LIVE CODING: AN OVERVIEW

IOhannes m zmölnig *Gerhard Eckel*
University of Music and Dramatic Arts, Graz
Institute of Electronic Music and Acoustics

ABSTRACT

Within the last few years a new performance practice has established itself in the field of intermedia art including computer music: Live Coding. By this a media performance is understood, where the performers create and modify their software-based instruments during the performance. One promise of this technique has been to give performers a way of improvising within the realm of algorithms rather than notes, while at the same time giving the audience a primary perception of the the algorithms used by means of projecting the source code, as opposed to the secondary perception of algorithms in traditional computer music by means of music alone. Another aspect is probably the development and demonstration of a technical mastery of the instrument "software".

In this paper we try to give an overview on the evolution of Live Coding within the last decade.

1. INTRODUCTION

> Programs should be written for people to read,
> and only incidentally for machines to execute.[4]

Historically, computer music and tape music are closely related. This is not only true for tape music created with the help of computers, but also for many so-called real-time computer music performances, at least from the audience's point of view. Often, there is not much difference between a laptop-performer who starts playback of a multi-track recording and then awaits the end of a piece, and a laptop-performer who starts dsp-engines, modifies effects and parameterises algorithms controlling the former in real-time.

While it is often not so interesting for the audience to watch pale faces illuminated by computer screens, it is at the same times often not very interesting for the performers to play their (however complex) systems and algorithms with the limited interface of keyboard, mouse and fader-boxes. [1]

The performers are almost exclusively the same people who have designed and written the software instruments in countless hours. The traditional separation into composer, instrumentalist and instrument maker is not valid for them anymore. And since these people spend most of their time at the design of their instruments (which, due to the power of general purpose machines, are not "just" instruments but can also hold scores and algorithms, which will eventually form the "composition"), it is only logical that this is the field where they gain the greatest skill and virtuosity: the design of algorithms and their implementation in source code.

It was only a matter of time until these people started to utilise their specific skills and explore them in live performances, in club concerts and at experimental music festivals.

Writing code before the audience ideally gives the performers an intellectual challenging way to play and improvise with their instrument. From the audience's point of view, the performers are also physically involved into the making of music, at least they are struggling with the computer in a perceivable way. In order to perceive not only the fact *that* the performers are doing something but also *what* they are doing, it helps to give the audience some secondary information about the code, for instance by projecting it onto a video wall.

2. A SHORT HISTORY

Emerging from the club scene at the end of the 1990s, a number of prominent Live Coding performers soon organised themselves into an international consortium "for the proliferation of live audio programming": *TOPLAP* [6, 2].

However, traces of Live Coding can be found far earlier.

2.1. The Early Days

Being an art form heavily depending on general purpose computers, not much Live Coding can be found before the 1950s. Some Authors [2] argue that the tournament on cubic equations between the two Italian mathematicians Nicolo Fontana Tartaglia and Antonio Maria Fior about 1539 might be considered an early Live Coding performance (albeit it lasted for several weeks and is thus not directly comparable to today's short-lived performances).

With the advent of the microcomputer in the 1970s, computers finally became small enough to fit on stage, where they could be used as performance instrument. One of the first known Live Coding sessions is attributed to Ron Kuivila, performed at STEIM, Amsterdam, in 1985[5].

"The Hub", one of the early experimental network computer bands in the 1980s [7], allowed the audience to per-

[1] This does not deny all the efforts being undertaken in the field of New Interfaces for Musical Expression. And doubtless there are performers who are happy with limited interfaces that make it quite impossible to break the entire setup during performance.

ceive their decision making by giving them read access to their monitors:

> "The Hub's composers [...] allowed us to walk around, observe their computer screen messages, and assuage our curiosity."[12]

The languages of choice at that time were *Lisp* dialects (the Hub) and *Forth* (the Hub and Kuivila).

2.2. The Dark Ages

In the 1990s, the energies of the computer pioneers to do Live Coding seem to have ebbed away. The hype about the Internet and (related to it) Open Source/Free Software seems to have bound most of the creative potential within the hands of software experts. However, both the FLOSS movement and the Internet have contributed a lot to the development and the public perception of software art[29].

2.3. A New Dawn

At the end of the 1990s, software art (and its sub-category code art) [10] evolved from *net.art*. Exploring the beauty of algorithms, this movement dealt with code as a new form of expression[13].

With the ever-growing power of computers, interpreted languages (or rather: their interpreters) had by now become fast enough to be used to generate audio and even video in real-time.

In 2000, the duo *SLUB* (Alex McLean and Adrian Ward) did their first Live Coding performance (including projection of source code), utilising a self-written environment hacked together in languages like Perl and REALbasic.

At about the same time, Julian Rohrhuber did first experiments with Live Coding in SuperCollider.

Since then an increasing number of people have expressed themselves by the means of source code on stage, which eventually lead to the founding of an organisation dedicated to Live Coding, *TOPLAP*[2].

While at the beginning Live Coding was confined to the *ab*use of general purpose languages (Perl) and the extension of existing computer music frameworks (SuperCollider), soon the development of integrated environments dedicated solely to the Live Coding of sound and multimedia (ChucK, impromptu; see Section 5) started.

3. PARALLEL EVOLUTIONS

3.1. Hacking hardware

While Software-Code has become more and more publicly available and perceptible within the last 30 years, a similar development has taken place within the field of hardware: electronic devices have started to be designed as become Open Hardware instead of black-boxes[26].

Within an artistic context a development comparable to Live Coding can also be spotted: pioneers like Tetsuo Kogawa have done soldering performances, for example explaining how to build a mini-FM transmitter[16].

Although many Live Coding artists claim that one of the core concepts of Live Coding is the making of decisions about algorithms in real-time, one aspect is certainly to allow the audience to see how things are done that are usually concealed. Naturally, this applies to the live assembly of electronic parts as well: in traditional music performance, the instrument (whether it is hard- or software based) is build a long time before the performance: the performance itself then consists of parameterising (aka "playing") the instrument.

The two counterparts of creating hardware and software in real-time go along well, as is shown in combined performances such as the "Live Coding versus live circuit building performance/duel" by Nick Collins and Nicolas Collins at NIME-07[3].

3.2. eXtreme Programming

Another parallel development can be observed in the computer sciences: agile programming techniques such as *eXtreme Programming* introduced the principle of *Pair Programming*. *Pair Programming* basically involves two programmers who work together at one workstation, writing software. One of the programmers does the typing, while the partner is watching the code evolve, commenting on it and hopefully finding problems as soon as they come into existence [18].

While his technique is not directly related to artistic performance, it provides similarities; for instance in both Live Coding and Pair Programming decisions about algorithms (in general) and code (in particular) to be used are made in the presence of an audience. While the performer will eventually react on the audience's behaviour via an indirect feedback, the co-programmer can directly influence the evolving code.

4. PARADIGMS AND AESTHETICS

In the Lübeck04 manifesto[28], the members of TOPLAP state what they believe is important for Live Coding. The two main issues (apart from the obvious fact that software has to be written in real-time in order to be called "Live Coding") are:

- Live Coding is about algorithms rather than tools
- "Obscurantism is dangerous. Show us your screens."

Adhering to the "Show us your screens" demand, in a usual Live Coding performance, the source code is projected onto a video wall, a practice that somewhat limits the "open aesthetics"[33]. Ensembles like *Powerbooks Unplugged*[1] try to overcome this by letting the performers take place within the auditory and presenting the code not via big video projections but in the privacy of their laptop screens[11].

At first glance, Live Coding seems to celebrate the performer as a virtuoso, who is in total control of algorithms, source code and the keyboard.

In contradiction to this, several prominent live-coders, are following (anti-)postmodernist aesthetics, trying to overcome the traditional ideas of genius: While Amy Alexander idealises a "goofy" anti-aesthetics[5] as often found with "geeks" within the current software culture[17], Tom Hall and Julian Rohrhuber take a more serious approach by proposing a *slow code* movement, which tries to eliminate the virtuosity of speed typing from Live Coding performances: "The slow code movement is to music what the slow food movement is to cooking."[15]

5. ENVIRONMENTS

In theory, every programming environment with the ability to produce sound and a reasonably fast implementation-execution cycle can be used for Live Coding.

While theoretically it is possible to use compiled languages for Live Coding, in practice the slow edit-compile-run cycle allows too little direct interaction for most improvisers.

Since Live Coding is still at its infancy, there are not many Live Coding systems available yet. Therefore Live Coders either have to write their own Live Coding environment from scratch, or extend existing real-time systems.

Alex McLean and Adrian Ward are well known for using *REALbasic*[8], *Perl*[21] and the Unix command line interpreter *bash*[20] as a Live Coding environment.

Compared to multi-purpose programming languages like *Perl*, (real-time) computer music languages ease the task of creating sound a lot.

5.1. SuperCollider & JITLib

Most likely, the first environment based on a computer-music system and dedicated to Live Coding (or *Just-In-Time Programming* as it is called here) has been the *Parcel* extension to SuperCollider[19] by Julian Rohrhuber, which was later developed further into *JITLib* [24].

JITLib provides a proxy-system for diverse processes (like synthesis) to be added, modified and deleted at will, while providing a unified way to switch between various processes by means of interpolation[25].

Recent additions to this library also allow several performers connected via a physical network to share and collaboratively manipulate these processes[11].

5.2. ChucK

ChucK[30] is probably the first computer-music language dedicated to and designed for Live Coding (or *On-the-fly Programming* as the authors refer to it). *ChucK* provides language constructs to control, modify and replace *shreds* (tightly synched processes running in parallel at different speeds) programmatically[33]. In addition to these formal constructs, *ChucK* also provides an integrated environment called *Audicle* designed to handle them efficiently and to visualise the structure and system-interaction of the resulting software apart from simply showing the source-code[31], an important aspect for an audience that is not necessarily code literate.

5.3. Environments for Multimedia

While *ChucK* provides several graphical representations of the live-coded software, it is not meant for creating visual output (yet[32]).

In the meantime, several other Live Coding environments have been developed with a focus on graphics, video and multimedia. Since many of these environments are targeted at VJs producing "visuals", they have inherent multimedia capabilities like basic analysis of incoming sound, in order to tightly couple audio and video.

Other (more direct) multimedia approaches include the exchange of control data between various system-nodes dedicated to different media via a higher level protocol, or the creation of several stimuli from within a true multimedia environment[9].

One common problem of these environments is that both the primary artistic output (the images) and the secondary one (the code that creates them) are visual impressions and thus overlap in the presentation (at least, if it is important that the code is shown to the audience).

One solution to this is to present code and imagery on different screens, eventually with different sizes in order to focus the audience's attention on one of the two representations. Another solution is to integrate the code into the imagery, at the cost of making the source code less readable.

The Thingee and its underlying language *ThingeeLanguage* are based on Macromedia's Director and its scripting language *Lingo*. Contrary to most Live Coding environments where the focus is on the expressivity of the language, *The Thingee* aims at the (eventually software-illiterate) audience that wants to understand what the programmer does and how this translates into an artistic outcome [5].

A more traditional approach (with a focus on the language) is represented by *fluxus*, a *scheme/Lisp* based 3d rendering engine with a special editor for Live Coding[14].

Another *scheme* based environment that is dedicated to both audio and video creation is *impromptu*. Unlike most other environments described here, *impromptu* also provides ways for collaborative Live Coding, where several programmers interact on the code level[27].

5.4. Graphical Environments

Graphical computer music languages, such as Max/MSP or Pure data[23] have the advantage of offering a representation of the source code that is easily accesible by the audience. While it is arguable that graphical programs can be quite complicated to read and fully understand[22], they offer a certain familiarity and metaphors for people who would probably be unwilling to read text-based source code.

Many of these systems provide mature multimedia extensions (Jitter for Max/MSP; GEM, pdp and GridFlow for Pure data) for integrated Live Coding of both audio and video but do not offer any specific constructs for switching between discontinuities of processes.

6. CONCLUSION

Live Coding has established itself as an alternative to traditional laptop performances. Offering a form of improvisation at an algorithmic level, it gives an insight into the used algorithms to the audience by making the source code visible, while at the same time focusing on the (physical) presence of the performers.

Currently a lot of Live Coding performances take place either informally in clubs or more formally in experimental festivals, like the *LOSS Livecode Festival* or the *Linux Audio Conference*. The main focus is still on the joyful exploration of this new technique.

Once this performance practice has grown out of its infancy, it might well be that one day live-coding "software musicians" will be part of traditional ensembles.

7. REFERENCES

[1] Powerbooks unplugged. `http://pbup.goto10.org/`, 2003-.

[2] Toplap homepage. `http://toplap.org`, 2004.

[3] Toplap mailing list archives, 2004-2007.

[4] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 2 edition, 1996.

[5] W. Adrian, R. Julian, O. Fredrik, M. Alex, G. Dave, C. Nick, and A. Alexander. *Live Algorithm Programming and a Temporary Organisation for its Promotion*, pages 243–261. README, 2004.

[6] R. Andrews. Real djs code live. *Wired: Technology News*, 2006.

[7] C. Brown and J. Bischoff. Indigenous to the net: Early network music bands in the san francisco bay area. `http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html`, 2002.

[8] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding techniques for laptop performance. *Organised Sound*, 8(3):321–330, 2003.

[9] N. Collins and F. Olofsson. klipp av: Live algorithmic splicing and audiovisual event capture. *Computer Music Journal*, 30(2):8–18, 2006.

[10] F. Cramer. *Zehn Thesen zur Softwarekunst*, chapter 1, pages 6–13. Künstlerhaus Bethanien, Berlin, 2003.

[11] A. de Campo, A. Vacca, H. Hölzl, E. Ho, J. Rohrhuber, and R. Wieser. Code as performance interface - a case study. In *Proc. of NIME*, New York, to be published.

[12] K. Gann. The hub musica telephonica. *The Village Voice*, (6-23-87), 1987.

[13] G. Gohlke, editor. *Software Art - Eine Reportage über den Code*. Künstlerhaus Bethanien, Berlin, 2003.

[14] D. Griffiths. Live coding of graphics. `http://www.toplap.org/index.php/Live_coding_of_graphics`, 2004.

[15] T. Hall and J. Rohrhuber. Slow code. `http://www.ludions.com/slowcode/`.

[16] T. Kogawa. Tetsuo kogawa cooks a fm transmitter. `http://anarchy.translocal.jp/streaming/19911104tkcookstx.ram`, 1991.

[17] L. Konzack. Geek culture: The 3rd counter-culture. In *Proc. of FNG2006*, Preston, England, 2006.

[18] A. MacKenzie and S. Monk. From cards to code: How extreme programming re-embodies programming as a collective practice. *Journal Computer Supported Cooperative Work (CSCW)*, 13(1):91–117, 2006.

[19] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[20] A. McLean. Angry - /usr/bin/bash as a performance tool. In S. Albert, editor, *Cream*, volume 12. Twenteenth Century, 2003.

[21] A. McLean. Hacking perl in nightclubs. `http://www.perl.com/pub/a/2004/08/31/livecode.html`, 2004.

[22] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[23] M. S. Puckette. Pure data. In *Proceedings of the International Computer Music Conference*, pages 224–227. International Computer Music Association, 1997.

[24] J. Rohrhuber and A. de Campo. Uncertainty and waiting in computer music networks. In *Proceedings of the International Computer Music Conference*, 2004.

[25] J. Rohrhuber, A. de Campo, and R. Wieser. Algorithms today - notes on language design for just in time programming. In *Proceedings of the International Computer Music Conference*, Barcelona, 2005.

[26] G. Seaman. Free hardware design - past, present, future. In *Oekonux Conference*, 2001.

[27] A. Sorensen. Impromptu: an interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference*, pages 149–153, 2005.

[28] toplap. Toplap manifesto. `http://toplap.org/index.php/ManifestoDraft`, 2004.

[29] G. Trogemann and J. Viehoff. *CodeArt - Eine elementare Einführung in die Programmierung als künstlerische Praxis*. Springer, Wien, 2005.

[30] G. Wang and P. R. Cook. Chuck: a concurrent, on-the-fly audio programming language. In *Proceedings of the International Computer Music Conference*, 2003.

[31] G. Wang and P. R. Cook. The audicle: a context-sensitive, on-the-fly audio programming environ/mentality. In *Proceedings of the International Computer Music Conference*, 2004.

[32] G. Wang and P. R. Cook. Chuck: A programming language for on-the-fly, real-time audio synthesis and multimedia. In *ACM Multimedia*, 2004.

[33] G. Wang and P. R. Cook. On-the-fly programming: using code as an expressive musical instrument. In *New Interfaces for Musical Expression (NIME)*, Hamamatsu, Japan, 2004.