Preprint: To appear in 2016, Journal of Music Technology and Education 9(1)

Live Coding and Teaching SuperCollider

Nick Collins

Abstract:
SuperCollider is a highly customisable programming language for music; live coding is the dynamic rewriting of a computer program as a concert or exploratory act. Learning sessions on SuperCollider can make ready use of live coding techniques, with many interesting musical possibilities, and this article explores the author's twelve years of SuperCollider teaching. Contexts include SuperCollider workshops for adults, undergraduate computer music classes, and widening participation creative coding sampler sessions for 11-13 year olds. The article is a personal reflection on pedagogy, which can provide ideas for live coding and computer music teaching across a range of audiences.

keywords:
live coding, SuperCollider, computer music workshop

Introduction

In an introductory workshop for the International Computer Music Conference (ICMC) in Berlin in 2000, James McCartney conducted an afternoon session on his SuperCollider 2 software, an immediately reacting (interpreted) audio programming language. As an enthralled participant, who had only encountered a little SuperCollider (SC) in the past, the most striking aspect of his presentation style to me was the direct use of the SuperCollider text editor. In one flow of scrolled text, McCartney provided for his audience both slides and immediate stimulants to practical exercises. Given that SuperCollider is a text-based programming language specialised for sound and music (Wilson et al. 2011), this mode of learning seemed natural once encountered. Yet a prototypical capacity was evident here for what would become called 'live coding' (amongst other names, see Collins 2011). On-the-fly alteration of the synthesis engine through code was a new feature of SuperCollider at the time only just being exploited (see Julian Rohrhuber's discussion in Collins et al. 2003).

I'd previously met SuperCollider 2 as taught by Martin Robinson on Middlesex University's Sonic Arts degree at the end of the 1990s, a brave course which held one of the few SC2 site licenses. Since 2002 I have myself run many SuperCollider workshops, both solo and in combination with co-teachers. These have ranged over multiple continents, and different audiences, from academic to general musicians, and different durations from hour long teaser sessions through week long summer schools to multi-week university modules (locations include a Singapore computer music conference, STEIM in Amsterdam and the

Sonar festival in Barcelona, Harvestworks in New York, a pay what you like summer school in London, a circuits and code combined workshop in Mexico City, live translated workshops in Japan, and many more). A recent 2013 Widening Participation initiative at the University of Sussex brought the most challenging situation yet; eleven to thirteen year olds learning SuperCollider (and some Processing) through live coding, in the context of creative coding. SuperCollider teaching with live coding has also formed a core component of my university lecturing work over the last eight years; seven years teaching combined music and computing students from a computer science department, and the last year teaching music students in a music department.

With that in mind, this article is about my personal experiences introducing musicians and programmers to computer programming for music, using SuperCollider and live coding, and unashamedly embraces the first person as needed. The research is in the main anecdotal, a practitioner's experiences in pedagogy of live coding. It is offered in the hope that other teachers and researchers will find useful approaches for the introduction of live coding and computer music in general, illustrated through case studies.

As the computer is the instrument most central to the computer musician, and computer programming its deep practice, it seems inevitable in retrospect that live coding should have established itself as a core activity. Whereas earlier centuries saw composer-pianists (Mozart, Beethoven, Liszt, Chopin, the Schumanns, Rachmaninov, Prokofiev and many more), composer-programmers (McLean, Rohrhuber, Wieser, Magnusson, Sorensen, Yee-King, Knotts, Lorway and many more) might be a contemporary equivalent. Andrew Sorenson explicitly titled a recent keynote presentation 'The Concert Programmer' and proceeded to code a musical backdrop to his talk as he delivered his speech (Sorensen 2014).

Musicians at my workshops have learnt programming sometimes by stealth and sometimes by choice; previously programming-savvy computer scientists have taken part with gusto in music making. Most people's backgrounds have been an interesting mix of technical and artistic experience. Not all participants have rushed to the sort of in-concert virtuosity exhibited by Sorensen, but all workshops have taken advantage of the ease of live coding for reflective development, exploratory adjustment, and simple edutainment. My workshops have taken place over that period where live coding itself has developed as an activity, from the early days where the few practitioners were rather unsure of where they could go, to the recent renaissance of artistic approaches, and confluence with strands of 'live programming' in computer science education.

This article sits then within the hot topic of live coding and musically oriented computer programming in education, not only evidenced by this special issue, but by a number of existing initiatives and publications (Ruthmann et al. 2010, Blackwell et al. 2014, Moore 2014, Burnard 2014). The combination of arts with science and technology education, has recently led to promotion of the STEAM (Science-Technology-Engineering-Art-Mathematics) rather than artless STEM umbrella term (see for example, Delaney 2014). That Georgia Tech has, of August 2014, just obtained a three million dollar National Science Foundation (NSF) grant in arts-led computing education to follow-up their EarSketch project (McCoid et al. 2013), after another large (around $100k) NSF grant went to

CalArts for computing education through the ChucK musical programming language in 2012, might show the stakes and timeliness of reflection on these themes.

SuperCollider workshops

The primary methodology I have followed in teaching SuperCollider, and which has become more pure as the years have gone by, is learning by (guided) making in the sense of Papert's Constructionism (compare also Hancock 2014 for the PureData graphical programming language for audio). Live coding easily enables exploratory action (Collins et al. 2003, Rohrhuber 2005). Over the years I have found myself successively reducing the gap between encountering SuperCollider and a student performing their first musical piece, so that within minutes of starting, I would like all participants to have started up the synthesizer, typed something like:

```
{SinOsc.ar(MouseX.kr(400,800))*0.1}.play
```

to instantiate a sine oscillator whose frequency is controlled by the mouse screen x position, and be coaxing their first interactive SuperCollider patches into life. For good measure, through such code the group can perform together to initiate an immediate workshop computer music ensemble! Though fixed computers in a computer room provided for teaching are often the basis, increasingly, students bring their own laptops; a spatially distributed set of computers lead to marvellous soundscapes with independently voiced gestures at play across a room. Students can be coaxed into wandering around the soundscape, starting, stopping and most importantly adapting constituent coded elements, even at each others' computers and with each others' code within the room.

Programming languages are typically easier to work with for most learners if concrete examples are provided, with students constructing their own generalisations, rather than dry and abstract outlining of technical concepts. The media computation paradigm is becoming well established as more successful than traditional and drier approaches to teaching mainstream programming (Guzdial 2011). Audiovisual outputs tend to hold greater motivation than printing text or writing to databases. It would be particularly unnatural in learning music programming if there was no musical output at the heart of the process! Teaching is led by examples, and students are encouraged to modify the examples to pursue musically interesting variations, obtaining feedback on success of variation of the underlying programming code simultaneously.

There remains a question over the depth of training possible, especially for those new to programming. Whilst simple examples are accessible enough, larger projects and more involved programming structures and principles can lead to trouble. Sometimes, students are sufficiently addicted by this point that they will overcome all manner of obstacles in the pursuit of their personalized drum machine project; for others, core principles have not adhered. This can lead to a learner who is equipped to modify basic syntax such as control parameters, but not more broadly reuse, rework, and generalise. Whilst I have spent time in workshops on technicalities of programming syntax, this still tends

to be directed to some musical problem. There can be a tension where the difficulty of more advanced programming work intervenes before the real potential of computer programming in music.

An example seems appropriate. Treatment of musical material often involves use of an array programming construct, and in SuperCollider, a list of values appears within square brackets:

```
[60,64,67]
```

These values might, as here, indicate MIDI (Musical Instrument Digital Interface) note pitches, or might indicate any number of musically relevant objects from elements of microtonal pitch systems, to rhythmic values, to timbral parameters and beyond (including nested arrays of multi-dimensional/multi-parameter sound objects and the like). The MIDI note pitch C major chord here could be a component of a generative music system or at least a sequence; but bringing in generativity or sequencing quickly requires selecting between an array of possible chords:

```
[ [ 60, 64, 67 ], [ 65, 69, 72 ], [ 67, 71, 74 ] ]
```

and at this point we are already treating nested arrays. To generalise, the SuperCollider code:

```
[0,5,7].collect{|root| root + [60,64,67]}
```

reproduces the nested array above, but has suddenly leapt up in abstraction (the new programming power implicit is at a cost of immediacy of understanding). We could continue to develop the code; the line above can also generate other major chords by making changes in the roots in the first array, and we might rewrite to:

```
[60,65,67].collect{|root| root + [0,4,7]}
```

to abstract out the semitonal shape of the major chord ([0,4,7]). Greater generativity can be introduced with respect to the function on the right of the code, and further enclosing code might begin to play with different chord types, or choices applied to the root generation:

```
(
var chordtypes = [[0,4,7],[0,3,7],[0,4,8]];
var allowedroots = [0,1,4,5,7,8,9];
var basenote = 60;
(Array.fill(10,{allowedroots.choose}) + basenote).collect{|root| root + chordtypes.choose}
)
```

Now there is a real tension. To launch straight to this more complex code is to lose sight of the steps required to build up this complexity, and risk students missing the programming moves available for such exploration. The only

reasonable solution I have found is to use live coding as an instructor; to build the program in front of the students, explaining each step of the way and its musical justification. I might break out to a small and necessary component:

```
[0,1,4,5,7,8,9].choose
```

show that isolated code fragment in action returning choices, and only then return to the work in progress to show why it appears and fits in.

For smaller programs, it is often beneficial to involve the students copying the code as it is produced, and then setting time in class for students to spend with the code, trying adaptations and new generalisation. Code a student has spent time typing can provide a sense of personal investment, though programming's dire requirement for perfect syntax can still obstruct getting to the musical outcome and frustrate the student left behind. For example:

```
{ Sin0sc.ar(mouseX.kr(400,800))*0.1}.play //capitalisation wrong, number 0 instead of letter O!
```

or

```
{SinOsc.ar(0.1}.play //missing right parenthesis
```

or

```
{
    SinOsc.ar(0.1)  //student only runs the middle line, no sound
}.play
```

require the workshop leader to rush across a classroom to fix matters on the fly, quickly enough to preserve the momentum of the teaching (dual teaching/workshop accomplices are very helpful with larger classes in keeping up the pace in such circumstances).

Yet, above all else, the chance to explore is exciting, and workshops proceed on the basis of musically motivated adventure. So I have encouraged the students to make mistakes, and tried to show early on that the consequences of syntax errors are minor, are they will not be unintentionally launching a nuclear strike on the Musician's Union.

Although I have built up a set of tutorials for use in teaching, available online (http://composerprogrammer.com/teaching/supercollider/sctutorial/tutorial.html), and covering a wide variety of aspects of SuperCollider and computer music topics, my current preference is not to use these materials directly. Instead, I maximise use of live coding by the instructor, and reveal the nature of computer music programming work directly to the students. After teaching on the topic for so long, including coping with revisions to SuperCollider itself (such as a less intuitive shift from version 2 to version 3), the material is ready at the brain/fingers and few prompt slides are required. Occasionally the existing tutorials or other resources provide more complete examples for

demonstrations and discussion that would be problematic to code from scratch in class. Yet I find myself eager for students to call out requests, and challenge the musical direction of the class; a huge amount can be learnt working together with students through musical problems, jointly considering possible SuperCollider solutions. This embrace of improvisation in teaching may be related to a predilection for improvisation in my live computer music practice, and intimately tied to the rise of experience in performative live coding, but even without the parallel performance career, it is simply efficacious of conveying computer music principles. Instructor improvisation and student-responsive learning are the essential tools for workshop flexibility to participants and their learning needs.

Sometimes co-teaching, the methodologies of those I work with have always complemented my own. I have never failed to learn new tricks and new ways of explaining and exploring from fellow presenters. With a vast field of musical programming possibilities inherent in SuperCollider, and fifteen years of use in, there still remain corners of the software and its usage that can surprise. Such depth reflects both the flexibility of this software for computer music, but also reveals a certain multiplicity of routes that shows the experimental and open source nature of the platform. As a potential problem, this multiplicity includes 'syntactic sugar', multiple ways of expressing the same thing. This may be problematic in learning, but again, musical outcomes trump all else; whether || or arg is used for 'arguments' to a function is quickly bypassed when that function leads to a fun demonstration of analogue synthesizer modelling. Forewarned, the instructor can avoid syntax alternatives wherever possible.

To make clear the inter-weaving of musical and coding themes implicit in this material, Table 1 points out some correspondences between musical scenarios and computer science concepts, with specific SuperCollider constructs that illustrate the connections. This is not intended as an exhaustive list, but a way of pointing to the computer music dialogue engaged in within the educational setting of musical programming. Some computer science devices, such as recursion, can be trickier to find simple musical analogues for, though recursive procedures can be adopted for algorithmic composition, effects processing and the like, and musical hierarchy provides a possible further way in (nonetheless, a recursive function definition of a hierarchy is more easily replaced by tree structures and iteration). To complicate matters, many computer science concepts simply have a particular corresponding SuperCollider syntax, since SuperCollider is itself a full programming language, and certain students may fixate on the pure programming side above of musical examples. Ultimately, different students, and different workshops, may approach from either end of the table!

Table 1 Correspondences between programming and music with respect to SuperCollider code

Computer science	**SuperCollider construct[1]**	**Possible musical scenario**
Iteration	Pseq, Routine, {}.fork,	Sequence of events,

[1] In this table, server and language side constructs are not separated, and methods and classes are not differentiated

	Dseq, Stepper, Select, do…	generation/manipulation of many musical objects
Conditional	if, switch, Index,…	Choice between musical options
Object oriented programming	SynthDef/Synth and Class/instances	Grains within granular synthesis/ instruments within a virtual orchestra/types of sound object within a piece
State, variables	var, Integer, Float, Array,…	Musical data, current musical parameter value(s)
Random number generation, probability	Rand, rrand, choose,…	Stochastic music, generativity
Peripherals	MouseX, KeyState, SerialPort, Arduino,…	New interfaces for music
Graphical user interface	Window, Slider, ….	Performance patch

Undergraduate modules and pair programming

In undergraduate teaching at the University of Sussex, SuperCollider was covered as part of larger scale modules on computer music topics. Many more weeks were available for students to develop their skills, following a guideline of 10 hours per credit (thus for a typical 15 credit module, 150 hours of study over a 12 week module). Aside from lectures and workshops, student self-study formed the majority of the time, rather than the day by day intensive sessions typical to a week long summer school or weekend workshop. Assessment involved creative musical programming tasks, such as creating a sound synthesizer with user interface, or an algorithmic composition, contextualised with respect to computer music precedents; the 'nuts and bolts' of SuperCollider were only assessed in as much as they were necessary to pursue a computer music goal. Because assessment was open-ended, more advanced students had room to push ahead, and the exploratory nature of the software meant that they could undertake more complicated tasks on the side. Nonetheless, more advanced students continually appreciated coverage of more basic material, since they often consolidated knowledge and continued to see tips and tricks of practice from the lecturer.

The most common issues encountered were of the form of the 'generalisation' problems above, where students could make and adapt smaller examples, but hit problems when extending to larger projects. The solution was often to break the students' non-functioning code back into constituent parts and build up again, or to focus on a particular syntactic wall that needed scaling, motivated by the musical goal that actually mattered most to the student. Whilst not all students fully coped with larger projects and the demands of more intensive programming structures, it was always fascinating to see some BA music informatics students (who take more music classes) challenging BSc music informatics students (who take more computer science) in complexity of coding project, and vice versa in musical creativity.

Higher ability students often assisted lower ability students, sometimes informally and sometimes formally through a peer-learning initiative in Sussex computer science. Following John Anderson's comments on the application of psychological research in memory and learning to education (Anderson 2000), learning cannot take place without active engagement and construction of knowledge, no matter how many times something is repeated. Practical work is at the heart of learning SuperCollider through live coding, but social interaction, enforced reflection, and 'pair pressure' (Williams 2011) are all ways to further promote active engagement, whereas individuals working alone may find themselves going through the motions and taking less in. A postgraduate certificate in higher education (PGCertHE) qualification sponsored by the institution and pursued in 2006-2008 allowed me chance to explore this, by an investigation of pair programming as an additional educational methodology.

Pair programming is one part of a package of alternative approaches to code development adopted in the software industry, which also shows much promise as an educational tool (Moore 2014, Williams 2011, van Toll et al. 2007, Teague and Roe 2007). By actively involving students in group work, rather than the traditional model of lone programming, it provides a potential boosting mechanism to facilitate overall student learning. More advanced students may benefit in their representation and formalisation of knowledge from opportunities to pass on advice, whereas less advanced students have an opportunity to learn from peers in a less intimidating setting than direct tuition from a course tutor. Pair programming enforces a period of reflection alternating with action, which may be beneficial to active learning. Perhaps the best argument for an investigation of pair programming in a context of a computer music course is that music is a social art, and combining sociality with programming may assist in the basic liaison of music and computing.

A study was undertaken where students on a Level 2 computer music course dealing on a practical level with the audio programming language SuperCollider were given two afternoon sessions of pair programming separated by four weeks, working on computer music tasks (SuperCollider language exercises and musical pattern creation). They worked in pairs alternating 'driver' (the student at the computer keyboard) and 'navigator' (notionally an observer, but able to comment on the task in hand) around every five minutes. Six pairs took part in the first session, four in the second. Their attitudes were assessed by a written survey after each session. Responses provided interesting evidence of benefits from pair programming that included such comments as:

'differing opinions were really helpful'
'programming can be a social experience but we're not taught that way so not used to it. It's easier to learn something in a pair'
'solve many aggravating little errors'
'interest in the task sustained for longer'
'helpful to have a critical response', 'would like to do it again'
'good to get a second opinion'
'generates positive humility'
'helped me solidify my knowledge by explaining things to partner'
'a good addition to my progress'
'working with peers offers valuable experience'

‘don’t get stuck on minor syntax problems as much’
‘having to explain/verbalise ideas helps to solidify them’
‘easier to spot code errors. Discussing problems makes them easier to think about’.

The experience of pair programming was helpful to the students in general, and welcomed by them as facilitating learning, though there were a few individual concerns on the mismatch of ability level: ‘mismatch in computing styles/speeds can be frustrating’, ‘I feel more competent with programming than my partner’ (there were also those who had no problem as the more experienced partner ‘[I] found it useful to explain things I was writing and get another’s input on potential solutions’). A few students noted that they were new to any form of more sociable programming and hence less comfortable. One student stated that it was ‘odd for someone else watch me try to program - > self conscious of mistakes’ though she also said that it was ‘good…to have my own mistakes pointed out and discussed’.

Although the two sessions did not reveal a unanimous positive response, a majority were in favour of the pair programming approach. The small group sizes, with the second session being less well attended, mean that any generalisations must be approached cautiously. Nevertheless, many elements of the qualitative feedback as outlined above were encouraging.

In running these sessions myself, I was impressed by the overall self-sufficiency of the students and their willingness to engage with pair programming. The speed with which they accepted the process, despite any personal reservations on the efficacy of the technique for their own learning with respect to solo work, may reflect the small class size in the first place, held in a workshop setting which promoted easy group discussion. Elements of my lecturing, where I present work whilst all students can follow along on their own computers and comment and question as they wish, are in some ways analogous to all students having the role of observer/navigator, learning from the tutor-driver. Following Cockburn and Williams (2000) this might be related to ‘line of sight learning in apprenticeships’ and ‘expert in earshot’. I have continued to impose pair programming exercises in SuperCollider workshops since running this study, as a contrasting option and useful icebreaker for class interaction, without any exclusive commitment to it as the sole methodology. As a recent study by David Moore showed (Moore 2014), hybrid approaches which combine multiple modes of teaching delivery, from solo work to group collaboration, can be effective, and there is no need to use pair programming alone.

Widening participation workshops for school groups

In May 2013, widening participation workshops, notionally in ‘creative coding’, were held at the University of Sussex, in seven sessions of two hours reaching six different visiting school groups. Each group consisted of around 20 students from the first two years of high school (years 7 and 8, ages 11-13) and their accompanying teacher, and were held in the 'music informatics lab' of the Informatics department. The workshops were ably assisted by Rob Dawson, a music informatics final year student, who had also created a robotic glockenspiel

which proved highly effective for demonstrations. The promotional spiel for the sessions ran:

From Audiovisuals to Musical Robots: Creative Computing Workshop

This workshop will present a fun mixture of demoes and practical experiments in artistic computer programming. We'll use the computer to generate sound and visuals, try out some alternative controllers like light sensors, and even to control a musical robot! We'll meet two accessible and interesting systems for creative computing, Processing and SuperCollider, which will allow us to produce visuals and sound live. As well as finding out about generative art and computer music, students will gain hands-on experience with writing small programs that have entertaining results. Because both Processing and SuperCollider are free and available for different operating systems, the experimentation can easily continue after the introductory workshop if students get hooked!

In practice, the teaching was split into two hours with a short mid-session break, with the first hour dedicated to live coding in SuperCollider, and the second to introducing Processing with a further chance for students to code. In the latter case, they hacked at existing Processing examples prepared for the class, and in the former, a SuperCollider document of previously written examples was supplied, but the same blank screen that had been a starting point for adult learners was initially presented to the teenagers. All computers were used in pairs and students were encouraged to swap the lead frequently.

As a university educator, I was surprised by how unruly school kids are (though I intervened to manage them); the teachers attending were more surprised by how unruly they weren't! Teachers commented following sessions that the students were more engaged in the material than they had ever seen them before. Although anecdotal, this seems well in line with comments arising from the Sonic Pi project at the motivation of the new 'Minecraft' programming generation (Burnard et al. 2014). For a university lecturer, the sessions provided compelling evidence of the extremely hard work high school teachers put into classroom management and teaching, and although teaching was more exhausting than with university students, the actual ability of the school students was impressive. In all cases, they quickly got to grips with making simple alterations to code; in a few cases, generalisation was apparent, with students starting to make their own more complicated adaptations. Musical and visual coding tricks, like extreme tempo settings, speech synthesis of any text string, and fast strobing visuals, provided a lot of fun for all. The use of live coding with SuperCollider was almost unremarkable for these students; just as with James McCartney's Berlin ICMC workshop in 2000, it simply seemed natural once introduced. The speed of changing and running new code was adopted immediately for artistic outcomes. As for adult learners, more independent and larger scale projects would likely remain trickier, and these short introduction sessions did not allow time to assess that further potential. But the inherent motivation possible through musical and visual outputs under programming control was entirely uncontroversial.

Conclusion

The demands of novel computer music lead students into novel programming tasks. The educational benefits and challenges of live coding are under active scrutiny at the present time. This article has presented one SuperCollider teacher's experience of workshops over more than a decade; increasingly, I have embraced social and participatory learning, from pair programming techniques to student-responsive instructor improvisation (utilising live coding). The aim is to reveal the whole process of working in a musical programming language, motivated by real musical outcomes. As a teacher, I have found myself more and more willing to embrace a constructivist, or constructionist, stance, and to allow the class to dictate their musical problems and interests, and to explore possible solutions together. Improvisation, developed through live coding concert practice carried out in parallel to the acquisition of teaching experience, has proved in the end to be entirely compatible with workshop leading and lecturing.

The time to learn programming to a high level, or to learn a musical instrument to a standard approaching richer expertise, has been estimated at ten years or ten thousand hours (Nilson 2007). In a tongue in cheek barb at the expense of the Associated Board of the Royal Schools of Music, members of the less than recently formed Tempestuous Organisation for the Pushing of Live Art Programming proposed 'live coding grades' in 2004, to outline a pathway of assessment in learning live coding (http://toplap.org/wiki/LivecodingGrades). Yet, students of SuperCollider can be tackling smaller music programs within an hour of encountering the software; and these can be students who have never programmed formally before. Although mastery may take years, just as for accessible instruments such as the piano or guitar, there are possibilities for computer music performance within close reach. Perhaps the punk live coding bands, who never performed with musical computer programming before they walked on stage with their laptops, are soon to displace the dinosaurs of progressive code music.

Acknowledgements

With thanks to two anonymous reviewers, the guest editor of this issue, and all my former students.

Bibliography

Anderson, J.R., 2000. *Learning and Memory: An Integrated Approach* (2nd edition). New York: Wiley

Blackwell, A., McLean, A., Noble, J. and Rohrhuber, J., 2014. Collaboration and learning through live coding (Dagstuhl Seminar 13382). *Dagstuhl Reports* 3(9), pp. 130-168.

Brown, A. R., 2012. *Sound Musicianship: Understanding the Crafts of Music*. Cambridge: Cambridge Scholars Publishing.

Burnard, P., Florack, F., Blackwell, A., Aaron, S., Philbin, C.A., Stott, J. and Morris, S., 2014. Learning from live coding music performance using Sonic Pi: Perspectives from collaborations between computer scientists, teachers and young adolescent learners. *Live Coding Network Live Coding and Collaboration Symposium*, Birmingham.

Cockburn, A. and Williams, L., 2000. The Costs and Benefits of Pair Programming. In *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering* (XP2000), Cagliari, Sardinia.

Collins, N., 2011. Live Coding of Consequence. Leonardo 44(3), pp.207-211

Collins, N., McLean, A., Rohrhuber, J. & Ward, A., 2003. Live Coding Techniques for Laptop Performance. Org*anised Sound,* 8(3): pp. 321-30.

Delaney, M., 2014. Schools Shift from STEM to STEAM [online] Available at: <http://www.edtechmagazine.com/k12/article/2014/04/schools-shift-stem-steam> [Accessed 6 October 2014]

Guzdial, M., 2011. Why is it so Hard to Learn to Program? In: A. Oram, and G. Wilson, eds. 2011. *Making Software*. Sebastopol, CA: O'Reilly Media, Inc. pp. 111-124

Hancock, O., 2014. Play-based, constructionist learning of Pure Data: A case study. *Journal of Music, Technology & Education*, 7(1), pp. 93-112.

McCoid, S., Freeman, J., Magerko, B., Michaud, C., Jenkins, T., Mcklin, T., and Kan, H., 2013. EarSketch: An Integrated Approach to Teaching Introductory Computer Music. *Organised Sound*, 18(2), pp. 146-160.

Moore, D., 2014. Supporting students in music technology higher education to learn computer programming. *Journal of Music, Technology & Education*, 7(1), pp. 75-92.

Nilson, C., 2007. Live Coding Practice. *Proceedings of NIME*, New York.

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

Ruthmann, A., Heines, J. M., Greher, G. R., Laidler, P. and Saulters, C., 2010. Teaching computational thinking through musical live coding in scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education* (SIGCSE '10). New York, NY.

Sorensen, A., 2014. The Concert Programmer. Open Source Convention (OSCON) Keynote. [online] Available at: <http://www.youtube.com/watch?v=yY1FSsUV-8c> [Accessed 6 October 2014]

Teague, D. and Roe, P., 2007. Learning to Program: Going Pair Shaped. ITALICS 6(4), October 2007. Available at http://www.ics.heacademy.ac.uk/italics/vol6iss4.htm

van Toll, T., Lee, R. and Ahlswede, T., 2007. Evaluating the Usefulness of Pair Programming in a Classroom Setting. *Computer and Information Science*. pp.302-308

Williams, L., 2011. Pair Programming. In: A. Oram, and G. Wilson, eds. 2011. Making Software. Sebastopol, CA: O'Reilly Media, Inc. pp. 311-328.

Wilson, S., Cottle, D. and Collins, N., eds. 2011. The SuperCollider Book. Cambridge MA: MIT Press.