CHAPTER 16

TENSIONS AND TECHNIQUES IN LIVE CODING PERFORMANCE

CHARLES ROBERTS AND GRAHAM WAKEFIELD

16.1 Introduction

Various definitions of the term 'live coding' have been suggested; these variations and their subtleties depict the tensions that riddle the field.[1] Let us begin with a working definition: *in live coding performance, performers create time-based works by programming them while these same works are being executed*. Although many would likely argue for greater specificity, we believe the above definition encompasses most live coding performance practices, and will use the remainder of this chapter to explore the tensions encountered and techniques utilized in live coding that make a one-size-fits-all definition so difficult to achieve. Although the chapter is primarily concerned with musical performance, we would be remiss if we did not mention that live coding is active in visual media as described by Griffiths (2014), Della Casa and John (2014), and Wakefield, Smith, and Roberts (2010), and other domains of performance such as dance (Sicchio 2014).

Modern, canonical live coding performances began in the early 2000s, with performances given using custom live coding systems such as feedback.pl as well as musical programming languages such as SuperCollider (Ward et al. 2004), though the origins of live coding can be traced to earlier dates (Collins 2014). One characteristic of these performances is the writing of algorithms and their subsequent manipulation, and for some this is integral to its attraction: 'the more profound the live coding, the more a performer must confront the running algorithm, and the more significant the intervention in the works, the deeper the coding act' (Collins 2011). While the primacy of the algorithm in live coding performance has been identified by other authors (Magnusson 2011a; Ward et al. 2004), it is not universal, and there is a veritable spectrum of use and modification of algorithmic processes in live coding practice. In the live coding environment LOLC, by Jason Freeman and his research group at Georgia Tech, variation and development of musical pattern emerge from collaborative ensemble practices instead of algorithms. Performers submit musical patterns to a queue that is accessible to all ensemble members, so that each can easily download, modify, and execute musical patterns as they see fit (Freeman and Van Troyer 2011). Although algorithms are not strictly necessary for her work, live coder and choreographer Kate Sicchio employs them both for their creative potential and as aesthetic commentary: 'Some of my work uses generative elements but this is less about typing or transmission and more about the computer providing new possibilities for scores. My work does not need algorithms to be achieved but by using them I am commenting on both coding and choreographing and their similarities in being organisational practices' (Sicchio, personal communication 2015).

To further whittle away at our initial definition, such performances often included (and continue commonly, but not always, to include) projection of the live-edited source code documents. Privileging either the composition or the source code as the primary output of a performance may elide significant complexities; for example, as reported by Collins and McLean: 'Dave Griffiths, of the live coding band slub, considers the music he makes to be a side product, rather than an end-product of his live coding languages, where the visual aesthetics of his interfaces are more important' (2014). Although projecting source code can both reveal activity and potentially intent, there is debate inside the live coding community (and inside individual members of it) about the merit of such projections; it has been suggested that the source code can distract from the artistic focus of a performance (Bruun 2013).

There are dozens of live coding systems available, using a wide spectrum of general-purpose to domain-specific programming interfaces and languages. Where some are widely used in communities far beyond live coding performance, others are so idiosyncratic that they become statements of a personal performance oeuvre. Live coders have been active across many academic disciplines, including education, the psychology of programming, and aesthetics; performances are regularly given around the world in both concert halls and nightclubs.

We have already touched upon some tensions at play in live coding; in the rest of the chapter we use a lens of five recurrent tensions to examine the range of practices found in this young and vibrant community.

(1) How do performers consider risk when performing? An errant algorithm can send an audience running for the doors covering their ears, while a memorized performance risks boredom and inflexibility.
(2) What is the responsibility of the performer to convey algorithmic processes to the audience, and how does this responsibility hinder the immediacy of their interface?
(3) What abstractions are necessary and appropriate in live coding performance? What models, metaphors, and levels of abstraction are performers working with? Does complexity limit or enable freedom?

(4) How do live coders maintain pace and flow in performance? What techniques and notations make this easier—and how do live coders divide their attention over the temporal limits of real-time performance?
(5) What happens at the meeting of the deterministic machine with the indeterminacy of improvisation? And how do live coders deal with the much larger durations of time that code puts within their reach?

16.2 Security, Stability, Risk, and Improvisation

Live electroacoustic performance clearly contains more risk than the unmodified playback of a pre-rendered audio file. While Clowney and Rawlins (2014) argue that such risk is an essential part of all musical performance, the consequences are intensified in live coding performance, where typographical errors can lead to system crashes and performance nightmares. Despite these risks, the live coding community generally seems attracted to uncertainty. Shelly Knotts states: 'I like to perform in circumstances that are risky or have a high cognitive load. . . . I'm also interested in how unintentional outputs of algorithmic processes become part of musical structure, so I don't do too much to reduce risk of error in live coding performance' (Knotts, personal communication 2015). In a similar vein, live coder Tanya Goncalves notes that although beginning programmers 'may fear that they will introduce a mistake into their code, and that the audience will recognize it . . . this tension is something I personally wait for during a performance and look forward to' (Goncalves, personal communication 2015).

In addition to the attraction that uncertainty and risk hold for performers, Alperson (1984) notes that audiences are more forgiving of performances if they know that significant musical risks are being taken. He also notes the attunement that audiences can feel with an improviser: 'as if the improviser's audience gains privileged access to the composers mind at the moment of musical creation'. This tradeoff is especially relevant to live coding practice, where errors can cause system failure, whereas success can provide unique insight into the improvisational process. Accepting that risk is an integral element of musical performance, the following section examines strategies for managing and encouraging it in live coding performance.

16.2.1 The Allure of Failure

Over the course of many performances, it is almost inevitable that performers will experience crashes. In a notable prototypical live coding performance that took place in

1985, Ron Kuivila accidentally ended a performance with a system failure, described in Curtis Roads's review as follows:

> Ron Kuivila programmed an Apple II computer onstage to create dense, whirling, metric sounds that layered in and folded over each other. Considering the equipment used, the sounds were often surprisingly gigantic in scale. Kuivila had trouble controlling the piece due to system problems. … The reality is that personalized home-grown hardware and software, for all the freedom of expression they can afford, are especially subject to flakiness. (Roads 1986).

David Ogborn notes that despite the danger that errors in performance pose, risk yields potential rewards for audience members: 'I think too much risk and you could alienate, or even hurt, your audience, and that is something which is hard to come back from. At the same time, my experience as an audience member is that one of the exciting things about live coding are those moments where you can see people really wrestling with decisions—where you see "authentic" moments of decision being taken in front of you' (personal communication 2015). After giving a performance in which one crash occurred and another was narrowly averted, the first author of this chapter noted: 'multiple audience members commented on how the errors lent the performance a sense of danger, and emphasized the improvisatory nature of the performance. Some went as far as to say the errors were their favorite part; we remain unclear what this says about the quality of the error-free segments of the performance' (Roberts, Wright, Kuchera-Morin, and Höllerer 2014).

16.2.2 Practice Makes Perfect

As with traditional instruments, practising is one way of mitigating risk in performances. The amount of preparation accompanying a performance is a tension that each performer must navigate. Developing the fluency of gestures required to improvise in live coding is difficult, and must be potentially balanced with the desire to not 'pre-compose' public performances.

In one experiment, Nick Collins and Fredrik Olofsson committed to an hour of live coding practice each day for a month. Both felt their skills as live coding performers improved 'by introducing various shortcuts, by having certain synthesis and algorithmic composition tricks in the fingers ready for episodes, and just by sheer repetition on a daily basis' (Nilson 2007). Aaron followed this with a more nuanced description of what he believes live coding practice should ideally consist of: 'Preparation for performance should involve activities that are neither original engineering, nor simple repetition' (Aaron, Blackwell, Hoadley, and Regan 2011, 385) Aaron contends that the goal of practice is not to extend the system being used, nor is it to build up to a performance consisting of pre-composed, memorized code. Instead, the goal is to develop 'a fluent repertoire of low-level coding activities [that] will allow the performer to approach performance at a higher level of structural abstraction—based on questions such as where am I going in this performance and what alternative ways are there for getting there' (385–386). Such practice can both reduce the risk of technical errors during a performance and increase the aesthetic freedom of the performer.

16.2.3 Collaborative Performance

One of the simplest methods for managing risk in live coding performance, at least conceptually, is collaboration. If one person is forced to reboot or cease musical output for other reasons, other performers are there to fill in until that person can get up and running again; there is safety in numbers. Although the technical challenges of developing systems for ensemble live coding performances can be daunting, many environments (such as the previously mentioned LOLC) provide affordances for group performances out of the box, including the ability to share musical timing and source code. Jason Freeman, who led the research group creating LOLC, shared the following: 'With LOLC, I think scale is a big mitigating factor to risk. I've done performances with LOLC with 30+ laptops on stage, and so if I as an individual performer crash and burn, or don't generate any sound for a minute or two or three, it's no big deal. In fact, the biggest risk with LOLC tends to be that musicians feel they need to be making sound all the time and never step back to listen and choose wisely when to contribute to the texture' (Freeman, personal communication 2015).

Ensemble live coding performances are not limited to sharing code and timing; users of the Republic extension for SuperCollider can even share laptop speakers, as the extension enables users to execute code remotely on the computers of any ensemble member. The group PowerBooks UnPlugged uses this extension and sits among the audience, in effect creating a spatially distributed instrument that is controlled by all ensemble members simultaneously (Rohrhuber et al. 2007). In addition to executing code on any ensemble member's computer, the live coding environment Gibber also lets performers see and edit everyone's source code and provides a shared clock for rhythmic synchronization (Roberts, Yerkes, et al. 2015). While Republic and Gibber users are limited to specific live coding systems, the extramuros system enables distributed live coding performances using heterogeneous environments (Ogborn et al. 2015). In extramuros, performers program in a central, browser-based editing interface; each can see what their fellow ensemble members are doing. The commands from this interface are then piped to audio synthesis environments, such as Tidal or SuperCollider, pointing towards a promising future where ensembles can easily perform together regardless of the software that individual members use.

We return to another 'safety in numbers' technique to reduce risk in the discussions of scheduled parallelism later in the chapter. The topic of ensemble performance is also discussed in chapter 20, 'Network Music and the Algorithmic Ensemble', by David Ogborn.

16.2.4 Previewing Algorithms

Many of the risks in live coding stem from the relative brittleness of code. Code is rarely fault-tolerant and has the potential for unpredictable and far-reaching consequences. Renick Bell notes that his Haskell-based live coding system Conductive both adds and removes risk from live coding performances. His choice of Haskell, a strongly typed language, removes the risk of type-based errors breaking a performance. At the same time he states: 'The algorithms I use also provide risk, since they use random or stochastic processes to varying degrees and do not give me an audio preview of the output.' (Bell, personal communication 2015)

DJs mitigate risk by *previewing*: experimenting with and evaluating tracks or loops before adding them to a mix, via headphone cue mixes (and in software, via graphical representations of the audio). Yet few live coding environments emphasize previewing material before it is presented to audiences. Perhaps this is because the temporal pressures of coding in a performance, arguably a more intensive task than DJing, preclude patient preview and search? Live coding performances often suffer from a lack of variation, and having a performer previewing generative content instead of immediately presenting it could exacerbate this problem. Moreover, even deterministic algorithms can result in unexpected sounds (Rohrhuber, Campo, and Wieser 2005). Bell (personal communication 2015) notes that he *can* preview the output of algorithms by looking at their numeric output, but 'at this point that output is not as useful as I would like' and 'I just listen to their output in the performance and switch quickly if I do not like the results.'

Previewing can be a useful technique for ensemble performances, where multiple performers help ensure greater variation and provide temporal cover for other members. Some live coding ensembles have given performances where a central computer is projecting output to the audience while each performer submits code to it for execution (Ogborn et al. 2015; Roberts et al. 2014; Wang, Misra, Davidson, and Cook 2005). In such performances it is easy for performers to preview the results of executing code using the audio output of their laptops before sending it to the central computer (although this only works if the code does not heavily rely on the state of the central machine). We will return to these themes in the discussion of 'extended presents' at the end of the chapter (section 16.6.2).

16.3 Legibility and Immediacy for Audience and Performer

Canonical live coding practice encourages performers to project their screens to audience members in order to provide transparency about methods of production. Although, as mentioned in the introduction, there is debate in the live coding community about the necessity and desirability of this, it remains entrenched in live coding culture. Blackwell and Collins (2005) suggest that the audience members are end-users or consumers of the code being produced; but that without knowledge of the language, all code effectively becomes *secondary notation*. Blackwell (2015) later suggests that audience members who concentrate on following the projected code may have a greater cognitive load than performers.

Given this, what is the code of a performance able to convey to an audience? Is it merely reinforcement that content is being created in real time? Or is there the potential to convey the development of algorithmic processes to the audience? Other possibilities also exist, from political (Bruun 2013) to educational. Shelly Knotts notes: 'One of the more important aspects to me in projecting my screen is revealing mistakes and errors by the performer which I see as having a humanizing effect on electronic music performance' (personal communication 2015). However, she also feels that projecting source code does not typically reveal underlying processes, even when the language is familiar. Regardless of motivation, it is incumbent on performers who project code to consider if and how they make it comprehensible to audiences, and to balance this with the cognitive and temporal costs improving comprehension potentially incurs.

16.3.1 Annotations and Live Visualizations

A first step to improving legibility of source code is the addition of comments and the thoughtful naming of variables. As Brown and Sorensen note: 'we make an effort to use function and variable names that people will recognize and that may assist in their interpretation of the code. Symbol names such as 'outrageous-kick' and 'grunge-it-up' never fail to communicate our intent!' (2009). In the above example there is a tradeoff in verbosity (which increases typing and potentially decreases immediacy) and legibility.

As with many other techniques for increasing the legibility of source code, this requires deliberate action on the part of the performer that incurs both a temporal and cognitive cost. However, the authors clearly feel the extra effort is worth it. In the case of descriptive variable names, there is a mutual benefit to the performer, who does not have to spend time later trying to remember what was stored in a variable named 'x2'. Another technique that improves legibility for both performance and audience is the clear indication of the current cursor location. In other cases, however, increasing legibility for the audience can potentially decrease it for the performer. For example, increasing the font size of a source code document to improve its legibility to the audience means that less code is simultaneously visible to the performer.

A number of environments provide affordances for visualizing the state and timing of algorithms without requiring extra effort by performers. Alex McLean's feedback.pl was the first system for live coding performance to explore this; it used code comments to both present the current state of data structures and also afford manipulation of their contents (Ward et al. 2004). The ability to display the current state of data is also included in Thor Magnusson's ixi lang live coding environment, where algorithmic manipulations of musical patterns automatically update the source code document with their results (Magnusson 2011b). Dave Griffith's Scheme Bricks live coding system flashes visual programming elements to indicate control flow (McLean, Griffiths, Collins, and Wiggins 2010). Gibber inherits elements of all three of these systems, while also revealing the output of functions triggered over time and displaying the state of running programs by dynamically changing font characteristics of source code (Roberts, Wright, and Kuchera-Morin 2015).

While these systems all use the source code itself to indicate state and timing, other designers of live coding environments have experimented with incorporating additional graphical widgets. Impromptu and Extempore both use graphical overlays to reveal information such as timing of musical sequences and progress in audio file playback (Swift, Sorensen, Gardner, and Hosking 2013). These widgets appear primarily intended to inform performers about system state as opposed to the audience (their impact on audiences is not mentioned in papers describing the research), but it is interesting to consider how blurring the line between source code text and user interface elements could improve audience understanding, as it could open up opportunities for comprehension to people who might be intimidated or disinclined to attempt parsing source code text. Lee and Essl (2014) also provide techniques for visualizing state in the urMus system for live coding musical instruments; similar to the work of Swift and Sorensen, audience comprehension is not considered but could be an important side effect.

As mentioned in the introduction, it has been argued that viewing and attempting to comprehend code is at odds with appreciating the musical output of live coding (Bruun 2013). However, we believe visual annotations of source code can improve appreciation of music cross-modally. In the same way that watching the pendulum of a metronome can improve perception of pulse, annotations such as those found in Scheme Bricks and Gibber have the potential to accentuate appreciation of the rhythms running through both algorithmic processes and generated music as they unfold.

16.3.2 Privileging Legibility

A number of live coding syntaxes have been designed to be understandable by audiences (Freeman and Van Troyer 2011; Magnusson 2011b; McKinney 2014). Despite this being a design concern when creating the environment LOLC, Jason Freeman notes: 'Audiences, for the most part, did not understand the code fragments as they were projected on the screen; the natural musical terminology used for durations and dynamics was of little help to them. There is, perhaps, an inherent design paradox here. Performing musicians tend to prefer a concise syntax that requires minimal typing, whereas audiences tend to have difficulty understanding text that is not sufficiently verbose' (Freeman and Van Troyer 2011). Freeman goes on to note that the ensembles using LOLC obtained better audience responses from projecting text messages passed between ensemble members in LOLCs built-in chat room, a practice first performed by The Hub (Brown and Bischoff 2002). In order to help associate performers with their text messages, each LOLC performer wore a baseball cap that matched the color of their messages in the chat room. Such messages can be considered a type of meta-score, where ensemble members are considering and debating the direction of a performance; in this role the chat dialogues can convey a good deal of musical structure to an audience.

Perhaps there is a need for a live coding environment that considers audience legibility its primary motivation, so that the limits of legibility in the genre can be more fully explored. Such an environment would not only consider the language syntax from the perspective of the audience but also require visual annotations for every expression evaluated, as suggested by Georg Essl in a 2014 email to the TOPLAP (live coding) mailing list, where he hypothesized a language in which: 'all run-time states/consequences induced by code must have a visualizable code-side representation'. As a simple yet thought-provoking example, Andrew Sorensen gave an inspiring live coding performance at the 2014 OSCON conference, where he vocally narrated what each block of code was doing as he was writing and evaluating it, successfully conveying the algorithmic processes he developed. Could such a narrative be automatically generated and inserted into the source code as comments immediately above or below the lines of code they are associated with? Alternatively, a running commentary could be generated alongside the source code document via multiple autonomous commentators with different personalities and aesthetic sensibilities.

A recent survey on the visualization of state and timing in source code in Gibber found widespread support for improving visualization of algorithmic processes in live coding environments, with over 96 percent of the 102 participants believing that visualization techniques should be explored in other live coding environments (Roberts, Wright, and Kuchera-Morin 2015).

16.4 Abstractions

Many live coders begin performances with a blank screen or empty page; for some this is essential (Swift, personal communication 2015). No matter the degree to which an audience member can understand the code, the relation between the accretion of code and the emergence and development of music makes a dramatic communication of creative endeavour. This may serve a cultural agenda, evoking 'an understanding that anyone else could start from the same place' (Biddle 2014), or may simply be a commitment to improvisation. Regardless, being able to rapidly code from nothing to sound speaks to the remarkable strengths of live coding environments. However, as Bell notes this is hardly 'coding from scratch … since all coding rests upon layers of *abstractions*' (personal communication 2015); Swift echoes that 'with a few textual characters I can orchestrate *(with the right abstractions)* quite rich output material' (personal communication 2015; emphasis added). With these layers of abstractions—including interface capabilities, language features, and library code—no page is empty, no screen void; more pregnant with potential than an empty canvas, they are volatile and excitable by computation.

In this section we explore the tensions that arise due to the abstractions used in live coding performance, starting from computational abstractions and moving towards musical abstractions. But first, a quick definition: abstraction (literally, 'drawing away') is the process of extracting from a diversity of concrete specific examples general rules, patterns, or concepts no longer dependent upon those examples, by eliding or suppressing some of their specific differences and details.

Computational abstractions suppress lower-level complexities in order to establish higher-level interaction. By doing so, we effectively sacrifice one class of affordances for another. Naturally, high and low are relative terms. For example, zmölnig (personal communication 2015) describes building synths from raw oscillators using the visual programming language PureData as 'low level', but this would be high level compared to a live coder operating on raw audio signals with bitshift operations. Bearing this in mind, we consider the gamut of abstraction levels live coders use, and why. Bell (personal communication 2015) is explicit in his preference for high-level affordances: 'I always seek higher and higher levels of abstraction … to quickly make sudden changes in my performance with minimal code. … I do not wish to program at lower levels of abstraction.' Meanwhile, Swift (personal communication 2015) argues for a broader spectrum of affordances: 'I like low-level control of memory since I want to signal-rate stuff (e.g. DSP) as well as note-level stuff. But from a [programming language] perspective, I love higher-order functions/closures, and prefer to map & reduce rather than [to] use a for loop.' Indeed, one of the attractions of higher-level abstractions is their generality: by being so abstract, the very same tools can utilized in a wider diversity of concrete applications. Moreover, to the extent that such high-level abstractions are *composable*—that they can be flexibly combined—they also effectively reduce the number of primitive concepts that need to be borne in mind.

Several live coding environments place quite low-level mechanics of the machine within reach during performance. Brown and Sorensen (2009) agreed that in general 'there is no possible way for us to deal with the complexity of the underlying operating system and hardware without levels of abstraction', yet Sorensen (2014) endeavoured to make Extempore 'efficient enough, and general enough, to make its own audio-stack implementation available for on-the-fly code modification at runtime.' Though coding 'down to the metal' of an audio driver seems unlikely to be needed during a performance, the ability for high-level code to directly manipulate low-level implementations can certainly improve performance through dynamic compilation of high-level algorithms into native machine code,[2] and thus increase the quantity, diversity, and complexity of the algorithms that can effectively be coded live (Sorensen 2014; Wakefield 2012).

16.4.1 Discourse with Models and Metaphors

Abstractions are not merely structural convenience: through their constraints and affordances, abstractions effectively present a *model* of a world with which a live coder maintains discourse (Rohrhuber, Campo, and Wieser 2005). In the case of musical performance, this model serves as 'a scaffold for externalising musical thinking' (Magnusson 2014b).[3] McLean suggests that by matching abstractions to the semantic needs of the problem world 'we are more able to engage the right kind of cognitive load, and become more absorbed in programming as a live experience' (2014a). For Magnusson, the cognitive load should be musical rather than a computational (Magnusson 2014b). Consequently, many live coders have explored embedding mental models of musical structure and process into abstractions at the level of programming languages themselves, resulting in domain-specific mini-languages, and sometimes even mini-languages within mini-languages. McLean's Tidal is a domain-specific language embedded within Haskell (a general-purpose functional language), but itself embeds a further mini-language for polyrhythmic cycles. Magnusson's ixi lang is a DSL for live coding with an embedded mini-language for agent scores, that is written in the audio programming language SuperCollider, itself derived from the general purpose SmallTalk language and implemented in C++. For Magnusson, the underlying power of SuperCollider was a pragmatic shortcut to providing ixi lang with rich sonic material. Although it remains possible to descend to the SuperCollider language live, the primary goal is a level of user-friendliness such that his language 'frees performers from having to think at the level of computer science, allowing them to engage directly with music through a high-level representation of musical patterns' (Magnusson 2010).

To be more easily understood (not *too abstract!*), these abstractions are often grounded in metaphor. Computer science has utilized metaphors of containers, privileges, inheritance, agents, and so on (Travers 1996), while music software has leveraged metaphors relating to instruments and scores as well as studio technology concepts such as mixers and patchable synthesizers. Not surprisingly these reappear in live coding. For example, ixi lang is designed around three simple concepts: *agents* use *instruments* to perform *scores* (Magnusson 2011a). Agents recur in Alive, to which *properties* and *tags* can be associated, and with which *queries* can be made (Wakefield et al. 2014). Tidal's pattern transformations have an explicit relation to weaving and knitting (McLean 2013a, 2014b). Language-level abstractions for live coding also often utilize notational analogies. The score language within ixi lang relates text-space to musical-time, in that 'spaces between the notes represent silence, spatial organization therefore becoming a primary syntax of the language' (Magnusson 2011a). Other examples include the iconic => and =^ operators used in ChucK, or the -> and >> operators in ixi lang, to denote the creation of connections between objects and projections into the future.

By creating model cognitive worlds, rich abstractions substantially influence what kinds of ideas can be expressed, and what discourse can ensue. Collins and McLean

(2014) note how many live coding interfaces incorporate looping and layer-centric patterning typical of dance music. The more closely an interface is fitted to the semantics of a particular model or metaphor, the more likely that it also becomes too rigid to be manipulated algorithmically outside of these assumptions, and thus can be described as *abstraction hating* in the Cognitive Dimensions of Notations framework (Blackwell and Collins 2005; Green 1989). Magnusson (2014b) suggests that, by affording certain practices and preventing others, the highly constrained abstract model of ixi lang should itself be considered 'a compositional form'. For Blackwell, 'every new live coding tool can become the starting point for a miniature genre, surprising at first, but then familiar in its likely uses' (2015, 59). Here becomes evident another tension of abstraction, in terms of *the complexity of freedom*. As Brown and Sorensen (2009) state, such influences have both positive and negative implications. While, as Magnusson notes, 'constraints inherent in the language are seen as providing *freedom from complexity*' (2014b; emphasis added), and that some 'users report that these limitations encourage creativity' (Magnusson 2011a), other live coders prefer interfaces that grant the *freedom to create complexity*. For example, live coder Shelly Knotts prefers the relative complexity of SuperCollider because she wants greater control of detail during a performance: 'Writing simple patterns and sequences can be complicated, but the complexity of what you can achieve is high. Languages such as ixi … sacrifice the micro level control' (personal communication 2015)

16.4.2 Temporal Abstractions

Let us focus in detail on a domain of abstraction of special relevance to live coding: the representation of time. Oft neglected in computer science (Lee 2009), temporality is vital to the domain of music; not surprisingly many live coding systems present musically oriented abstractions of time. For example, many live coders *measure* time in terms of *meter*. Renick Bell counts beats with floating point numbers in order to represent intervals or events between them; 'relevant functions return a value which is valid at any point within the requested span' (Bell, personal communication 2015). Rather than floating-point numbers, Tidal uses rational numbers in order to capture a musical conception of time that incorporates both cyclic repetitions and linear progressions. For example, 'a time value of 8/3 would be the point that is two-thirds through the third cycle' (McLean 2013b). Meter may also be part of the coding environment itself; for example, code fragments in Gibber can be triggered to automatically execute on the next bar. And in systems where time is primarily represented in terms of seconds or audio sample counts, performers can easily create (or reuse) 'metronome' abstractions in order to trigger things with a metric rhythm (Swift, personal communication 2015).

Parallelism is another important temporal musical concept, found in polyphony, polyrhythm, and ensemble performance, that has been addressed through live coding abstractions. The ChucK live coding environment provides a system for easily launching and running multiple routines concurrently. These routines, or 'shreds', support precisely timed pauses within their execution, expressed in code through the assignment of new values to the variable now—as if the code itself directed the passage of time! In reality the assignment yields the instruction flow (which is thus a *coroutine*[4]) to an underlying scheduler, which will later resume it at the precisely designated time. In LuaAV and Alive precisely scheduled coroutines present a more passive metaphor of 'waiting' or 'sleeping' to achieve the same result (Wakefield et al. 2014; Wakefield, Smith, and Roberts 2010).

The Extempore system and its precursor Impromptu provide a closely related vehicle for concurrency via *temporal recursions*. A temporal recursion is defined as: 'any block of code (function, method, etc.) that schedules itself to be called back at some precise future point in time' (Sorensen 2013). Where the coroutine does this by yielding mid-function, the temporal recursion does this by reinvoking the function's entry point by name, granting an easy opportunity to rewrite the function between recursions. Both approaches support complex procedural flows with precise timing. (With sufficiently low-latency audio drivers, these precisely scheduled intentions in musical time become practically indistinguishable from real clock time.)

The procedural approaches to time described so far distribute the control flow of step-by-step procedures within the flow of musical time. As control passes from one instruction to the next, time passes as interruptions to that control flow; interruptions that are accurately placed in musical time. In Tidal, there is no such control flow, and time is instead treated in such a way that it can be manipulated, for example stuttered or reversed, through successive application of functions. Tidal centres on *patterns*: functions that map a given time span to a set of events that occur within that span (where events themselves occur over time spans). Rather than specify imperative instructions which act upon the current state, behaviour is declared in terms of pure functions that can operate on multiple levels of abstraction. An interesting result of working with time spans rather than instantaneous time points is that the representation works for notionally continuous as well as discrete patterns (McLean 2013b). Declarative statements describe rules, relationships, or ongoing activities that operate over an indefinite, cyclic timeline. Because composing a Tidal pattern is a matter of composing behaviours, the temporal structure and contents of patterns can be manipulated without having to calculate their entirety. Furthermore, these compound behaviours may operate on time distinctly from the specific contents of events, and vice-versa.

As with the 'safety in numbers' of collaboration mentioned earlier, the computational abstractions of parallelism above provide another way of reducing risk in performance. Ideally, the unexpected termination of any individual 'shred' in ChucK, or any temporal recursion in Impromptu or Extempore, will not crash the system as a whole. (The capability is somewhat dependent on the use of code with minimal side effects; temporal recursions that launch and control other temporal recursions are more powerful but potentially more destructive.) Ben Swift, who has used both Impromptu and Extempore in performance, shares: 'The fact that I can *change* things during a performance is important, and the fact that I can *break* things is a corollary of this. Having multiple temporal recursions running in parallel (which I do a lot) mitigates the risk, since if I make an error in one of them the others keep going. This happens not infrequently' (personal communication 2015). Moreover, ChucKs dedicated performance environment provides a user interface for monitoring, starting, stopping and replacing shreds during performances (Wang and Cook 2004), and likewise performers using Extempore can monitor the state of running temporal recursions through graphical overlays (Sorensen and Gardner 2010).

16.5 Flow and Pace

Coding live in front of an audience can clearly be a stressful experience, not least 'at 2am at a dance club after a couple of beers'! (Guzdial 2014) Languages such as ixi lang are explicitly designed to reduce stress by lessening the cognitive load. However, in discussions at the 2013 Dagstuhl Seminar on live coding it was noted that 'cognitive load is not necessarily to be avoided' (McLean 2014a); it can be part of what makes improvisation enticing. For example, in an informal qualitative evaluation, performers described of one of our prior live coding environments as 'stressful', 'frustrating', and 'difficult', *while at the same time* also as 'playful', 'fun', and positively 'live' (Wakefield et al. 2014). Some degree of cognitive load is necessary to enter the deep engagement of *flow* (Csikszentmihalyi 1990), 'a happy medium somewhere between frustratingly difficult on one side, and distractingly boring on the other' (McLean 2014a). This brings us neatly to our next tension topic: the achievement of this happy medium through the meeting of the capabilities of the models of abstraction with the intentions and capabilities of the performer.

In particular, live coders have developed various techniques to rapidly express musical ideas and move from the empty page to the production of sound. In some cases this becomes essential, since 'a long slow build up doesn't work very well if it's a middle-of-the-night set' (Knotts, personal communication 2015). The simplest technique is to make use of materials prepared in advance, such as patterns, instruments, and presets that can be dropped in and reused on the fly. The core tension here is control versus pace. McLean notes that since many interesting-sounding signal-processing algorithms are nontrivial to author, triggering and manipulating pre-made material is 'essential for a performance with any kind of pace.' Nevertheless, overuse of prepared material is dissatisfying: 'with improvisation you can really mold things to fit the sound system and mood, and when I trigger something pre-made it just sounds flat and lifeless, and breaks the flow of the performance' (McLean, personal communication 2015).

16.5.1 Succinct Expression?

We noted above that some live coders design mini-languages oriented to the author's conceptual music spaces, but these languages (and their environments) are also frequently designed to support immediacy in the interface. Both are evident in McLean's summary: 'I have developed Tidal to make it fast to improvise the kind of music I like to make' (personal communication 2015). Language- and environment-level abstractions enable complex articulations to be made far more succinctly. And as language interfaces more closely resemble the abstract (musical) models in play, these succinct statements are (hopefully) the kinds of things performers want to express. In consequence, code edits can more closely correspond to rates of musical decision-making. For example, 'ixi lang is a live-coding system designed with the criteria that it be fast (maximum 5-second wait before some sound is heard)' (Magnusson 2011a), and for Conductive, 'minimal notation is important, and I make efforts to reduce line noise to allow code to be input and edited efficiently' (Bell, personal communication 2015).

Interestingly, succinct expression does not necessarily imply brevity of notation. It can also be achieved in the programming environment by making prepared 'snippets' of code accessible from minimal keypresses (Swift, personal communication 2015). For example, in Extempore and Impromptu, some quick-to-type fragments of code are automatically *macro-expanded* in-place into more detailed pre-prepared implementations. We note that this automation from terseness to verbosity constitutes *a reversal* of the customary use of software abstractions to reduce code length, code repetition, and *conceal* implementation details. (It certainly presents exceptions to the tendency observed in live coding performances that code rarely fills a single-page; Biddle 2014.) From a different perspective, however, it can be seen as simply sharing involvement between coder and machine at a later stage in the translation from abstraction to execution. In fact Brown and Sorensen, who perform together as aa-cell, argue against the encapsulation of abstraction (the affordances of 'minimal code') in favor of *descriptive transparency*, since 'when programmers make a decision to abstract code away into an abstract entity, a black box ... the ramification is that they no longer have the ability to directly manipulate the algorithmic description' (2009). By making more of the mechanics of an algorithm explicit in the code, they expose more affordances to manipulate its ongoing process. But in a counterargument, Bell (2013) points out that higher-level abstractions need not be opaque, referencing the example of higher-order functions taking functions as parameters (which allows internal structures to be redefined later), and furthermore that in many languages abstractions may be accessible for modification during performance without having to have their implementation shown at all times.

Some artists prefer to minimize reliance on prepared materials by instead following an almost developmental approach, in which a complex algorithm is achieved piecewise from simpler but sound-making components that can 'carry the musical progression until the larger algorithm is working', such that 'it starts producing (non)musical events early in its life-cycle' (zmölnig, personal communication 2015). David Ogborn echoes this sentiment, noting that gradual development from low to high levels of abstraction during a performance also 'helps the audience grasp the abstractions ... like changing a harmony by typing it out, then making that a sequence/pattern, then making it a sequence/pattern with some variable elements, then making those elements/sequences/patterns, etc.' (personal communication 2015).

16.5.2 Frantic Expression?

A concrete result of the study by Swift et al. (2014) is that the live coders performed an average of fifteen significant edits per minute. Considering that an average computer user transcribes at around thirty-three and composes text at around nineteen *words* per minute (Karat, Halverson, Horn, and Karat 1999), this is no leisurely activity! Indeed many live coders agree: 'I'm generally coding all the time' (Mclean 2015); 'i usually spend the entire performance coding (unless i have to spend some time debugging)' (zmölnig, personal communication 2015); 'my gut feeling is that I'm almost always on the keyboard, either navigating, inserting or deleting' (Swift, personal communication 2015). For others, however, typing is not so uniformly dense: 'Often I may only type in one line of code every minute, though in other parts of the performance I may be more frantic and type in 8 or 10 lines a minute' (Freeman, personal communication 2015). Of course live *coding* is not just live *typing*: 'I code continuously through a performance [but] I do not consider *thinking time* as not coding' (Bell, personal communication 2015). A performer's subjective attention is necessarily spread across multiple activities. For example, for Freeman, planning and listening form the principal activity (Freeman, personal communication 2015). Similarly for McLean: 'You also have to be fully aware of the passing of time. . . . I might not know what I will change, but I will know when. If I miss a deadline then I have to wait until the next opportunity . . . depending on what feels right' (personal communication 2015).

16.6 Time

This brings us to our final tension: the open-ended multiplicity of becoming yet ultimate constraint of being within time—slippery concepts, whether regarded from music, philosophy, or computing. For example, we often think of time in terms of the past, the present, and the future; and these concepts might appear to have clearly defined correspondences in languages equipped with temporal abstractions. At the limit, the present is that instantaneous step from one instruction to the next. The future would be the list of scheduled instructions (events in a sequenced pattern, callbacks for temporal recursions, resumptions for suspended coroutines, etc.). In a purely declarative functional language, this encompasses the entire *current* program. The recallable past would be all markings recorded into memory that remain accessible to ongoing or scheduled code, including materials prepared in advance of the performance. Yet already the analogy breaks down, since any *scheduled* future is already a marking made in the past. A less imperative perspective might separate the program *state* as containing the accessible past with the program *code* as a description of the future, since 'a program obviously is a plan of how something is supposed to happen, an anticipation of future events' (Rohrhuber, Campo, and Wieser 2005). This distinction can become complicated when considering systems such as feedback.pl (Ward et al. 2004), ixi lang (Magnusson 2011b), and Gibber (Roberts, Wright, and Kuchera-Morin 2015), in which events that modify code can be thrown into the future, and in which the program code updates itself to continually represent the most recent changes of state (i.e. the past). This brings our attention to the subtlety by which live coding not only operates in multiple *levels* of time, it operates in different *kinds* of time.

16.6.1 Code ≠ Execution

We can usually distinguish between the *program as description*, which is a plan for what happens in a world (i.e. the code), and the *program as process*, which is the actual unfolding history of the consequences of that plan (i.e. the execution). However, when programs are modified during execution, as in live coding, this distinction is subverted.

For example, Rohrhuber showed that this leads to a limit, which he characterizes as *algorithmic complementarity*, in attaining complete access and control over both code and execution, which forces us to choose between privileging continuity of a particular history, or privileging the accuracy of that history's representation in the code. More generally, semantic gaps between description and process can lead to unexpected or unpredictable behaviour.

It is the indeterminacy of interaction, not simply program modification, that creates the formal division between description and process. From a theoretical computer science perspective, a program that modifies itself noninteractively can be reduced to an equivalent non-self-modifying program due to its logical determinism. Only changes coming from *outside the system*, such as live coding edits, constitute irrational cuts that divide islands of rational history.[5] The self-modifying programs mentioned above (feedback.pl, ixi lang) are no exceptions: the effect of rewriting on the execution is deterministic, but the effect on the code *representation* impacts what kinds of actions the performer can and may take. There are theoretical precedents to draw upon here, such as the *interaction machine* model discussed by Wegner, which behaves perfectly like classical Turing machines between each interaction point, but indeterminately across interaction points; and thus becomes *more powerful* than a regular algorithm (and arguably, a regular Turing machine) (Wegner 1997). In fact, Turing proposed an extension of his famous theoretical machine by adding an infinite read-only data stream returning results of a question not computable from within—which he called an 'Oracle machine' (Turing 1939). From the perspective of the computer, the decisions made by live coders (such as, what would be musically interesting to do next?) are exactly this kind of stream.

As Rohrhuber concludes with regard to algorithmic complementarity, tensions between description and process are positive, in that not only do the 'surprises or frictions with intuition or convention inspire creative solutions', but more importantly this 'prevents live coding from becoming a *merely technical* problem' (Rohrhuber 2014a).

16.6.2 Spectra of Extended Presents

Rather than a past and future divided by a singular infinitesimal cut, in many ways our subjective experience of time can be better understood as a superposition of spectra of *presents of varying duration*. This can be illuminated by phrases in the *present progressive* tense such as 'what are you doing?', which could refer to spans as short as a handful of seconds up to presents comprising months. This perspective is particularly relevant to live coding, in which performer's actions can span a longer spectrum of duration than with acoustic instruments.

At the narrower end of the spectrum, most acoustic instruments respond to gesture within milliseconds, but this raw expressive immediacy is largely lost to code. To recoup this immediacy, some live coders have augmented their interfaces with rapid-trigger key-bindings, some have mapped hardware input devices to global symbols in code (Brown and Sorensen 2009), some have collaborated with instrumentalists (such as 'vocalists, thrash guitarists, drummers, and banjo players'; Stowell and McLean 2013), and some have performed with both code and instrument simultaneously (such as Dan Stowell's live coding while beatboxing, David Ogborn's live coding with guitar, or live coding with bio-sensors as performed by Marije Baalman 2013–2014).

At the broader end of the spectrum, whereas the effects of an acoustic gesture might persist for a handful of seconds, live coding can easily spawn processes that are indefinitely extended. In this regard the improvising live coder 'is primarily a composer, writing a score for the computer to perform' (Magnusson 2011a). Improvisation has always involved planning ahead and reflecting back at a range of musico-temporal scales, but rarely through such persistent automation. Swift (personal communication 2015) notes that with a few keystrokes a complex and never-ending sequence of events can ensue without effort, 'whereas on the piano I have to play each note directly. … I have to stay there or it goes quiet'. This freedom is echoed by Freeman (personal communication 2015): 'I'll sometimes schedule musical content to loop a lot … so I can get more going without having to type so much.' The connotation is that these extended durations create more space for performers to multitask: 'I gain the ability to affect simultaneous change across of number of processes that would be impossible with any traditional musical instrument' (Bell, personal communication 2015).

What is left unspoken is that, although activity is multiplied, attention is not (Swift, personal communication 2015, teases that he could even 'go and have a beer'). Infinite repetitions, even of repetitive conditionals (Collins and McLean 2014), are not the same as repeated efforts. In this regard, zmölnig mainly avoids the broader spectrum of long durations in order that *performance time* is forced onward through the performer's actions of coding, rather than the code alone (personal communication 2015).

Returning to our first tension, we close with the observation that in any case, throwing algorithmic plans of action into the further future is a risky venture. First, algorithms might not unfold in the way expected due to nondeterministic or probabilistic components or simply the unpredictability of generative complexity. Second, algorithms may be influenced to an unpredictably changing context. We noted earlier the absence of

DJ-style previewing in live coding, which we can identify as largely unrealistic: in comparison to the fixity of a DJ's audio file, the effect of a line of code can be radically different according to the stateful context in which it is executed—thus live coding becomes live debugging (Blackwell and Collins 2005). Third, projected plans might no longer be *musically* appropriate by the time the future comes around, and require immediate modifications to stay relevant. As Sicchio (personal communication 2015) contends, 'it is less about managing the time in terms of authoring and more about sensing what that moment in time needs'. This is compounded within an environment of collaboration: 'I can't predict what other musicians will do between now and the future and their decisions will inevitably make irrelevant whatever far-in-the-future plans I may have had' (Freeman, personal communication 2015). Finally, many live coders acknowledge that projecting code into the further future is just too attentionally demanding: 'I do not plan what kind of algorithms I want far in advance' (Goncalves, personal communication 2015); 'I still struggle to listen, reflect, and create across so much time … it's hard for me to think ahead while still listening to the present' (Freeman, personal communication 2015); 'With the diversity of time scales involved, I will occasionally do a bit of preparation if I have an idea in advance of when I want to put it into practice, but this only happens if I'm feeling very awake!' (McLean, personal communication 2015); and 'I don't tend to do so much future planning in performance … I often feel like I don't have enough time to sit back and listen because of time pressure to keep coding and making changes to the music' (Knotts, personal communication 2015).

16.7 Closing Remarks

Given the tensions we have described and the plurality of practices they generate, perhaps it is useful to close by returning to the why of live coding.

Blackwell and Collins (2005) raised the question why any live performer would choose the challenges of a programming language over the 'comfortable ride' of software such as Ableton Live. A suggested response was to escape the confines of stylistic bias in such mainstream application interfaces, and instead to embrace the vast exploratory potential for experimental music that full-fledged programming languages provide. A decade on however, we see plenty of domain-, style-, and even artist-specific bias in the live coding languages of today. Moreover, it is not clear that the unlimited potential of programming is profoundly utilized during performance. What, after all, differentiates live coding from noncoding in performance? The diversity of interfaces and interface abstractions evident in live coding practice suggests it is better to look to the underlying models of discourse that coding live make possible. Specifically, computational abstractions here include complex conditional control-flows, symbolic reasoning, and the ability to create *new abstractions*. Without these, it could be easily argued that the abstract model of discourse utilized is little different from adjusting patch cables and knobs of a modular synthesizer. But Magnusson notes that, according to such a 'strong' definition of live coding, many so-called live coding performances do not include coding at all (Magnusson 2014a).

Moreover, the contrast with modular synthesizers is unconvincing: not only do many live coders use visual interfaces directly inspired by modular synthesizers, but actual modular synthesis patching can be clearly articulated as a form of live coding, sharing many practical similarities with its digital counterparts (Hutchins 2015). Indeed, many live coders confirm that relatively little performance time is dedicated to the more abstract algorithmic complexities made possible by programming. (It could also be argued that performance of live circuit creation, such as by *The Loud Objects*, is a form of abstraction-free live coding.) For example, Knotts spends 'more time writing immediate updates than on algorithm design in performance, and in general only use very simple algorithmic control' (personal communication 2015). Too much anticipatory algorithmic work is at risk of losing the audience: 'I'd certainly rather watch a performance of parameter tweaking that gets moving quickly than wait minutes for someone to pull off some very complicated algorithm' (zmölnig, personal communication 2015). As Blackwell (2015) observes, complex algorithms are rarely explored in performance because an executing program's structure can only be changed *gradually*.

In contrast to the image of the live coder as modern concerto artist—'the virtuosity of the required cognitive load, the error-proneness, the diffuseness' (Blackwell and Collins 2005)—there is perhaps a greater interest in its simplicity and potential to democratize code in a fun, exploratory fashion. As Biddle (2014) notes, many performers 'want their audience to understand that the process is visible and reproducible: anyone can make music just like this. As the magician says, there's nothing up my sleeve.' Supporting this claim is a growing use of live coding in education. Sicchio (personal communication 2015) relates, 'I also recently have been teaching middle schoolers Sonic Pi to create music. They have performed several times including at their middle school dance,' pointing towards live coding performances that are the antithesis of virtuoso, concert hall performances.

And, why *not* code live? Canonical live coding emerged not long after programming languages such as SuperCollider made it possible to concisely define sounds and hear them expressed close to real time; which is to say at the point at which experimental approaches to code-based musical composition became possible during live performance. This suggests that programming and live performance have been thought of as separate only *by convention* (Rohrhuber 2014b). As Goncalves simply states: 'I choose to live code because I think writing a musical expression and hearing its result is very powerful' (personal communication 2015). Many composers would agree with this, and there is little reason besides cultural intractability to demote code beneath piano, pen, and paper as a compositional tool.

For those who code, compose, and perform, live coding combines these activities into a single gestalt, and by blurring boundaries it enables them to engage with the coincident juncture of all parts. For example, Freeman (personal communication 2015) states that his goal is to 'abstract the process of creation, transformation, listening, organizing, collaborating, and improvising in a structured way that mimics some techniques we often use to do these activities in other forms of music'. Combining all three of these activities comes with its own costs, but these are potentially appealing in and of themselves.

Since its inception, live coding has progressed from something with technological and cultural promise to what is now a worldwide set of communities, regular performances, workshops, and international conferences. As it continues to evolve, so inevitably will its tensions, providing space for continued experimentation. Live coding performers are free to memorize code employing no algorithms whatsoever, type it verbatim during a performance, and not project their code to the audience. They can also improvise algorithms that in turn improvise music while potentially revealing intricate details of their compositional process to the audience via source code projection. Although some practices might not take full advantage of the medium, this alone does not determine aesthetic results. Which brings us to the members of the audience, who are free to choose how they engage with these performances. They can attempt to understand the algorithms created during a performance or ignore projected code in favor of appreciating the resulting music, leaving live coding performances to be both produced and consumed in an absence of absolutes.

Acknowledgements

This chapter was supported by the Social Sciences and Humanities Research Council, and the Canada Research Chairs program, Canada; the Robert W. Deutsch Foundation, and the Media Arts and Technology Program at the University of California Santa Barbara. Special thanks to the live coding performers who completed our original survey: Renick Bell, Jason Freeman, Tanya Goncalves, Shelly Knotts, Alex McLean, David Ogborn, Kate Sicchio, Ben Swift, and IOhannes zmölnig. We'd additionally like to thank Alex McLean for his helpful suggestions in regards to this text.

Notes

1. See for example the list compiled at http://iclc.livecodenetwork.org/2015/definitions.html.
2. The related principle of 'dogfooding'—building a live coding system's libraries from within its own language—has also shown its value in performance (McLean and zmölnig, personal communication, through online survey: http://www.charlie-roberts.com/live_coding_survey 2015).
3. In a formal study based on the transcription of thirteen live coding performances using Impromptu, Swift, Sorensen, Martin, and Gardner (2014) found that coding actions were relatively consistent between performers, whereas musical activities showed much greater divergence. Although limited in scope, the study may serve to highlight the semantic gap between the coding environment and the musical models of the performers.
4. Coroutines are a procedural representation of collaborative, single-thread multitasking. In computer science terms, they are one-shot continuations. In layman's terms, they create the illusion of a procedural function that can be paused in the middle and resumed later. Note that tasks are not truly concurrent in that only one task can be executing at any given moment.

5. We borrow the term 'irrational cut' from Deleuze's treatment of nonclassical cinema, where it refers to cuts that subvert the otherwise rational narrative flow; an incommensurable link between shots that is not a member of the series that preceded it, nor of that which follows, and thus creates a direct presentation of time (Deleuze 1989). Deleuze appropriated the term from the concept of an irrational Dedekind cut in number theory, which belongs to neither of the sets it produces.

Bibliography

Aaron, S., Blackwell, A. F., Hoadley, R., and Regan, T. 'A Principled Approach to Developing New Languages for Live Coding'. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 381–386. Oslo, 2011.

Alperson, P. 'On Musical Improvisation'. *Journal of Aesthetics and Art Criticism* 43, no. 1 (1984): 17–29.

Baalman, M. *Wezen-Gewording*. 2013–. https://www.marijebaalman.eu/?p=404.

Bell, R. 'Pragmatically Judging Generators. ~~Live Performance: Improvisation~~'. In *Proceedings of the Generative Art Conference*, 221–232. Milan, 2013.

Bell, R. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 5 April 2015.

Biddle, R. 'Clickety-Click: Live Coding and Software Engineering'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014): 154–159.

Blackwell, A. F. 'Patterns of User Experience in Performance Programming'. In *Proceedings of the First International Conference on Live Coding*, 53–63. Leeds: ICSRiM, University of Leeds, 2015.

Blackwell, A., and Collins, N. 'The Programming Language as a Musical Instrument'. *Proceedings of Psychology of Programming Interest Group* 3 (2005): 284–289.

Brown, A. R., and Sorensen, A. 'Interacting with Generative Music through Live Coding'. *Contemporary Music Review* 28, no. 1 (2009): 17–29.

Brown, C., and Bischoff, J. *Indigenous to the Net: Early Network Music Bands in the San Francisco Bay Area*. 2002. http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html. Accessed 20 June 2017.

Bruun, K. *Skal vi danse til koden*? *Kunsten.nu*, 18 November 2013. http://www.kunsten.nu/artikler/artikel.php?slub+livekodning+performance+kunsthal+aarhus+dave+griffiths+alex+mclean+algorave.

Clowney, D., and Rawlins, R. 'Pushing the Limits: Risk and Accomplishment in Musical Performance'. *Contemporary Aesthetics* 12 (2014).

Collins, N. 'Live Coding of Consequence'. *Leonardo* 44, no. 3 (2011): 207–211.

Collins, N. *Origins of Live Coding*. 1 April 2014. http://www.livecodenetwork.org/files/2014/05/originsoflivecoding.pdf.

Collins, N., and A. McLean. 'Algorave: A Survey of the History, Aesthetics and Technology of Live Performance of Algorithmic Electronic Dance Music.' In *Proceedings of the Conference on New Interfaces for Musical Expression*, 355–358. London, 2014.

Csikszentmihalyi, M. *Flow: The Psychology of Optimal Experience*. New York [illegible]perPerennia l, 1990.

Deleuze, G. *Cinema 2: The Time-Image*. Translated by H. Tomlinson and R. Galeta. Minneapolis: University of Minnesota Press, 1989.

Della Casa, D., and John, G. 'LiveCodeLab 2.0 and Its Language LiveCodeLang'. In *Proceedings of the Second ACM SIGPLAN Workshop on Functional Art, Music, Modeling and Design*, 1–8. New York: ACM, 2014.

Freeman, J. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 18 March 2015.

Freeman, J., and Van Troyer, A. 'Collaborative Textual Improvisation in a Laptop Ensemble'. *Computer Music Journal* 35, no. 2 (2011): 8–21.

Goncalves, T. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 19 June 2015.

Green, T. R. 'Cognitive Dimensions of Notations.' In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, 443–460. New York: Cambridge University Press, 1989.

Griffiths, D. 'Fluxus'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014): 149–150.

Guzdial, M. 'Live Coding, Computer Science, and Education'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014): 162–165.

Hutchins, C. C. 'Live Patch / Live Code.' In *Proceedings of the First International Conference on Live Coding*, 147–151. Leeds: ICSRiM, University of Leeds, 2015.

Karat, C.-M., Halverson, C., Horn, D., and Karat, J. 'Patterns of Entry and Correction in Large Vocabulary Continuous Speech Recognition Systems.' In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 568–575. New York: ACM, 1999.

Knotts, S. 2015. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 31 March 2015.

Lee, E. A. 'Computing Needs Time'. *Communications of the ACM* 52, no. 5 (2009): 70–79.

Lee, S. W., and Essl, G. 'Communication, Control, and State Sharing in Networked Collaborative Live Coding'. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 263–268. Goldsmiths, University of London, 2014.

Magnusson, T. 'Designing Constraints: Composing and Performing with Digital Musical Systems'. *Computer Music Journal* 34, no. 4 (2010): 62–73.

Magnusson, T. 'Algorithms as Scores: Coding Live Music'. *Leonardo Music Journal* 21 (2011a): 19–23.

Magnusson, T. 'ixi lang: A SuperCollider Parasite for Live Coding'. In *Proceedings of the International Computer Music Conference*, 503–506. Huddersfield: International Computer Music Association, 2011b.

Magnusson, T. 'Herding Cats: Observing Live Coding in the Wild'. *Computer Music Journal* 38, no. 1 (2014a): 8–16.

Magnusson, T. 'ixi lang'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014b): 150.

McKinney, C. 'Quick Live Coding Collaboration in the Web Browser'. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 379–382. Goldsmiths, University of London, 2014.

McLean, A. 'The Textural x'. *xCoAx 2013: Proceedings of Computation Communication Aesthetics and X*, 81–88. Porto: Universidade do Porto, 2013.

McLean, A. 'Stress and Cognitive Load'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014a): 145–146.

McLean, A. 'Textility of Live Code'. In *Torque 1: Mind, Language and Technology*, 141–144. Link Editions.

McLean, A. 'Tidal: Representing Time with Pure Functions'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014c): 142–145.

McLean, A. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 6 April 2015.

McLean, A., Griffiths, D., Collins, N., and Wiggins, G. 'Visualisation of Live Code'. *Proceedings of Electronic Visualisation and the Arts 2010*, 26–30. London, 2010.

Nilson, C. 'Live Coding Practice'. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 112–117. New York, 2007.

Ogborn, D. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 7 April 2015.

Ogborn, D., Tsabary, E., Jarvis, I., Cárdenas, A., and McLean, A. 'Extramuros: Making Music in a Browser-Based, Language-Neutral Collaborative Live Coding Environment'. In *Proceedings of the International Conference on Live Coding*, 163–169. Leeds: ICSRiM, University of Leeds, 2015.

Roads, C. 'The Second STEIM Symposium on Interactive Composition in Live Electronic Music'. *Computer Music Journal* 10, no. 2 (1986): 44–50.

Roberts, C., Wright, M., and Kuchera-Morin, J. 'Beyond Editing: Extended Interaction with Textual Code Fragments'. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 126–131. Baton Rouge, LA, 2015.

Roberts, C., Wright, M., Kuchera-Morin, J., and Höllerer, T. 'Gibber: Abstractions for Creative Multimedia Programming'. In *Proceedings of the ACM International Conference on Multimedia*, 67–76. New York: ACM, 2014.

Roberts, C., Yerkes, K., Bazo, D., Wright, M., and Kuchera-Morin, J. 'Sharing Time and Code in a Browser-Based Live Coding Environment'. In *Proceedings of the International Conference on Live Coding*, 179–185. Leeds: ICSRiM, University of Leeds, 2015.

Rohrhuber, J. 'Algorithmic Complementarity, or the Impossibility of "Live" Coding'. In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014a): 140–142.

Rohrhuber, J. 'SuperCollider and the Just In Time Programming Library'. In *Collaboration and Learning through Live Coding*, Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014b): 150–151.

Rohrhuber, J., de Campo, A., and Wieser, R. 'Algorithms Today: Notes on Language Design for Just in Time Programming'. In *Proceedings of the International Computer Music Conference*, 455–458. Barcelona, 2005.

Rohrhuber, J., de Campo, A., Wieser, R., van Kampen, J.-K., Ho, E., and Hölzl, H. 'Purloined Letters and Distributed Persons'. In *Proceedings of the Music in the Global Village Conference*. 2007. http://www.wertlos.org/articles/Purloined_letters.pdf.

Sicchio, K. 'Hacking Choreography: Dance and Live Coding'. *Computer Music Journal* 38, no. 1 (2014): 31–39.

Sicchio, K. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 22 June 2015.

Sorensen, A. 'The Many Faces of a Temporal Recursion'. 2013. http://extempore.moso.com.au/temporal_recursion.html.

Sorensen, A. 'Extempore.' In *Collaboration and Learning through Live Coding*. Dagstuhl Seminar 13382, *Dagstuhl Reports* 3, no. 9 (2014b): 148–149.

Sorensen, A., and Gardner, H. 'Programming with Time: Cyber-Physical Programming with Impromptu'. *ACM Sigplan Notices* 45, no. 10 (2010): 822–834.

Stowell, D., and McLean, A. 'Live Music-Making: A Rich Open Task Requires a Rich Open Interface.' In *Music and Human-Computer Interaction*, edited by S. Holland, K. Wilkie, P. Mulholland, and A. Seago, 139–152. London: Springer, 2013.

Swift, B. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 23 March 2015.

Swift, B., Sorensen, A., Gardner, H., and Hosking, J. 2013. 'Visual Code Annotations for Cyberphysical Programming'. In *Proceedings of the 1st International Workshop on Live Programming*, 27–30. Piscataway, NJ: IEEE Press, 2013.

Swift, B., Sorensen, A., Martin, M., and Gardner, H. 'Coding Livecoding'. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, 1021–1024. New York: ACM, 2014.

Travers, M. D. *Programming with Agents: New Metaphors for Thinking about Computation*. PhD dissertation, Massachusetts Institute of Technology, 1996.

Turing, A. M. 'Systems of Logic Based on Ordinals'. *Proceedings of the London Mathematical Society* 2, no. 1 (1939): 161–228.

Wakefield, G. *Real-Time Meta-Programming for Interactive Computational Arts*. PhD dissertation, Media Arts and Technology, University of California at Santa Barbara, 2012.

Wakefield, G., Roberts, C., Wright, M., Wood, T., and Yerkes, K. 'Collaborative Live-Coding Virtual Worlds with an Immersive Instrument'. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 505–508. Goldsmiths, University of London, 2014.

Wakefield, G., Smith, W., and Roberts, C. 'LuaAV: Extensibility and Heterogeneity for Audiovisual Computing'. In *Proceedings of the Linux Audio Conference*, 31–38. Utrecht: Hogeschool voor de Kunsten, 2010.

Wang, G., and Cook, P. R. 'On-the-fly Programming: Using Code as an Expressive Musical Instrument'. In *Proceedings of the Conference on New Interfaces for Musical Expression*, 138–143. Hamamatsu, 2004.

Wang, G., Misra, A., Davidson, P., and Cook, P. R. 'CoAudicle: A Collaborative Audio Programming Space'. In *Proceedings of the International Computer Music Conference*, 331–334. Barcelona: ICMA, 2005.

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N., and Alexander, A. 'Live Algorithm Programming and a Temporary Organisation for its Promotion'. In *Read Me: Software Art and Cultures*, edited by O. Goriunova and A. Shulgin, 243–261. Aarhus: Aarhaus University Press, 2004.

Wegner, P. 'Why Interaction Is More Powerful than Algorithms'. *Communications of the ACM* 40, no. 5 (1997): 80–91.

zmölnig, I. Personal communication via online survey. http://www.charlie-roberts.com/live_coding_survey. Survey completed on 6 April 2015.