

Code as information and code as spectacle

Charles Roberts

School of Interactive Games and Media, Rochester Institute of Technology, Rochester, NY, USA

ABSTRACT

In this artist statement, I will discuss the tension between source code as an interactive system for performers and source code as information and entertainment for audiences in live-coding performances. I then describe augmentations I developed for the presentation of source code in the live-coding environment *Gibber*, including animations and annotations that visually reveal aspects of system state during performances. I briefly describe audience responses to these techniques and, more importantly, how they are critical to my own artistic practice.

KEYWORDS

Live coding; psychology of programming; notation; audiences; algorithms

How can the text of programs engage an audience that doesn't understand code? And how can the programs I write do so in a way that I also find meaningful? The interfaces that live coders use (and present) in live-coding performances become, perhaps, the most critical visual elements of performances, more so than in other musical idioms where the appearance and actions of the performer are the primary visual stimuli. Ideally, live-coding interfaces should inspire and inform audiences and performers alike.

When I first began live coding, I was primarily attracted by its potential for real-time composition. But in my first live-coding performance, I also fetishised what I thought (at the time) were the virtuoso aspects of live coding. I created small animations for source code that would shake it, blur it, confuse it; why not add to the pressures of a live performance with obfuscation and obstruction? While I was intrigued by the idea of increasing the difficulty of a performance practice that already seemed complex, I was also trying to make code, and the act of coding, more exciting for the audience. This first performance ended with a crash after about five or six minutes; perhaps I should have focused my attention on algorithms instead of appearances.

But years later I find I am still very interested in the appearance of code, and particularly in improving the understanding of code and algorithms for both audiences and performers. I primarily perform using a live-coding environment I developed, *Gibber* (Roberts and Kuchera-Morin 2012). *Gibber* is a browser-based live-coding environment¹ with affordances for generative audiovisuals. Its end-user programming language is JavaScript, and the techniques described in this paper are used to automatically annotate the JavaScript written by users. These annotations are the key feedback mechanism informing both audiences and performers of algorithmic development.

Annotating patterns

Most of these annotations provide insight into the development and playback of musical patterns. For example, consider a pattern of numbers, each corresponding to positions in a musical scale, which is used to trigger notes on a synthesiser:

0 1 2 3

One possible annotation for the playback of this pattern would be to bold each pattern member as it generates output. If pattern playback progresses linearly, the resulting annotations would change as follows:

0 1 2 3 -> 0 **1** 2 3 -> 0 1 **2** 3 -> 0 1 2 **3**

This simple annotation provides a variety of indicators. First, the change in the annotation is a visual indication of activity. It attracts the eye and subsequently attracts the ear as we try to hear corresponding changes in the generated music. Second, by highlighting the current outputs of patterns performers and audience members can associate changes in sonic properties with pattern progression. Finally, the annotation can help reveal underlying algorithms that affect pattern playback. For example, perhaps an algorithm is running that only outputs even-numbered pattern values while the odd values are skipped. Given a longer pattern we would see the following progression:

0 1 2 3 4 5 6 -> 0 1 **2** 3 4 5 6 -> 0 1 2 3 **4** 5 6

Such manipulations are quickly revealed when watching annotations change over time, while hearing their sonic results in isolation might not enable audiences to identify their behaviour.

As playing back a static pattern over the length of an entire performance can be rather dull, transforming patterns over time becomes an important activity in the practice of many live-coding performers. Before I began work on Gibber, I saw Thor Magnusson perform with his *ixi lang* (Magnusson 2011) environment, where program text dynamically changes to reflect the current state of patterns after algorithmic or direct manipulation (Magnusson 2015). For example, let us consider our previous pattern but transpose it up by a value of one each time it plays; the pattern goes through the following changes over four iterations:

0 1 2 3 -> 1 2 3 4 -> 2 3 4 5 -> 3 4 5 6

Seeing the transformation take place in the code itself in addition to hearing it is compelling. For audiences unfamiliar with programming language syntax, the code needed to transpose this pattern over time might be unreadable. While this simple algorithm might be understood by a person with a musical background through listening alone, I believe representation through multiple modalities clarifies how the algorithm functions. I was immediately struck by the potential of the technique when I first saw it used, and implemented it in Gibber (see [Figure 1](#)). And rhythmically rotating/altering patterns creates programs moving to the beat of a performance; the source code is dancing.

```

gibber  code  browse  publish  chat / collaborate  console  preferences  help  credits
x| id #: 1  language: javascript  #  ?

Notation.on()
a = Synth('shimmer').chord.seq( Rndi(0,15,3)/* [5,13,2] */ , [ 2 ] )
a.pulsewidth.seq( Rndf(.05, .95)/* 0.1018 */ , [1/16] )

kick = Kick('bright').note.seq( [70], Euclid(4,4).concat( Euclid(5,8) ) )

reverb = Bus('stereospace')

hat = Hat({ offset:1/4 }).note.seq( [1500], [1/2] )

glitch = Sampler().loadDir( local('sineKit') )
glitch.note.seq( [-4,-2,-1,-.5, .5, 1, 2, 4].rnd(), [1/4, 1/8, 1/16].rnd(1/16, 2) )

delayL = Bus('oneshot').pan( -1 )
delayR = Bus('oneshot').pan( 1 )
delayL.fx[0].time.seq( [1/16, 1/8, 3/16, 1/4, 5/16].rnd(), [1/4, 1/2, 1].rnd() )
delayR.fx[0].time.seq( [1/16, 1/8, 3/16, 1/4, 5/16].rnd(), [1/4, 1/2, 1].rnd() )

h2o = Synth('watery').note.seq( [14, 11], [2, 6] )

pops = Drums('x*ox*xo-', 1/16 )
pops.note.values.reverse.seq( [1, 1] )
pops.fx[0].cutoff.seq( Rndf()/* 0.4756 */ , [1/16] )

lead = Synth('bleep').note.seq( [20, 18, 16, 14, 15], [1/2, 1/2, 1/2, 1/2, 6] )
lead.note.values.transpose.seq( [1, 8] )
lead.note.values.reverse.seq( null, 8 )

```

Figure 1. A screenshot from a Gibber performance by the author. Note the bordered boxes which depict the value currently being triggered by sequencers, and code comments (demarcated by `/*` and `*/`) that reveal the output of random number generators as they are triggered. All rights to this image are held by the author.

Dancing code, dancing data

This dancing is both information and spectacle, and the potential of code as spectacle is a tantalising concept that I continue to try and capitalise on. In one experiment, I extended Gibber so that the font characteristics of code could be manipulated and used as continuous displays of information (Roberts, Wright, and Kuchera-Morin 2015). For example, imagine a line of code that creates a synthesiser moving vertically in response to changes in the pitch the synthesiser is playing. Or that the size of a code block could be used to dynamically portray the current volume of a drum loop it created. These ideas, while originally compelling to me, quickly lost interest due to their highly impractical effects on interaction; it is difficult to edit text that is rapidly changing position and size in response to audio signal analysis. The obfuscation of source code, once perversely interesting to me in my first live-coding performance, was now simply interference, and using font characteristics to convey continuous data proved ineffective.

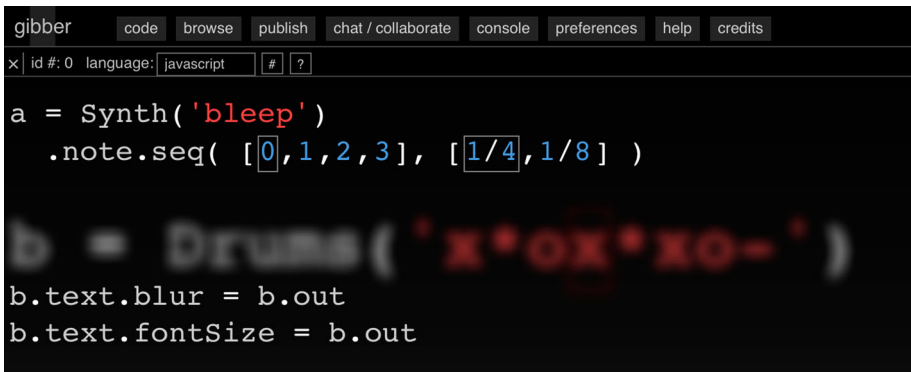
However, the potential for visual metaphor using these types of typographic manipulations still intrigues me. As a simple example, imagine gradually fading the volume of an entire performance, and having the source code fade from view in response. The visual state of the source code becomes a reinforcement of the musical transition in

progress. This particular example is effective because a fade of the master volume in a performance typically signals the performance's end; as the live coder does not need to continue to interact with source code obfuscating the text by gradually fading it doesn't impede interaction. A similar idea was explored by Wilk (2015) in his performance titled 'End of Buffer', in which the source code of his live-coding performance rapidly flies across the screen and disintegrates as his performance comes to a halt.

There is thus a tension between source code as spectacle and source code as an interactive medium in live-coding performances. This was captured in two back-to-back responses to a survey I conducted on the use of visual annotation techniques in live coding (Roberts, Wright, and Kuchera-Morin 2015). As part of the survey, I asked over a hundred subjects to watch video of a source code fragment that was blurred to continuously visualise the amplitude of the drum loop that it generated. Thus, as the drum loop played louder notes, the source code that created the drum loop became increasingly blurred in its editor (see Figure 2). While subject #8 remarked that 'Size change and blurring look like gimmicks rather than useful HCI features', subject #9 offered a counterpoint: 'the blur effect looks dope'. And although the word 'gimmick' was perhaps intended pejoratively by subject #8, I don't disagree with this assessment. Attracting audience attention was always the goal for implementing these types of font manipulations in Gibber and the response by subject #9 suggests the manipulations are at least somewhat successful in this regard. Unfortunately, it turns out that, at least as they are currently implemented in Gibber, the impediments to interaction these typographic manipulations impose outweighs their dramatic benefits. One alternative approach is to present different screens to audiences and performers; we are then no longer limited by the constraints of interaction and can freely use the code as material for generative visualisations that are presented to the audience.

Performers like spectacle too

We should also consider the possibility that the visual presentation of programmes might be the primary content of a performance. For example, consider the live-coding performances of Dave Griffith, who '... considers the music he makes to be a side product, rather



```
gibber  code  browse  publish  chat / collaborate  console  preferences  help  credits
x id #: 0 language: javascript  # ?
a = Synth('bleep')
  .note.seq( [0,1,2,3], [1/4,1/8] )
b = Drums('x*ox*xo~')
b.text.blur = b.out
b.text.fontSize = b.out
```

Figure 2. In the top code fragment, the current playing note and duration is annotated with a border. In the bottom fragment, the source code that creates a drum machine dynamically blurs and changes in size in response to the drum machine's output. All rights held by the first author.

than an end-product of his live coding languages, where the visual aesthetics of his interfaces are more important' (Collins and McLean 2014). This approach is possibly more intuitive in the visual programming languages that Griffith often creates, although the idea has interesting implications for textual programming languages as well.

The balance between code as spectacle and information is a moving target in my performances. But as long as the live-coding community continues to project source code (an active topic of contention), it seems reasonable to assume that some attention should be given to how the code is presented, and what it is capable of conveying to audiences. While one could argue that a pianist pays little attention to the visual appearance of their piano during a performance, and live coders should similarly bear no responsibility for the appearance of their interface, there are direct physical correlations between the physical movements of performers and pianos and the sonic output of the instrument; the pianist gets these 'for free'. Faster vertical movements of keys on the piano result in faster progressions of notes. Moving up the keyboard results in higher pitches. These gestures are intuitively understandable by audiences in ways that code is not, at least for those who are unfamiliar with programming conventions. The implication here is perhaps counter to an acknowledgement in the TOPLAP live-coding manifesto that reads: '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' (Ward et al. 2004). The manifesto was written over a decade ago and perhaps this assertion still holds true, but some effort to establish relationships between code and musical output would no doubt be helpful in furthering appreciation.

Despite my interest in how audiences perceive code during performances, a simpler rationale for the annotations described here is their impact on my personal live-coding practice. When I sit down to practice or perform, I immediately notice if I've forgotten to enable the annotations and quickly stop to do so. Coding without the feedback provided by the annotations now seems wrong at a visceral level; I miss the feedback, I miss the animations, I miss viewing the program as a dynamic, changing entity. In the end, I miss the dancing.

Note

1. <http://gibber.cc>.

Disclosure statement

No potential conflict of interest was reported by the author.

Funding

This research is supported by the Rochester Institute of Technology, with funding from Sponsored Research Services and the Golisano College of Computing and Information Sciences.

Notes on contributor

Charles Roberts is an Assistant Professor in the School of Interactive Games and Media at the Rochester Institute of Technology, where his research examines human-centred computing in digital arts

practice. He designed and developed a creative coding environment for the browser, Gibber, that is used both for educational research and audiovisual performances. Gibber is used to teach computational media to middle school, high school and university students in locations around the world, and Roberts has performed with it throughout the US, UK and Asia in the experimental performance genre known as live coding.

References

- Collins, Nick, and Alex McLean. 2014. "Algorave: A Survey of the History, Aesthetics and Technology of Live Performance of Algorithmic Electronic Dance Music." *Proceedings of the New Interfaces for Musical Expression Conference*, London, 355–358.
- Magnusson, Thor. 2011. "ixi lang: A SuperCollider Parasite for Live Coding." *Proceedings of the International Computer Music Conference*, Huddersfield, UK, 503–506.
- Magnusson, Thor. 2015. "Code Scores in Live Coding Practice." *Proceedings of the First International Conference on Technologies for Music Notation and Representation*, Paris, France.
- Roberts, Charles, and JoAnn Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." *Proceedings of the International Computer Music Conference*, Ljubljana, Slovenia, 64–69.
- Roberts, Charles, Matthew Wright, and Joann Kuchera-Morin. 2015. "Beyond Editing: Extended Interaction with Textual Code Fragments." *Proceedings of the New Interfaces for Musical Expression Conference*, Baton Rouge, LA, 126–131.
- Ward, Adrian, Julian Rohrerhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. 2004. "Live Algorithm Programming and a Temporary Organisation for Its Promotion." *Proceedings of the README Software Art Conference*, Aarhus, Denmark, vol. 289, p. 290.
- Wilk, Joseph. 2015. "Animations with Emacs." Accessed January 2015. <http://blog.josephwilk.net/art/emacs-animation.html>.