lasting music creation. We will this opportunity to study some of the System Functions that form the vocabulary of the scripting language of Opusmodus… in just a few hours!

… **and finally improvise together with Opusmodus showing the features of the "Live Coding Instrument" (LCI)**:

The LCI gives the composer an intuitive control panel and the possibility of working in true live coding style directly with the script. In practice composers who use the Live Coding Instrument often begin with a script, making a change, then 'playing' that change from the buttons of the LCI control panel. Where it becomes a true composer's instrument is in its "Save" button. Yes the "save button". This function captures to midi-file every nuance of a 'live' session . . . and from midi-file, via the Export menu, to OMN script and from there, well let's say there is no frontier between technology and creativity.

4. Live Coding with Utopia

- Birmingham Ensemble for Electroacoustic Research

This workshop will introduce participants to networked live coding in SuperCollider using the Utopia library ([https://github.com/muellmusik/Utopia]). Utopia is a modular library for the creation of networked music applications, and builds upon the work of the Republic Quark and other older network systems in SuperCollider. It aims to be modular (features available largely 'à la carte'), secure (provides methods for authentication and encryption), and flexible (to the extent possible, it tries not to impose a particular design or architecture). It provides functionality for synchronisation, communication, code sharing, and data sharing.

For the last two years, the Birmingham Ensemble for Electroacoustic Research has used Utopia as the primary platform for its activities, and actively developed it in workshops, rehearsals, and concerts. In these sessions, members of BEER will introduce participants to the library and its possibilities for live coding, with a particular emphasis on creating networked structures for live coded improvisation.

5. Livecoding relations between body movement and sound

- Marije Baalman, nescivi

Workshop on livecoding relationships between body movement and sound. Using wireless sensors that capture movement of the body and a data sharing network, the participants will engage in collaboratively create mappings of sensor data to sonic or other media output.

5.1. Description

As (wireless) sensor interfaces become more and more available and accessible to artists, the creation of the the relationships between the data the sensors produce and the output in media has become more and more a subject of research. Rather than fixed relationships or mappings between sensor data and output media, livecoding can create a dialog between all the performers in an interactive performance, between the movement and the code, between the movers and the coders. Even in preparation of interactive performances, livecoding as a skill is very valuable in order to be able to quickly prototype different approaches of mapping data, try them out, and evaluate their artistic quality. The adaptations can range from changing of parameter ranges, to rewriting the algorithms that establish rules for mapping. As a mode of performance, this approach can enable a new form of improvisation, creating a dialog between dancers and livecoders, as the dancers adapt their movements based on the mappings to output media that are created by the livecoders, and the livecoders who adapt their code based on the movement of the dancers.

During the workshop we will collaboratively explore the livecoding of such mappings, using a wireless sensor system (the Sense/Stage MiniBee ; [https://docs.sensestage.eu]), equipped with accelerometers and a selected range of other body-based sensors, and a data sharing network ([https://github.com/sensestage/xosc]).

The workshop will start with an exploration of the framework within which we will work, before going on to exploring body movements, the sensor data this produces and strategies for mapping this data to output media - mainly sound. While examples of algorithms will be given in SuperCollider, for the experienced livecoder they should be easy to translate into his or her preferred programming language, as long as a quick access to incoming OSC data is available. The workshop should end with a collaborative livecoding and movement session of all participants.

5.2. Participant requirements

- A laptop with the livecoding programming language that you are familiar with.
- The livecoding programming language should be capable of receiving (and sending) custom OSC messages.
- The laptop should have WiFi capabilities (or you should bring an ethernet cable).
- Participants with a background in movement (dance, theatre, etc) are also welcome - even if they may not be livecoders themselves, but are interested in the collaboration with livecoders.

6. Slow coding, slow doing, slow listening, slow performance

- Hester Reeve, Sheffield Hallam University
- Tom Hall, Anglia Ruskin University

6.1. Overview

Explorations of slowness over the last generation have ranged from the ‘slow food’ movement, ‘slow scholarship’ and ‘slow technology’ in design practice (Hallnäs and Redström 2001), to recent academic work on aesthetic slowness (Koepnick 2014) and in popular culture, the emergence of ‘slow television’ (BBC Four Goes Slow, [http://www.bbc.co.uk/programmes/p02q34z8]). Whilst speed has been an ubiquitous trope surrounding cultures of the technological, in this workshop we focus on the implications of different notions and approaches to slowness in relation to the body (hearing, seeing, performing), sound and composition (composition of code, music, writing or a mixture of all three).

This workshop is open to coders/performers and writers/theorists alike. Our objective is to allow participants to experience and understand the aesthetic value of altering the timescale of live coding and live performance, and to consider the

wider social and theoretical meanings of doing so. We consider this workshop to be an experimental laboratory rather than a masterclass.

What to expect: firstly, through simple workshop situations, we will facilitate participants to experience the 'texture of the present' via a series of slow activities aimed at the senses. Secondly, through participants' own work, we will provide a framework for participants to consider and experiment with how insights gained from the above might open up potentials for their own practice of composition (and therefore for the experience of their audience or readership). There will be an opportunity for an open group discussion of issues and discoveries that have arisen at the end of the session.

Some of the questions that have informed the content of our workshops are:

- How does timescale adjust the meaning and experience of a work? Can we meaningfully distinguish between aesthetic and purely temporal slowness?
- How can slowness be used as a non-oppositional force of the contemporary?
- 'To slow code, is to know code'? (Hall 2007)
- What might a mindful experience of the electronic feel like?
- To what extent does subjectivity–authorship control code and under what circumstances does it collaborate with code to let in creative 'unknowns'?

We ask participants to bring a current piece of work in process with them to the workshop (and a laptop if a coder, an instrument if a musician and writing means if a writer).

6.2. Workshop leaders

Hester Reeve has been working with time based live art performance for over 20 years. Many of her solo works are over 4 hours long, sometimes spread over a period of days. In 2012 Reeve devised the AHRC-funded "Live Notation – Extending Matters of Performance" with live coder Alex McLean ([http://blog.livenotation.org]). [http://www.shu.ac.uk/research/c3ri/people/hester-reeve]

Composer, performer and musicologist Tom Hall coined the music-related meaning of the term 'slow code' in a 2007 manifesto (Hall, 2007) and has a decade's worth of experience teaching and performing using SuperCollider. Recent compositional work uses traditional instruments with electronics whilst embracing notions of aesthetic slowness. [http://www.ludions.com]

6.3. References

- Hall, Tom. 2007. *Towards a Slow Code Manifesto.* Available online, http://ludions.com/texts/2007a/
- Hallnäs, Lars and Redström, Johan, 2001. *Slow Technology: Designing for Reflection.* Personal and Ubiquitous Computing, 5 (3). pp. 201-212.
- Koepnick, Lutz. 2014. *On Slowness.* New York: Columbia University Press.

ISBN 978 0 85316 340 4