LIVE ALGORITHM

PROGRAMMING

AND A TEMPORARY

ORGANISATION

FOR ITS

PRO

MOT

IO

N

ADRIAN WARD, JULIAN ROHRHUBER,
FREDRIK OLOFSSON, ALEX MCLEAN,
DAVE GRIFFITHS, NICK COLLINS,
AMY ALEXANDER

Why, his hands move so fluidly
that they almost make music.
(Ba.Ch creates a chess program
for Crab, Hofstader, 2000, p.729)

We can do better than almost.

Live computer music and visual performance can now involve interactive control of algorithmic processes. In normal practise, the interface for such activity is determined before the concert. In a new discipline of live coding or on-the-fly programming the control structures of the algorithms themselves are malleable at run-time. Such algorithmic fine detail is most naturally explored through a textual interpreted programming language. A number of practitioners are already involved with these possibilities using a variety of platforms and languages, in a context of both public exhibition and private exploration.

An international organisation, TOPLAP, has recently been established to promote the diffusion of these issues and provide a seal of quality for the audiogrammers, viders, progisicians and codeductors exploring the art of live coding. Practitioners provide descriptions of their work in later sections of the paper. For the act of performance, a manifesto, Lubeck 04, presented herein, attempts to qualify the expertise required in live programming for an acceptable contribution to this new field.

Introduction

Live coding is the activity of writing (parts of) a program while it runs. It thus deeply connects algorithmic causality with the per-

ceived outcome and by deconstructing the idea of the temporal dichotomy of tool and product it allows code to be brought into play as an artistic process. The nature of such running generative algorithms is that they are modified in real-time; as fast as possible compilation and execution assists the immediacy of application of this control. Whilst one might alter the data set, it is the modification of the instructions and control flow of processes themselves that contributes the most exciting action of the medium.

Live coding is increasingly demonstrated in the act of programming under real-time constraints, for an audience, as an artistic entertainment. Any software art may be programmed, but the program should have demonstrable consequences within the performance venue. A performance that makes successive stages of the code construction observable, for example, through the use of an interpreted programming language, may be the preferred dramatic option. Text interfaces and compiler quirks are part of the fun, but a level of virtuosity in dealing with the problems of code itself, as well as in the depth of algorithmic beauty, may provide a more connoisseurial angle on the display.

The Satanic advocate might ask whether a live coder must understand the consequences of an algorithm change, or whether their curiosity to make a change is sufficient? A codician might tamper with Neural Net weights or explore emergent properties. Many algorithms are analytically intractable; many non-linear dynamic processes twist too swiftly. A continuum of the profundity of understanding in such actions is accepted. Perhaps the prototype live coders are improvising mathematicians, changing their proofs in a public lecture after a sudden revelation, or working privately at the speed of thought, with guided trail and error, through possible intuitive routes into a thorny problem.

There are always limits to the abstract representations a human mind can track and which outcomes predict. In some cases trial and error becomes necessary and cause and effect are too convoluted to follow. As a thought experiment, imagine changing on-the-fly a meta-program that was acting as an automated algorithm changer.

This is really just a change to many things at once. Iterate that process, piling programs acting on programs. Psychological research would suggest that over eight levels of abstraction are past the ability of humans to track.

Live coding allows the exploration of abstract algorithm spaces as an intellectual improvisation. As an intellectual activity it may be collaborative. Coding and theorising may be a social act. If there is an audience, revealing, provoking and challenging them with the bare bone mathematics can hopefully make them follow along or even take part in the expedition.

These issues are in some ways independent of the computer, when it is the appreciation and exploration of algorithm that matters. Another thought experiment can be envisaged in which a live coding DJ writes down an instruction list for their set (performed with iTunes, but real decks would do equally well). They proceed to HDJ according to this instruction set, but halfway through they modify the list. The list is on an overhead projector so the audience can follow the decision making and try to get better access to the composer's thought process.

A Motivation: Coding as a Musical Act

There are many comparisons to be made between software and music. For example, both exist as a set of instructions to be interpreted and executed to produce a temporal form. I play this music I've scored, I run this software I've hacked together; I breathe life into my work.

Indeed, some musicians explore their ideas as software processes, often to the point that a software becomes the essence of the music. At this point, the musicians may also be thought of as programmers exploring their code manifested as sound. This does not reduce their primary role as a musician, but complements it, with unique perspective on the composition of their music.

Terms such as "generative music" and "processor music" have been invented and appropriated to describe this new perspective on composition. Much is made of the alleged properties of so called

"generative music" that separate the composer from the resulting work. Brian Eno likens making generative music to sowing seeds that are left to grow, and suggests we give up control to our processes, leaving them to "play in the wind."

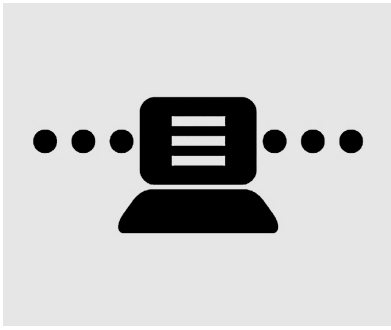
This is only one approach to combining software with music, one that this paper wishes to counter quite strongly. We advocate the humanisation of generative music, where code isn't left alone to meander through its self-describing soundscape, but is hacked together, chopped up and moulded to make live music.

And what works for generative music should extend to further time based algorithmic arts.

An Organisation Supporting Live Coding

The organisation TOPLAP (www.toplap.org), whose acronym has a number of interpretations, one being the Temporary Organisation for the Proliferation for Live Algorithm Programming, has been set up to promote and explore live coding. TOPLAP was born in a smoky Hamburg bar at 1am on Sunday 15th February 2004. A mailing list and online community has since grown up, this paper being one outcome of that effort.

TOPLAP does not set out to judge live coding, in that much will not take place as a public display. But where the world of performance intrudes, we can encourage some healthy principles of openness and quality to foster the explorations of this new field. Profundity of insight into code and representation is to be applauded. The



Lubeck04 manifesto below, begun on a Ryanair transit bus from Hamburg to Lubeck airport, tackles the performance side of live coding.

The Lubeck04 Manifesto for Live Coding Performance

The act of programming on stage is controversial, especially since it typically involves the much derided laptop. Both Roger Dean (Dean 2003) and Andrew Schloss (Schloss 2003) are set against aspects of laptop performance. Whilst Dean finds the notion of projecting a laptop screen somehow a cheapening of music (I expect he holds his eyes closed in every concert so as not so see conductors, violins, or, ugh, computer performers) Schloss cautions against the lack of causality and gesture in laptop performance. He makes no reference to McLean's notion of laptop projection as essential to confront the former (McLean 2003, Collins 2003); the latter is an acknowledged physical restriction of the computer keyboard interface, but sidestepped by the intellectual gestures of live coding. The seventh point in his concluding list is damning:

"People who perform should be performers. A computer music concert is not an excuse/opportunity for a computer programmer to finally be on stage. Does his/her presence enhance the performance or hinder it?" (Schloss 2003, p242)

The only hope is that he be outflanked by the position herein; programming becomes performance. We now qualify what constitutes a good performance.

We the digitally oversigned demand:

- Give us access to the performer's mind, to the whole human instrument.
- Obscurantism is dangerous. Show us your screens.
- Programs are instruments that can change themselves
- The program is to be transcended—Language is the way.
- Code should be seen as well as heard, underlying algorithms viewed as well as their visual outcome.

- Live coding is not about tools. Algorithms are thoughts. Chainsaws are tools. That's why algorithms are sometimes harder to notice than chainsaws.

We recognise continuums of interaction and profundity, but prefer:

- Insight into algorithms
- The skillful extemporisation of algorithm as an impressive display of mental dexterity
- No backup (minidisc, DVD, safety net computer)

We acknowledge that:

- It is not necessary for a lay audience to understand the code to appreciate it, much as it is not necessary to know how to play guitar in order to appreciate watching a guitar performance.
- Live coding may be accompanied by an impressive display of manual dexterity and the glorification of the typing interface.
- Performance involves continuums of interaction, covering perhaps the scope of controls with respect to the parameter space of the artwork, or gestural content, particularly directness of expressive detail. Whilst the traditional haptic rate timing deviations of expressivity in instrumental music are not approximated in code, why repeat the past? No doubt the writing of code and expression of thought will develop its own nuances and customs.

Performances and events closely meeting these manifesto conditions may apply for TOPLAP approval and seal.

A Historical Perspective

TOPLAP artists are not the first to explore the world of programming as a live activity. The earliest traced performances in computer music have been attributed to the Hub (see Collins, McLean, Rohrhuber and Ward 2003, Brown and Bischoff 2002), who used interpreted Forth as an interface to sound processes, passing parameters amongst the ensemble in further pioneering network music activity. Programming activity was essential to the Hub's workflow, and did take place during live performance, where the audience were often invited to wander around the computers.

"Constructing and coding were the way we practiced, and were "the chops" that were required to make the music happen." (Brown and Bischoff 2002 Section 2.4 Hub Aesthetics)

"Chats kept track of the progress of the band through this shape, and were often used to describe the character of the music that resulted, providing a running commentary on how the performance was going. During the New York performance the audience was free to wander around the band, observing the band's evaluation of its own performance." (Brown and Bischoff 2002 Section 3.2 Text Based Pieces)

The explicit role of programming under the scrutiny of audience as an insight into the performer's mind was not a primary outcome, however.

Recently brought to our attention by Curtis Roads was a 1985 performance by Ron Kuivila at STEIM in Amsterdam, described in (Roads 1986).

"I saw Ron Kuivila's Forth software crash and burn onstage in Amsterdam in 1985, but not before making some quite interesting music. The performance consisted of typing." (Curtis Roads, personal communication)

Kuivila also used live interpreted FORTH, on an Apple II computer, without projection.

TOPLAP Field Reports

We note some recent designs and performances that explore live coding.

Prominent live coding systems have been described in previous papers. ChucK (Wang and Cook 2003) is a new audio programming language which is intended for live use. A prototype performance took place in New York in November 2003, with double projection of a duet of live coders (Ge and Perry), and is described in (Wang and Cook 2004). Two highly developed live coding systems are revealed in (Collins, McLean, Rohrhuber and Ward 2003). The slub software of Alex McLean and Ade Ward produces music from command line antics with Perl scripts and REALbasic live

coding; slub have performed in public since the start of the millenium. Julian Rohrhuber's JITLib is an extension set for the SuperCollider (McCartney 2002) audio programming language, dedicated to live coding prototyping and performance. Julian's own work often uses it in network music, exploratory film soundtracks and improvisation where he can react to the setting and audience and describe events through audio code. Julian was instrumental in bringing together many live coding practitioners for the 'changing grammars' conference in Hamburg, Feb 12–14 2004, which promoted the genesis of TOPLAP. There are a number of other SuperCollider users working with live coding methods due to the easy applicability of this interpreted language to live coding of audio algorithms. Fabrice Mogini, Alberto de Campo and Nick Collins (Collins 2003) have all given live coding performances in the last few years.

Further systems under development under the approval of TOPLAP are now described; whilst early systems were pure audio, healthy perspectives are also arising in the visual modality and mixed audiovisual worlds. These descriptions are sometimes edited highlights from the TOPLAP mailing list; we enjoy the healthy excitement of such postings.

Alex McLean: feedback.pl

A painter, Paul Klee, moves to make a mark on a canvas. Immediately after this initial moment the first counter motion occurs; that of receptivity, as the painter sees what he has painted. Klee therefore controls whether what he has produced so far is good. This is the artistic process described by Klee in his excellent Pedagogical Sketchbook [Klee, 1968]. The same process occurs when an artist edits a computer program. The artist types an algorithm into the computer, the artist considers it in its place and moves to make the next mark in response.

However, live programming allows us to play with this relationship somewhat. Let me describe how my live programming environment, 'feedback.pl' works.

The marks that the artist is making in feedback.pl don't just affect the artist, but also affect the running code. While the artist considers the possibilities of a fresh algorithm that they have just typed, feedback.pl is already executing it.

What's more, the algorithm running in feedback.pl can edit its source code. I should explain carefully... While the artist types in a computer program, feedback.pl is running it; that running process may edit its own source code. Those self-modifications take immediate effect, changing the running program. And so it goes on.

The running code may also do other things. I use feedback.pl as an environment to make music. So I write live code in feedback.pl that makes live music.

And so we have at least three complementary feedback loops. One loop is that of the artist and their work in progress—expressions enter the program, the artist observes the effect on the sourcecode and reacts with further expressions. A second loop is that between the sourcecode and the running process—the process reads the sourcecode and carries out instructions which modify the sourcecode, then reads in its modifications and follows those.

The third, outer feedback loop appears when the program creates external output. The program might generate some sounds that the artist interprets as music, reacts to by changing the sourcecode, immediately changes the process which in turn alters the generated sound, which the artist reacts to once more.

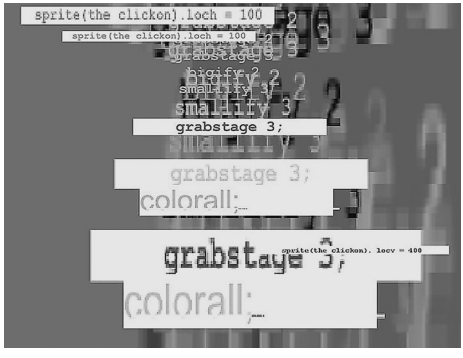
When I'm performing before a live audience, then there is a fourth feedback loop, encompassing the audience who are interpreting the music in their own individual ways, making their own artistic expressions by dancing to it, which affects the choices I make, in turn changing the music.

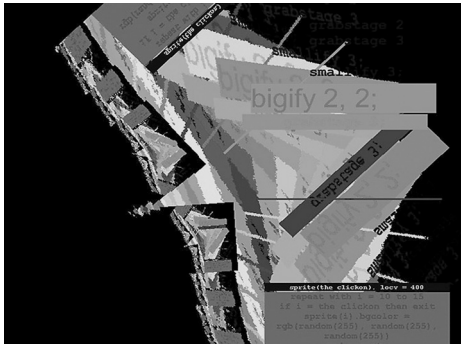
If the programmer is to share the continuous artistic process of a painter, their performative movements must be must be reactive, they must be live. Code must be interpreted by the computer as soon as it can be written, and the results reacted to as soon as they can be experienced. The feedback loops must be complete.

Amy Alexander: The Thingee

I threw together a rough quick funny Thingee that's a livecode editor and VJ instrument—both at the same time. You type the code on the screen, and it actually executes the code as you type it, but all the visuals are made of the onscreen code too, though I'm experimenting with including the ability to pull in the whole desktop sometimes as well. It generously abuses the concept of feedback; in this case, the whole thing's a feedback loop anyway. This is somehow reassuring because the concept of video feedback in digital VJ tools always disturbs me a little—since computers have no optics they can't "see" their own output as did the original video feedback with cameras. I lie awake nights worrying about that.

Rough preliminary screengrabs:





Well it may look a bit pretty in these screenshots, but it usually seems quite a bit goofier when you see it move. And of course, since you're seeing it respond to the code as I type it in, it has that live-code appeal that you can't see in the stills. I think I've got things worked out now so it doesn't crash when I type illegal code in. However, there's still plenty of room for me to make a fool of myself in performance, as I occasionally type things by accident that make the code-boxes go whizzing around out of my control and then I have to try to "catch" one to make it stop. (I am working on a "panic button" implementation—but of course it's much more entertaining if I at least *try* to catch them... :-))

Some more details: there are currently six code-boxes (editors) on the screen, plus two buffers which can take images of the app or the whole screen for feedback purposes. You can type a one line command or a many line snippet in each box, assuming enough space—the box will expand, up to a point. Of course, you could type some livecode to make the font smaller, and that would give you more room. The idle loop will execute the code in any or all of the code-boxes that end with a semicolon at a given moment; the code takes effect as soon as you type the semi-colon. Thinking of performative speech and performative code—it's kind of neat (I think) to see the *semicolon* actually perform what it does! (and also will be a funny inside joke to people who know that Lingo doesn't use semicolons like all those "difficult" languages. :-)).

As to the question of whether all code in a livecode performance should be written in "longhand" (the original programming language) or whether it's OK to write custom shortcuts — I'm working it out as I go along. :-) Sometimes it makes sense to type it all out, but for certain stuff, it's just too much in live performance—so shortcuts become essential. The discerning screengrab viewer may note the beginnings of a custom shortcut grammar beginning, based on my personal tendency toward verbification: therefore, the following commands exist in ThingeeLanguage: smallify, rotify, skewify, colorify, bigify and biggify (because I cannot

seem to spell that one consistently). There are some other commands too that don't match that convention—I'll probably go back and "ify" them too.

In making the shortcuts, of course I'm essentially writing a "language" one level higher/more abstract than the one I was writing in (Lingo.) Which is itself written on top of something else, which is written on top of something else, eventually going down to the 1's and 0's of machine language and then to the processor. If I kept going to successively higher level "languages"—I could eventually get down to pre-scripted one-keypress actions, which is more traditional user interaction for performative software. So this illustrates—for me anyway—that the line between programmer and user is really a continuum, not a line in the sand. The user is in some ways programming, but the programmer is also very much a user, influenced in what she does by the language in which she writes. Control is a sliding scale, which can sometimes slide in unexpected directions.

Of course the devil is in the details and I should really find some time to make The Thingee a bit less rough than it is, and actually get good enough at performing it to make sure it's entertaining to watch and not just a concept piece. Entertainifying with software is very important to me.

In keeping with my new "foot-operated software" crusade, I may also add an option where you can do the live coding with your feet on a DDR [Dance Dance Revolution] pad. (This would have the added benefit of bringing the world one step closer to the elusive goal of dancing about architecture.) I think I've actually figured out a warped hack for doing that, but I'd really have to be much less tired than I am now before I could attempt such a feet <sic>.

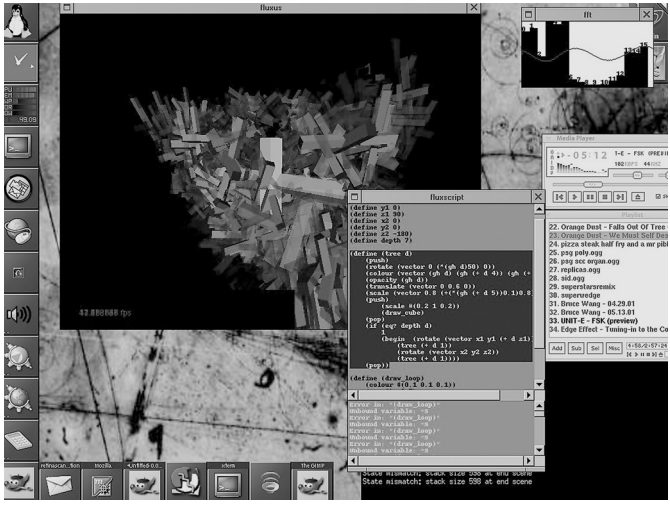
Dave Griffiths: fluxus

Fluxus is an OpenGL based render engine which reads audio input and makes realtime animations. The "scenes" you build are entirely described by scripts that are written in realtime.

The actual script interface is in a separate window to the renderer currently. It resembles the maya mel script editor, in that you write code and highlight it to execute it in fragments. That way you can build up a palette of function calls and bits of code and execute them with the mouse and a keypress. The whole time you're doing this, the renderer is doing its thing, so it's usable live.

If no code is selected, hitting F5 will execute the whole script. Also you can set a bit of code that's called each frame—if you make this a function call, you can modify the function, highlight and execute the scheme function definition again, and it will update instantly. There is also a simple GUI that just consists of buttons that execute code snippets (again, usually function calls, but it can be whatever you want). You can build this simple GUI live, or load it from prebuilt scripts.

In practice, script errors don't really break the flow of the visuals. No crashes. Well, that's the idea anyway—I keep everything really simple, there have been a few issues with the physics library (which is superb in other respects: http://www.ode.org/) but I've got them mostly under control now.



Peter Halley

The lower area of the scripting window spews script errors out. With scheme it's mostly mismatching ((((())))) so I added parenthesis highlighting in the editor which makes it miles easier. If there is a syntax error in a script the interpreter aborts the code there, and it's quite easy to sort out. The only thing that seems to go wrong is if the code generates a mismatching (push) or (pop) trashing the renderer's state stack—but this often results in visually interesting ;) results anyway...

Fluxus is very much designed for live use (and live coding)—and I would love to do so at some point.

One day I'd like to make the code editor a quake style console thingy, so the code is visible to all, on top of the 3D. Figuring out the interface for this is a bit more tricky, but I guess it could work in a similar fashion.

Project homepage: http://www.pawfal.org/Software/fluxus/

Fredrik Olofsson: redFrik

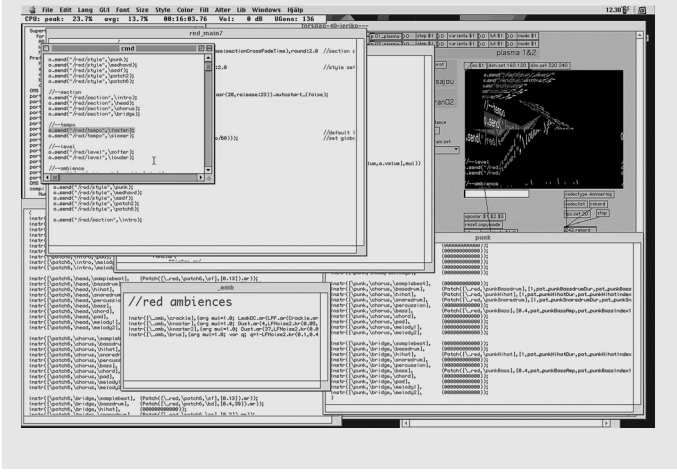
I did a disastrous gig exactly a year ago in Malmö in the south of Sweden using my now defunct SuperCollider2 redFrik livecoding framework. It was not true livecoding but more of a syntax interface for routing sounds through effects, changing tempo, filling patterns, starting and stopping processes with different fading times and so on. I thought typing live both looked cool and could engage the audience in the process. Adding to this I had, running simultaneously in the background, a Max/Nato.0+55 videopatch that grabbed a small area of the screen. So whatever syntax was visible in this area got passed through two or three video effects, scaled up to fullscreen and sent to a beamer. These trashing and pixelating effects were in turn controlled by parameters in the music like tempo and kickdrum pattern. Hereby the code I typed was visually transformed and rearranged in sync with the sound.

It was a big event and I screwed up totally. I did not rehearse properly due to my redesign of the framework up until the last minute. And during the set my brain went into chimpanzee mode as so

often when performing. I could not keep all things in my head controlling video and music simultaneously so both stagnated for long periods of time while I tried to think clearly. Add to that that the projector was horrible and my video effects possibly a little too brutal so the mess was totally unreadable. Me still shivers when thinking about it. So I figured I needed to start rehearsing livecoding on a daily basis, as any instrument, and also fully automate certain things (like the video part) and that is where I am still at.

I only used this system for two concerts. The main reason was that it was notoriously dangerous to do live-coding in SuperCollider2. Slightest typo and the sound would choke and you were forced to reload your system. As that would take a few long and embarrassingly silent seconds, you tended to play safe and not change the code a lot. It promoted copy&paste of known bug free syntax.

My framework did have some nice features though. One of them was that commands that for example filled a pattern with 60% random hits, could be undone with the revert n steps command.



The black videoscreen in the right upper corner is the projected output and corresponds to what is visible in the left hand 'cmd' window

o.send("/punkBassdrum/patAmp", 16, 0.6);//fill with new random values
o.send("/punkBassdrum/patAmp", \revert, 1); //go back to the previous if a poor sound

or a similar example filling a 16 slot pattern with random pitches between 40 and 80 Hertz:

o.send("/punkBassdrum/patPitch", 16, 40, 80);

With revert I could jump back to good sounding patterns. I just needed to remember which indexes were good sounding -sigh.

o.send("/punkBassdrum/patPitch", \revert, 1);

There's no cure for the Halting problem but having safe positions, in this case good sounding settings, to retreat back to as you branch out I think is vital. Like a mountain climber at regular intervals drilling for firm stops. And these presets couldn't be prepared in advance—they're made up as you go.

My current setup builds around Julian Rohrhuber's JITLib for SuperCollider Server and is more a set of tools helping me to quickly find snippets of code and sound definitions—all built to support live coding in JITLib. I am still not confident enough to play with this in public; I feel I need more practice.

Julian Rohrhuber, Volko Kamensky:
Interactive Programming for Real Sound Synthesis

The film scenes in "Alles was wir haben" (8-mm, 22 min) are a chain of slow 360¡ moves filmed in a small town of northern Germany, a town, so we hear the representative of the Homeland Museum say, that has a long history, a long history of fires that have been destroying the place over and over again. The museum, according to him, is a place dedicated to the identity of the historical roots, without which the inhabitants would have no true existence. The Homeland Museum was rebuilt several times after bring destroyed by fire. The problem the film maker Volko Kamensky had was that atmospheric sound recordings create the impression of an outward world that is as it is, not made, not constructed—the film as a documentation of a factual town.

As he wanted to create the impression of a constructed, isolated, artificial reality, just as constructed as the idea of history, of reality in general, we had the idea to use interactive programming to synthesize synthetic sound imitations of all actions that are expected to produce sound. We used JITLib to slowly approach a sound portrait of what we imagined to be the proper sound for each scene. Without any nature study it was the talking about the sound and the possibility to write the program on the fly that allowed us to find real sound simulations, caricatures sometimes, that were not a result of an exact physical model, but, similar to a painting, of a perceptual and conversational process. The simple fact, that we did not have to stop the program to modify it, allowed us to develop an awareness of differences which were essential for this work. There were some unexpected findings which are sometimes ridiculously simple, and also the visibility of the code formed a kind of second causality that made the artificiality of our documentary construction even more perceivable. In the film, the code is not present in a visual form. Nevertheless I hope the process of its creation can be felt and read as part of the world this film projects.

This description may serve as an example of live programming that does not happen in a classic performance situation, but as part of an everyday culture of programming. Although much of this activity is never seen in public, it is an undercurrent of algorithmic music practice which I hope to have shed light on by this example.

```
//presslufthorn (feuerwehr)
(
~feuer = {
    var freq, f0, out;
    //freq = MouseX.kr(400, 500);
    //freq = 450; //altes martinshorn
    freq = 418;
    freq = LFPulse.kr(0.6, 0, 0.5).range(freq, freq * 1.36);
    out = BPF.ar(
        Mix.fill(4, {
            Pulse.ar(freq * SinOsc.kr(rrand(5.0, 7.5), 0, 0.005 * rrand(0.1, 1), 1),
            rrand(0.2, 0.5), 0.3)
        }),
        freq * 2, 0.2
    );
    Pan2.ar(out)
}
)
```

I would say that doing live programming alone, I am the audience and programmer in one, which is not a trivial unity, but a quite heterogeneous one, in which the language, my expectations, my perceptions, errors and my poetic and/or programming style play their own roles. The situation with one or more persons as audience, or, not to forget, as coperformers, is an interesting extension of this situation, but it is by no means primary. There are many specific problems of this larger interaction, such as readablility, performance style etc. which are well worth discussing, but I would not constrain interactive programming to this specific public situation.

For the film "Alles was wir haben" this was the situation that was responsible for how the whole idea worked and it was what I did while working on JITLib. I could describe the compositional work I did more in detail, but maybe this is already a general description of a type of situation which I find very interesting in itself.

In my experience it allows a very spontaneous interaction with the music in combination with talking about the music. This means I can, while playing, talk with others about what they hear, think, etc. or about other topics even and as the sound keeps playing I can react to these conversations by changing the code. Then I am not in the situation of the one who is looked at (especially as it is not my body movements which are so interesting, hmhm..) but the code / sound relation is in the center of attention. Of course, one step further, in networked live coding there is a flow of code between all participants which can, to different degrees, interact and contribute. Nobody knows who does what anyways.

JITLib: *http://swiki.hfbk-hamburg.de:8888/MusicTechnology/566*

Conclusions

Here's looking forward to the day that MC XVIII,
Terminalator X and Scope Codey Code are
household names. Big up arrow the
code massive. We know
where you
code
.

Thanks to: Curtis Roads, Ron Kuivila, John Bischoff and Alberto de Campo

Literature

Chris Brown and John Bischoff. 2002. "Indigenous to the Net: Early Network Music Bands in the San Francisco Bay Area". *http://crossfade.walkerart.org/*

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding Techniques for Laptop Performance". *Organised Sound*, 8(3):321–30.

Nick Collins. 2003. "Generative Music and Laptop Performance". *Contemporary Music Review*, 22(4):67–79.

Roger Dean. 2003. *Hyperimprovisation: Computer-Interactive Sound Improvisation*. A-R Editions Inc., Middleton, Wisconsin.

Douglas Hofstader. 2000. *Godel Escher Bach: An Eternal Golden Braid*. (20th anniv. ed.) Penguin.

James McCartney. 2002. "Rethinking the Computer Music Language: SuperCollider". *Computer Music Journal*, 26(4), 2002.

Alex McLean. 2003. "ANGRY — /usr/bin/bash as a Performance Tool." In S. Albert. (ed.) *Cream* 12. Available online from *http://twenteenthcentury.com/saul/cream12.htm*

Curtis Roads. 1986. "The Second STEIM Symposium on Interactive Composition in Live Electronic Music". *Computer Music Journal,* 10(2): 44–50.

Julian Rohrhuber and Alberto de Campo. 2004. "Uncertainty and Waiting in Computer Music Networks". ICMC 2004. (forthcoming)

Andrew Schloss. 2003. "Using Contemporary Technology in Live Performance; the Dilemma of the Performer". *Journal of New Music Research,* 32(3): 239–42

Ge Wang and Perry R. Cook. 2003. "ChucK: A Concurrent, On-the-fly Audio Programming Language". In Proceedings of the International Computer Music Conference, Singapore.

Ge Wang and Perry R. Cook. 2004. "On-the-fly Programming: Using Code as an Expressive Musical Instrument". In New Interfaces for Musical Expression (NIME), Hamamatsu, Japan.