Live coding for free

Alex McLean

ive Coding

Live coders write programs on the fly. They program in conversation with their machine, playing with instructions while a computer follows them. Here, there is no distinction between creating and running a piece of software – programs run while they are being created, gaining complexity via source code edits. We can think of coding live in the sense of working on electricity live, re-routing the flows of control around a program with the real danger that a faulty loop will get activated, causing the program to crash in sparks of logic.

Live coding environments make it possible to write code to generate music and video in real-time without having to restart any processes. Every adjustment to the code such as adding analgorithm to modulate a rhythmic back beat, or adjusting the inner workings of a video filter is immediately reflected in the audio or video output with no break.

It is not only the relationship between programmer and code that defines live coding, but also that between programmer and audience. Live coding can be a performance art, where an audience watches an artist write code while enjoying the output. As with all improvisations some preparation is necessary, but for many the aim is to begin with an empty text editor and live code their performance from scratch.

It can be difficult to pin down the definition of live coding. For example, what is the real difference between changing a number in a piece of source code and moving a slider in a graphical interface? The following points clarify some misconceptions underlying this kind of confusion.

Rules must be explicit. We may be inventing and changing rules all the time in our heads, but unless those rules are written down and modified while they are being followed by a computer (or other agent), that is not live coding.

Higher order functions must be defined and manipulated. A human musician could be described as an intricate, perhaps beautifully composed function that live codes itself. Here however we are interested in live coding within formal languages, with support for abstraction and composition.

An audience is not required. We can live code on our own, with a few friends or in a stadium, it is up to us (and our publicists).

Live coding is not an island. Live coders often perform with other kinds of musicians and indeed most live coders have musical training wider than only computer music.

Mechanics

Live coding of music, video and other time-based works presents a technical problem – how to dynamically change a running program without unwanted discontinuities in the output. There are many different solutions, with a few described below.

In Object Oriented Programming (OOP), *hot swapping* methods are common. A method is a responder to some kind of message, so it is quite straightforward to arrange for a message to arrive at an alternate, newly coded method.

In lisp-like languages, *temporal recursion* is a common form of live coding, for example as described by Sorensen and Brown (2007). These are functions which call themselves with some temporal delay, where functions may be replaced for the next call.

In the pure functional language Haskell, live coding is made possible through *state injection* (Stewart and Chakravarty, 2005). Here state is managed by monadic computations, and passed back to a static core during the reload of dynamic parts.

In SuperCollider there is something more like a a contemporary form of *conversational programming*, where an object oriented language is used to manipulate and communicate with synthesis graphs.

Source code feedback is an technique whereby a live coded program may make edits to its own source code (McLean, 2004). This is most useful in inserting comments to give the programmer feedback on the running of the program.

Live Coding Culture

> *Computers're bringing about a situation that's like the invention of harmony. Sub-routines are like chords. No one would think of keeping a chord to himself. You'd give it to*

> *anyone who wanted it. You'd welcome alterations of it. Subroutines are altered by a single punch. We're getting music made by man himself: not just one man.*
>
> *John Cage (1969).*

A number of fully fledged environments designed for live coding music and/or video have emerged in the past few years. The most well known are SuperCollider (McCartney, 2002), ChucK (Wang and Cook, 2004) and Fluxus (Griffiths, 2008), all three of which are FLOSS (Free/Libre/Open Source Software). The scheme-based Impromptu (Sorensen and Brown, 2007) is gaining traction, with freeware binaries available. We should also add Pure Data Puckette (1996) to the list; while the syntax is in the form of a graph, it remains a language with textual identifiers where live edits are the norm. The use of home brew environments is also common, often built around general purpose languages such as Perl, Python, Ruby and Haskell.

The communities around live coding environments are strong, with commonplace swapping of synthesis unit generators, scripts and patches. Here there is no clear line between software and music; by using someone else's synthesis library, you are in collaboration with them, making an audio collage of their technique and your own. The code sharing goes beyond synthesis libraries however. The snippets of code passing by email, demonstrating some technique or sharing a pleasing pattern, are perhaps equivalent to an oral tradition where nothing is fixed, just modified and passed on. By participating you contribute towards a cultural evolution of music.

Informal sharing of code is lifted into live improvisation in the performances of powerbooks unplugged (Rohrhuber et al., 2007). The players, of which there may be six or more, play together by passing SuperCollider patches over a wireless network. The code is shared via a simple chat system like interface, where themes may be collaboratively developed and call-response games played through source code edits. They avoid what they see as problems of artistic ego by rejecting any stage and playing as audience members, with the only sound emitting from their laptop speakers.

Instead of being reduced to commercial product, live coded music places human activity in focus. According to Small (1998), music is defined by all human activity around a performance, and the focus on music as a product rather than an activity itself is only a very recent

aberration. Powerbooks unplugged can therefore be seen as returning live coding to the roots of musical activity, and indeed the group themselves see their laptops, unplugged from any central sound system, as acoustic folk instruments.

Live Coding History Before Computers

Linguistically performative statements include 'I apologise', 'I promise' and 'We find the defendant guilty as charged.' Saying is literally doing. If the speaker has sufficient power, for example as an elected official, they may use performative statements to make rules which others must follow. To prevent things getting too messy, they will make rules about how future rules are made. The result is a system of law which includes how laws are made and changed, perhaps with the unchangeable nucleus of a constitution. Yes, we can view parliament as hardware, the law as software and politicians and voters as live coders.

However, examples within the realm of music are elusive – it does seem as though live coding of music began after the invention of computers. The experimental musicians of the 1960s explored rule based composition but even then did not, as far as we have heard, improvise those rules during performances.

Dynamic Programming

Outside of the context of time based arts, live coding is generally termed dynamic programming. It began in the form of bit twiddling – modifications of low level machine instructions while they were being followed. This was done for debugging, experimentation, and hackerly fun, although in the early days of computing, hands-on access to computers was hard to come by. The demand for dynamic edits continued with arrival of the classic languages Lisp, FORTH and Smalltalk, which are indeed still used for live coding today.

The term conversational languages (Kupka and Wilsing, 1980) gained some traction in the late 1970s, where a computer operator worked by typing a line of code, getting a response, and then typing a further line. It seems a shame that this term has fallen into disuse, but the idea is very much alive as what we now know as command line languages or shells. Indeed, the standard shells found on UNIX based operating systems are fully fledged programming languages.

Live Music

There are claims that The Hub were early live coders, although band members themselves do not make this claim. They performed with early networked computers from the mid 1980s, but did not make substantial live edits to running source code. They did, however, allow audience members to walk around them and see their screens, a position which has relevance as we will see.

Nonetheless, dynamic interpreted languages such as Lisp, Forth and HMSL were used to improvise music around this time, the earliest known example being by Ron Kuivila at STEIM, Amsterdam in 1985. Further work in establishing a historical record of live coding of this era is badly needed.

Live coding as a cultural movement grew from the release of SuperCollider 3 and ChucK. In Europe, the Changing Grammars meeting in 2004 was a particular turning point, where practitioners met to explore new possibilities of live coding. The attendees where largely from the SuperCollider community but members of slub where also present, including Adrian Ward who demoed his Map/MSG and PureEvents live coding software. An international link up of sorts was made with Ge Wang of ChucK, and so the cross platform live coding community was formed. All it needed was a name, later plucked out of the smoky air of a late night bar; TOPLAP (Ward et al., 2004).

Show Us Your Screens

In contrast to musical instruments directly plucked, struck and bowed by a human performer, the movements of a live coded performance all happen inside a computer. The live coder may be typing furiously, but it is the movement of control flow and data inside the computer that creates the resonances, dissonances and structure of music.

It is no surprise then that many live coders choose to project their screens, so that the audience may see something of the music production. This live freeing of software as part of an improvised performance might itself be considered avant garde.

Whether the audience is expected to follow the projected code is an open question. We may see a guitarist move their fingers across a fret board, and feel closer to the music as a result, but do we need to follow

and understand what these movements mean? Absolutely not. But there remains a strong sense that the performance is opened up to us by being able to see the movement behind it.

No wonder then that we hear anecdotes of audience members feeling alienated by laptop performances where screens are hidden. They could close their eyes, but couldn't they then have stayed at home, listening to a recording? However, many question whether projection of screens distract from a musical performance. If an audience can see the code behind a performance they may feel obliged to read it, moving their attention away from the sounds produced from it. Worse, if they do not know the programming language in use, then they may end up feeling just as alienated.

There is a logic then behind overlaying, where several screens are projected on top of one another, for example as practised by the live coding band slub. This amounts to obfuscation, where the audience can see fragments of code but no overall picture. They can observe the music being made, but their attention is directed away from the detail and instead to more important things such as the music, their neighbour or their drink.

Another alternative is to try to make the language more understandable to a lay audience, for example by using language with a simplified syntax, choosing highly descriptive variable names and so on. Griffiths (2008) takes this approach to artful proportions, making highly colourful miniature live coding environments embedded in his 'Fluxus' live coding language, controlled entirely from a gamepad. One example is 'Al-Jazari', where on-screen musical robots are controlled from a set of instructions specified using icons, including jump instructions for Turing completeness.

Licensing

These projected screens allow ad-hoc software distribution. We can compare live coding culture to the hacker culture around early computers as characterised by Levy (2002), in that freedom is the unchallenged norm. No thought is given to licenses of performance code, instead permissive and respectful sharing of algorithmic ideas and techniques is assumed. That an audience member could photograph the screen, download a interpreter that night and play with the algorithms themselves adds something to the atmosphere around the music.

Whether this free atmosphere will persist, or will be tempered by future commercial involvement is unsure. However, even within free culture, issues of copyleft will undoubtedly rear their head. Projecting a screenful of code constitutes 'propagation' in the terms of licenses such as the GPL. Indeed producing live music via the ’fixed form’ of source code could have wider ramifications for copyright . Perhaps we should enjoy the free sharing idealism while it lasts.

Coder Creativity

The practice of programming is informed by the corporate world of business software, with its talk of formal design, unit testing and ISO quality assurance. This all attempts to drive the creativity out of programming so that software may be as predictable as possible. The result is a cultural role of programmer as implementer and facilitator rather than creative individual.

This can lead to the bizarre situation where programmers make commercial software which practically generates music, and yet somehow the users of the software are seen as being more creative than the programmers. Here the programmers encode their musical style in the software, and the users do little beyond guiding the software to a destination pleasing to them. This can be seen in filters and plugins of music studio software as well as explicitly generative commercial applications such as Sseyo Koan Pro. The creativity of programmers is tapped into flattery of paying users.

It has to be said that this commercially-driven culture at times influences FLOSS music software culture, where programmers work to produce musical interfaces non-paying users. There are dark moments when free software is accused of mimicking commercial software with some justification.

With live coding, everyone is a programmer. There is understanding and respect that end user programmers have for those developing live coding language environments that is very different to that users have for anonymous brands of closed source software.

Perhaps this is an area where live coding can contribute something back to FLOSS culture in the form of alternative role models. Instead of stifling early enthusiasm of young programmers with vocational training, ad-hoc human creativity with all the mess of dynamic, serendipitous explorations can be encouraged and supported.

Bibliography

John Cage. John Cage: Writer, chapter Art and Technology, Cooper Square Press, 1969.

Dave Griffiths, Fluxus - a rapid prototyping, livecoding and playing/learning environment for 3d graphics and games. online; http://pawfal.org/fluxus/, 2008.

Kupka and Wilsing, Conversational Languages. John Wiley and Sons, 1980.

Steven Levy, Hackers: Heroes of the Computer Revolution. Penguin Putnam, January 2002.

James McCartney, 'Rethinking the computer music language: Supercollider', Computer Music Journal, 26(4):61–68, 2002.

Alex McLean. 'Hacking perl in nightclubs', 2004, http://www.perl.com/pub/a/2004/08/31/livecode.html,

Miller Puckette. Pure data: another integrated computer music environment. In In Proceedings, International Computer Music Conference, pages 269–272, 1996.

Julian Rohrhuber, Alberto de Campo, Renate Wieser, Jan-Kees van Kampen, Echo Ho, and Hannes Hölzl, Purloined letters and distributed persons, 2007.

Christopher Small. Musicking: The Meanings of Performing and Listening (Music/Culture). Wesleyan, June 1998. ISBN 0819522570.

Andrew Sorensen and Andrew Brown, 'Aa-cell in practice: an approach to musical live coding', Proceedings of the International Computer Music Conference, 2007.

Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In Proceedings of the ACM SIGPLAN Workshop on Haskell. ACM Press, September 2005.

Ge Wang and Perry R. Cook. On-the-fly programming: using code as an expressive musical instrument. In NIME '04: Proceedings of the 2004 conference on New interfaces for musical expression, pages 138–143. National University of Singapore, 2004.

Adrian Ward, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander, Live Algorithm Programming and a Temporary Organisation for its Promotion 2004.