Liveness in Notation Use: From Music to Programming

Luke Church
Computer Laboratory
Cambridge University
luke@church.name

Chris Nash
Computer Laboratory
Cambridge University
Christopher.Nash@cl.cam.ac.uk

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

Abstract

In this paper we draw an analogy between musical systems and programming environments, concentrating on user experience associated with feedback and its implications for flow. We present a number of different analytical frames all of which, we suggest, influence the nature of this feedback and with it, the user experience. We introduce a new diagrammatic analysis format, and use it to explore the kinds of feedback loop present in musical systems, what such systems might teach us in the analytical description of programming languages, and vice versa.

Introduction

The value of user feedback when interacting with computer systems is well-known. The need for feedback is taught in standard textbooks (e.g. Preece, Sharp & Rogers 2007), undergraduate courses (e.g. Blackwell 2009), and professional best practice (e.g. Microsoft 2009). In mainstream user interface design, feedback is supported by the principles of direct manipulation (Shneiderman 1983). In the analysis of programming languages, as we discuss later, a variety of kinds of feedback are incorporated in Taminoto's characterisation of liveness (Tanimoto 1990). Furthermore, in recent analyses of domain specific programming for end-users, lack of feedback has been shown to be a key impediment to usability. (Church & Whitten 2009, Church et al. 2009)

However the concept of user feedback is not a simple continuum. By drawing an analogy between musical systems and programming IDEs (Integrated Development Environments), we can ask a number of questions about different aspects of feedback; the source of the feedback, the level of liveness of the feedback, and the trends over time as systems evolve.

We start this analysis with a discussion of feedback in the modern IDE, showing that Tanimoto's theoretical frame for types of liveness (Tanimoto 1990) is still relevant, and use it as the starting point of the analogy to musical systems. We then apply this analogy to liveness in programming and consider the different sources of feedback the programmer and the musician has access to.

Levels of liveness

Intuitively, liveness is an assessment of how 'responsive' a system is. When I perform an action, are my changes immediately apparent? Do I have to go through a number of auxiliary steps in order to understand the consequences of my actions? Liveness is a property both of the program notation, and also of its execution environment. (Tanimoto 1990)

- Level 1 liveness
  (informative; "ancillary")
  describes situations in which a visual representation is used as an aid to software design (Tanimoto was referring to a user document such as a flowchart, not a programming language). This provides a basic level of graphical representation, and can be made continuously visible, although mainly because of the fact that a paper document can be placed beside the screen, rather than on it.

- Level 2 liveness
  (informative, significant; "executable")
  describes situations in which the visual representation specifies a program that can be manually executed, possibly after compilation. This provides a basic kind of physical action mapping, in that modification of the representation will eventually change the behaviour of the program.
- Level 3 liveness
  (informative, significant, responsive; "edit-triggered")
  describes situations in which the representation responds to an edit with immediate feedback, automatically executing or applying the changes. This allows users to make rapid actions, and often (after noting the system response) an opportunity to quickly reverse an incorrect action.
- Level 4 liveness
  (informative, significant, responsive, live; "stream-driven")
  describes situations in which the environment is continually active, showing the results of program execution as changes are made to the program. This provides high visibility of the effect of actions.

Because it spans both notation and execution, the degree of liveness within a programming environment is not a single factor, but can vary significantly across different components. The user experience of feedback in a typical professional IDE, such as Visual Studio or Eclipse, is far from homogenous.

Some aspects have a high degree of live responsiveness which span multiple steps of the traditional programming and compilation cycle. For example, code completion (e.g. Visual Studio's *IntelliSense*), much loved by developers, offers real-time feedback on code, not only from its explicit feedback (suggesting options for code-completion) but from implicit feedback when the programmer notices errors in the code by virtue of the IntelliSense engine failing to correctly suggest possibilities. Whilst this is still very much feedback from the notation (the source code and the IDE) rather than the domain (the executing behaviour of the program) it is rapid, typically appearing in under a second, and brings some of the benefits of Level 3 liveness.

However, a professional programmer spends much of her time doing other things than entering new code. At the simplest level activities include compiling, debugging and deploying. The Level 2-live compile cycle of even a small application on a high-performance workstation can take in the order of 30s, potentially disrupting the programmers flow and requiring distracting context shifts of attention.

The story for debugging is equally variable, though step-through debuggers offer the ability to provide real-time feedback (Level 3 liveness), albeit at a much slower pace than typical execution. However the programmer may have to manually walk through a series of steps in order to put the system in the desired state for debugging. Tool support such as Inform 7's *Skein* (Nelson 2006) allows replaying of a set of pre-recorded steps, but this is still an unusual feature.

In order to look at the way the different levels of liveness affect the different aspects of the process of developing, we shall consider a domain where the importance of feedback has been appreciated for a long time, and where the boundaries of liveness levels are starker: music.

Feedback in music

In music, feedback comes in many forms – tactile, haptic, visual and, most importantly, aural. In acoustic performance, such feedback occurs in realtime, but when you move to the digital domain, a latency is necessarily introduced, to enable efficient buffered DSP processing. Nonetheless, the tolerable delay is stricter than typical program feedback (e.g. Nielsen 1993); only a few milliseconds can be distracting (Walker 1999).

Modern music production software, such as the *sequencer* or *digital audio workstation (DAW)*, is centred on the idea of recording such a performance, either in MIDI or audio, and exploits the provisionality of the digital domain to support experimentation and improvisation. However, after

capturing the initial realtime performance, editing is mediated through abstract visual notations, often entailing cumbersome WIMP interfaces (van Dam 1997) and overly-literalistic visual metaphors (Duignan 2004). Increasingly complex features and interactions slow down the feedback cycle, and interaction is driven by abstract feedback from the visual notation, rather than concrete feedback from the domain itself, such as sound.

The DAW can be seen as a reversal of the classical composition process, in which the composer creates the notation before the music is performed, and the musical structure may be sketched in abstract form before a more concrete instantiation is required (Sloboda 1985). Instead, sequencers are more suited to the scenario where the initial performance is more representative of the actual end product; the musical idea must be largely formed before the user interacts with the program. As such, it is the tight interaction and feedback cycle with the physical instrument (either acoustic or MIDI) that provides immediate musical feedback in the sequencer model, rather than with the notation itself.

Fluid and fluent user experiences have been identified as critical to the creative user experience (e.g. Norman 1993, Shneiderman et al 2005). Relatedly, the notion of 'flow' (Csikszenmihalyi 1990) has been closely linked with creativity, and advocates "direct and immediate feedback". Similarly, Leman (2007) argues that more "direct involvement" in the music can be afforded by fast feedback loops, which he calls *action-reaction cycles*.

Our own research has been investigating an alternative style of composition tool, which prioritises musical feedback over richer, graphical affordances. The *soundtracker* (e.g. Nash 2004) offers a spreadsheet-like interface, using a grid of text to describe musical phrases, where each cell has a fixed number of spaces to specify pitch, instrument, volume (or panning) and one of a variety of musical ornaments (or *effects*), for example: **C#5 01 64 D01** starts playing a note [C#] in octave [5]; instrument [01]; maximum volume [64]; with a slow [01] *diminuendo* [D]. Despite the unorthadox appearance of tracker notation (see Figure 1), the music produced by its users is quite conventional – from dance tracks and pop songs to film scores and orchestral symphonies. Using the computer keyboard, the musical text can be edited very efficiently, prompting some to liken the process to a form of "musical touch-typing" (MacDonald 2007).

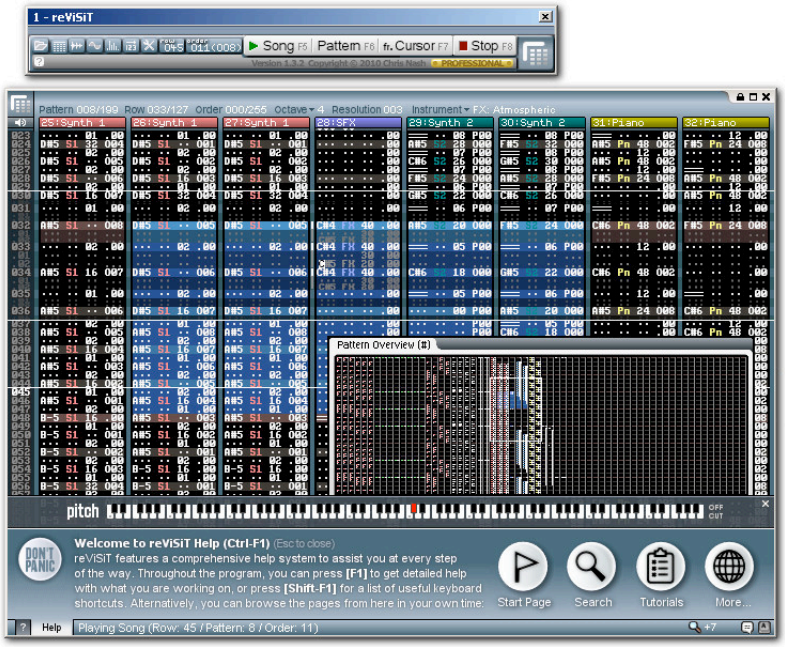
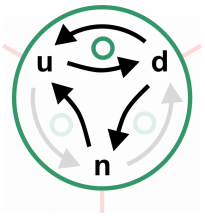
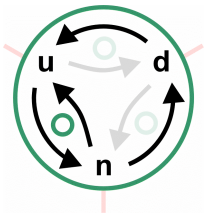


Figure 1 – The reViSiT soundtracker, a text-based music composition tool

Unlike the performance-based approach of sequencers, trackers encourage interaction with the notation, but crucially keep sound feedback close-at-hand – notes, phrases, parts or the whole song can be auditioned instantly, with a single keypress. In Figure 2, we illustrate this difference between the approaches, as a function of the feedback loops they produce.



(A) a performance-driven system, (e.g. a MIDI sequencer)



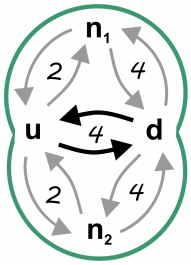
(B) a manipulation-driven system, (e.g. a soundtracker)

Figure 2 – Two computer music experiences, modelled as intrinsic feedback loops combining the user (u) and musical domain (d) with notation (n), connected by arcs representing common creative processes (performance, audition, data manipulation, visualisation, transcription and realisation).

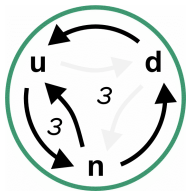
Performance-based systems as represented in Figure 2 (A) support flow by allowing episodes of tight feedback between the user (u) and the musical domain (d), and capturing the musical output of their instrument. Between these episodes, users must interact with some transcription of the performance, which may only represent a crude adumbration of the original performance (e.g. Figure 2 (A), in the case of DAWs). By contrast, manipulation-driven systems like trackers, as represented in Figure 2 (B), do not rely on such transcription, restricting the musical possibilities to those realisable in the notation, and driving interaction from sonic feedback from the encapsulated music, keeping the visual feedback between the user (u) and the notation (n) as direct, minimal, and tight as possible.

Adapting the earlier classification of liveness to our musical examples, we note that Level 4 liveness is that supported by "live" musical performance - where the effects of actions (on the instrument) are continuously and immediately *audible* - and, as such, we can clearly see its utility in sequencers and other performance-based music software. At the other end of the spectrum, Level 1 liveness can be seen in the offline notations used by composers, such as the sketching of ideas and musical processes on paper, and the ancillary visual representation presented by a sequencer's "Arrange Window".

The majority of other computer music scenarios are centred on editing a visual specification of what will happen in the music, as in both a sequencer/DAW's GUI and trackers, and thus such programs lie somewhere on a continuum between Level 2 and Level 3 liveness, based on the immediacy and quality of feedback provided. Sequencers are unable to sustain Level 4 liveness after the point at which the performance is captured, instead providing sub-devices (e.g. Arrange Window, Score Editor, Piano Roll, etc.) each allowing the visualisation and editing of specific and distinct aspects of the recorded data, where interaction is driven by visual feedback, and less frequently auditioned by spooling to the appropriate point and initiating playback. Although the music is then realised in realtime, the feedback is far from a continuous interpretation, and most often offers only Level 2 liveness, requiring the user to spool to an appropriate point and press play. Conversely, the rapid actions and prominent role of frequent sonic feedback, leads us to consider trackers more in terms of Level 3 liveness, where small edits are made and almost immediately auditioned - the execution is still triggered by the user, but the single keypress becomes *cognitively* automatic, following the edit.



(A) a digital audio workstation (DAW)



(B) a soundtracker

Figure 3 – Feedback loops in two music program types, annotated with levels of liveness.

In Figure 3, we annotate the feedback loops related to two scenarios in computer music interactions with the level of perceived liveness. DAWs (3A) do not have a central primary notation, instead offering numerous different tools to present different aspects a musical recording, splitting interaction between different editing features, views and windows, each using pointer-based direct manipulation and often triggering slow and a complex non-realtime ("offline") processes. By contrast, the tracker environment (3B) prioritises a single musical notation, and supports rapid sound feedback and a generally higher level of liveness, without supporting the Level 4 liveness of a direct performance.

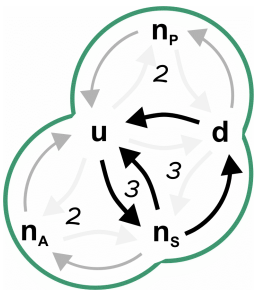
Towards Liveness in Programming

Level of Liveness	Programming	Music
1	flow chart, UML diagram	composer shorthand
2	code editor, compiler	score, sequencer/DAW GUI
3	code completion, syntax highlighting, realtime compilation, edit and continue	soundtracker, live coding
4	macro recording, Scratch, Data Canvas[1]	sequencer/DAW recording

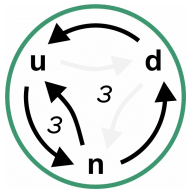
Table 1 - Examples of liveness levels in programming and music

Thus, in music, we can see that the tools strive to offer higher levels of liveness, supporting more fluid interactions, by tightening the feedback cycles between the notation and/or the domain. We can see the same trend in IDEs (as illustrated in Table 1). The introduction of *Edit-and-Continue* in Visual Studio, where a program can be paused, edited and then directly resumed, decreases the need for a full compile cycle after minor changes to the code, supporting Level 3 liveness. However, in other places we are moving in the opposite direction. The migration from compilers to in-line syntax checkers was an increase in liveness, but is now being supplemented by static analysis tools such as Coverity[2]. Currently, such tools tend to execute in a separate pass, being more a Level 2 tool.

Figure 4 shows the different cycles of feedback that occur in a typical modern IDE (4A) and a code editor (4B). We can see that the core interaction cycle between the users, the source code notation and the domain, is the same, with roughly the same degree of liveness, Level 3. However, the ancillary support tools, the profiler and the static analyser currently have lower liveness properties, Level 2.



(A) an integrated development environment (IDE) (e.g. Visual Studio)



(B) standalone code editor (e.g. EMACS)

Figure 4 – Two programming experiences, modelled as feedback loops combining the user (u), execution domain (d), source notation (n/n_s), profiler notation (n_p) and static analyser notation (n_a).

[1] introduced later
[2] www.coverity.com

There is a general theme here that when tools first get introduced they are at a relatively low level of liveness (e.g. Code Profilers are frequently found at Level 2 at the time of writing), but as technology progresses they migrate towards increasing liveness. This tendency lends weight to our suggestion that this is a crucial, but under-recognised, aspect of the user experience of programming, in that it seems to emerge in response to actual usage, rather than being designed-in at the outset.

Sources of Feedback: Notation, Domain

Previously, we have talked about the different sources of feedback that are available to the musician, the most obvious being the domain of sound itself. There is also feedback to be derived from the notation, in whatever form. This is similar to the experience of a programmer who gains feedback about their current task, say a programming or debugging activity, both from the executing behaviour of the program and from interactions with the notation, the source code and associated environment.

In most professional programming environments, these two worlds, the world of notation (source code) and of 'performance' (execution) are completely distinct. A common strategy for attempting to tighten the feedback loop is to batch-simulate the behaviour of small sections of the code in close to realtime, independent of the rest of the program. The Windows Presentation Foundation preview in Visual Studio does this, as does the Sprite view in Scratch (Malan & Leitner 2007).

Feedback in Scratch

The Scratch programming environment from MIT (ibid.), is one of the latest in a series of attempts to build educational programming environments for children, in which the main motivating element of the environment is that children are able to create their own videogames. There have been many other examples in the past of programming environments oriented toward videogame children, recently including Alice (Pausch et al 1995), Robertson and Good's AdventureAuthor (2005), and Microsoft's Kodu[3]. Scratch was explicitly motivated by a metaphor of media construction as musical improvisation, referring to the scratching techniques of turntable artists when they create new works from existing media. Many of these systems motivate children by providing rich libraries of media, artwork, and language primitives that can be rapidly composed into a satisfying result.

The feedback cycles in tools of this kind have two main effects. One is to maintain the level of motivation, by rewarding the child either with a functional product, or at least with a believable promise that a functional product is within reach. The other is to quickly correct faulty mental models of the system behaviour, in order that misconceptions about programming do not become entrenched. Both of these factors contribute to developing expertise. We believe that the same factors are likely to apply in adult development of expertise, although in children both are more dramatic (children generally have less patience, and are also more likely to acquire fundamental misconceptions).

A screenshot of the Scratch development environment can be seen in Figure 5. It shares the properties of many simple programming IDEs, and is very similar in overall structure to other recent instructional programming environments, such as Alice. It includes a) a live preview of the game display 'stage'; b) a list of those objects ('sprites') that appear on the stage; c) a canvas on which the behaviour of the sprite 'scripts' can be specified by assembling language primitives; and d) a navigation interface for finding and selecting primitives.

Most significant to our argument is the effort that has been devoted in this kind of environment to offering an experience in which the program is constantly 'running' (or using Tanimoto's term, 'live'). Any aspect of object behaviour can be evaluated at any time – whether a single language primitive or a whole script – to preview the effect it will have on the stage. An executing loop can be edited while it executes, and the next iteration of the loop will follow the new behaviour. All values can be inspected or modified at any time. No doubt it has taken substantial engineering effort to allow this much user freedom while maintaining a consistent execution state[4].

[3] http://research.microsoft.com/en-us/projects/kodu/

[4] From personal experience, and based on reports from its increasingly wide use in UK schools, Scratch is very robust.

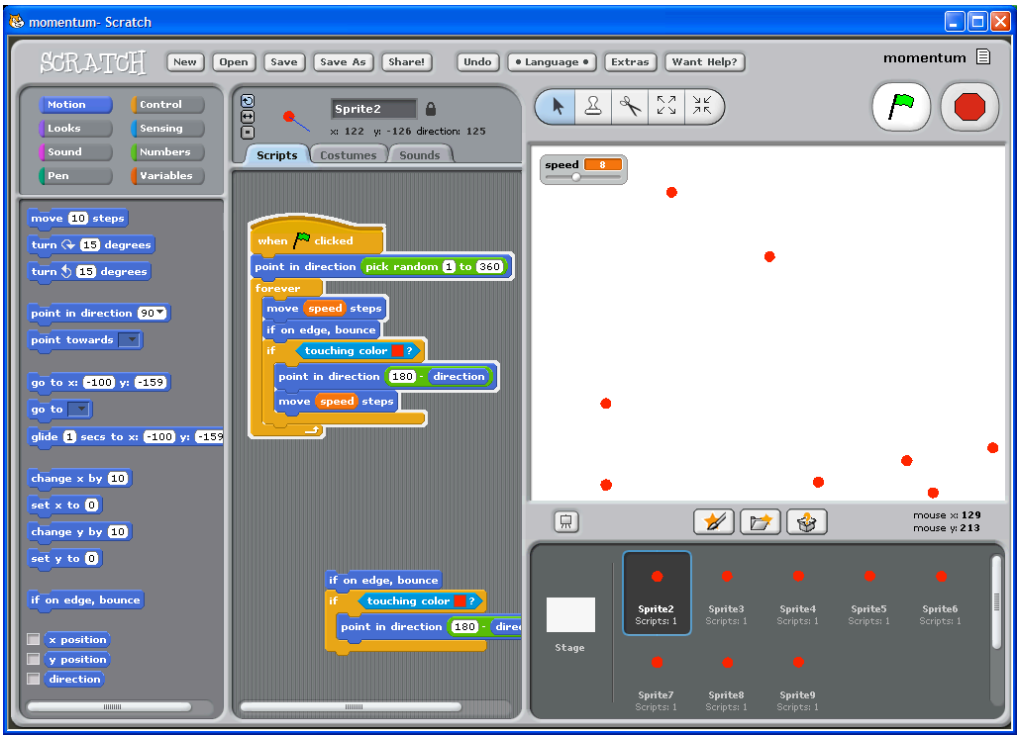


Figure 5 - The Scratch development environment

We find it interesting to reflect on the design principles that are apparent in this successful product. There are two kinds of direct visual feedback – sprites on the stage can be dragged with the mouse, and syntax elements can be moved around within the script window. However, both kinds of action also have effects on system state as a result of maintaining the consistency between the internal state of the execution engine and the direct manipulation interface. Dragging a sprite on the stage updates the current state of that sprite (potentially overriding state that resulted from script execution), but with the program continuing to run, so that it is possible to explore the effect of program execution from alternative screen states. Even if program execution is halted, the current position of a sprite is used as the default location for new operations added to the script – for example, allowing a character movement to be specified simply by dragging the sprite to the desired location, then inserting a move-to command in which the current location will have become the desired location.

These are examples of the close coupling between behaviour and notation that allow Scratch to provide feedback at multiple levels, and to maintain motivation during the acquisition of expertise.

Another challenge for abstraction

The increasingly close coupling of behaviour and notation that makes Scratch successful is, as we suggested earlier, part of a general progression of tools that work at the levels of the notation and the domain towards increasing liveness. An example is the range of refactoring tools, which started out as separate monolithic entities of the kind that many static analysis tools often are today, but have been progressively merged into mainstream interaction.

However, herein lays an interesting challenge for the psychology of programming. In many music production environments, a cumbersome visual metaphor fails to provide the kind of liveness experienced in trackers, where notation and domain are separate, but offer rapid feedback cycles. The abstraction manager/sub-device in an IDE constitutes an analogous obstacle. A common design manoeuvre in Cognitive Dimensions is to respond to viscosity by introducing an abstraction and an abstraction manager. However doing so typically encumbers the user interface and provides a slow rate of interaction, albeit more powerful interaction. Whilst initially from an Attention Investment (Blackwell 2002) point of view this may be a rational trade-off, we are suggesting that there is an

experiential difference, notably in the ease of achieving flow, between a small number of abstraction driven interactions and a large number of micro-interactions, as supported by (e.g. Resnick et al 2005). An interesting objective for programming usability is to achieve the same flow interaction, just at the higher level of abstraction – or "high-level hacking", as Thomas Green calls it (Green, personal communications).

One step along this path is to use technological innovations to minimise the cost of every operation. This is the approach the Data Canvas[5] takes. This project uses extensive pre-computation and cluster computing to achieve operations over statistical amounts of data at interactive speed, thus enabling the user to view and manipulate a statistical profile of their data in real-time. This, we predict, will have two effects:

1. It will increase the likelihood of the programmer achieving flow, by preventing disruptive delays and mode switching (e.g. Data Canvas, like Scratch, contains no notion of a separate mode for debugging)
2. It will allow the programmer to interact with emergent behaviours of their data through a large number of very rapid micro-operations. This will help decrease the premature commitment risk associated with abstraction over unknown data. (Church & Whitten 2009)

Sources of Feedback: Other Worlds

We have talked so far about feedback from the notation and the domain. However there are other sources of feedback that are important to both the programmer and the musician. First we have hinted above that there are analysis tools that do not form a strict layering of feedback but rather a graph. These feedback arcs can be extended beyond the technical, into the social (the feedback to the developer from the user in participatory design).

So the level of liveness of a technical ecosystem can be broken down into considering the properties of each sub-device and operation. Previous attempts to extend the analytic purchase of Cognitive Dimensions have struggled with the need to describe different notational 'levels' - structured representations that contain the same information, and can potentially be translated from one to another, but have different notational properties. The music analogy emphasises the ways in which the user's experience constitutes a web of interconnections between artefactual, cognitive, social and cultural structures. All of these can be represented digitally, and all offer different kinds of feedback between the computer and a user. We suggest that making the same kind of description for programming systems as for music systems may be a productive avenue for future work.

Conclusion

In this paper we have considered the liveness of a number of the interactive elements of musical systems and IDEs. We have observed structural correspondences between the systems and drawn analogies between the ways in which they influence their respective experiences of use. We discussed the trend towards increasing liveness in such systems, considering Scratch in detail. We introduced a diagrammatic model of the different places in which tightly coupled feedback might occur, and the way that this can be used to explain different kinds of usability relationship between notations and experienced domains. We concluded with a suggestion as to how this analysis may fix a difficulty with the Cognitive Dimensions' notion of levels. We believe that considering different kinds of feedback loop within musical systems provides new and interesting possibilities for the analytical description of the experience of programming languages.

[5] www.riversofdata.com

References

Blackwell, A.F. (2001). Pictorial representation and metaphor in visual language design. J. Visual Lang. Comput. 12, 3, 223--252.

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 (2009) Human-Computer Interaction (undergraduate course notes) http://www.cl.cam.ac.uk/teaching/0910/HCI/HCI2009.pdf

Carroll, J. M, Mazur, S. A. (1986) LisaLearning, Computer, v.19 n.11, p.35-49, Nov.

Church, L. and Whitten, A. (2009). Generative usability: security and user centered design beyond the appliance. In Proceedings of the 2009 Workshop on New Security Paradigms Workshop (Oxford, United Kingdom, September 08 - 11, 2009). NSPW '09. ACM, New York, NY, 51-58. DOI= http://doi.acm.org/10.1145/1719030.1719038

Church, L., Anderson, J., Bonneau, J., and Stajano, F. (2009). Privacy stories: confidence in privacy behaviors through end user programming. In Proceedings of the 5th Symposium on Usable Privacy and Security (Mountain View, California, July 15 - 17, 2009). SOUPS '09. ACM, New York, NY, 1-1. DOI= http://doi.acm.org/10.1145/1572532.1572559

Csikszentmihalyi, M. 1990. Flow: The Psychology of Optimal Experience. New York: Harper Perennial.

van Dam, A. 1997. "Post-WIMP User-Interfaces", in Communications of the ACM, 40(2):63-67. Association of Computing Machinery.

Leman, M. 2008. Embodied Music Cognition and Mediation Techology. Cambridge, MA: MIT Press.

MacDonald, R. 2007. "Trackers!", in Computer Music, 113:27-35. Bath, UK: Future Publishing, ltd.

Malan, D. J. and Leitner, H. H. 2007. Scratch for budding computer scientists. SIGCSE Bull. 39, 1 (Mar. 2007), 223-227. DOI= http://doi.acm.org/10.1145/1227504.1227388

Microsoft, 2009. User Experience and Interaction Guidelines for Windows 7 and Windows Vista. Available from: http://msdn.microsoft.com/en-us/library/aa511258.aspx

Nash, C. 2004. VSTrack: Tracking Software for VST Hosts. MPhil Thesis. Trinity College, Available from: http://vstrack.nashnet.co.uk.

Nelson, G. 2006. Inform 7. Available from: http://inform7.com/

Nelson, T.H. 1990. The right way to think about software design. In The Art of Human-Computer Interface Design. B. Laurel, ed. Addison Wesley, Reading, MA. 235-243.

Nielsen, J. 1993. Usability Engineering. Cambridge, MA: AP Professional.

Norman, D.A. 1993. Things That Make Us Smart. New York: Basic Books.

Pausch, R., Burnette, T. Capeheart, A. Conway, M. Cosgrove, D. DeLine R., Durbin, J. Gossweiler, R., Jeff S. K. Alice: Rapid Prototyping System for Virtual Reality White, IEEE Computer Graphics and Applications, May 1995

Preece, J., Sharp, H. Rogers Y. (2007) Interaction Design: Beyond human-computer interaction (2nd Edition).

Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., Selker, T., and Eisenberg, M. 2005. NSF Workshop Report on Creativity Support Tools, Available: Workshop on Creativity Support Tools, http://www.cs.umd.edu/hcil/CST/ [Accessed: 13 Apr. 2010].

Robertson, J. and Good, J. (2005). Adventure Author: An Authoring Tool for 3D Virtual Reality Story Construction. In theProceedings of the AIED-05 Workshop on Narrative Learning Environments, pp. 63-69.

Shneiderman, B. 1983. “Direct Manipulation: A Step Beyond Programming Languages”, in Computer, August 1983:57-69, Washington, DC: IEEE Computer Society.

Sloboda, J. 1985. The Musical Mind. Oxford, UK: Oxford Science Publications.

Tanimoto S.L. 1990. *VIVA*: A Visual Language for Image Processing. Journal of Visual Languages and Computing 1(2), 127-139.

Tanimoto S.L. 2003 Programming in a data factory. Proc. IEEE Symposium on Human Centric Computing Languages and Environments (*HCC*'03), pp.100-107.

Walker, M. 1999. “Mind the Gap: Dealing with Computer Audio Latency”, in Sound On Sound, April 1999. Cambridge, UK: SOS Publications Group.