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.