’Code considered harmful’: on Gesture and Mental Dexterity in Live Coding

Will Rinkoff

1 Intro

Gesture has always been difficult to reconcile with electronic music, as we have 20 years of NIME conference proceedings trying to tackle the question. We design incredibly complex systems capable of synthesizing any sound we can conceptualize, yet we have to reduce that complexity to an interface we can actually perform with. Dobrian and Koppelman allude to this problem as a distinction between control and expression, ‘control’ being the exposed state of the system that we can endlessly tweak and design sound with, and ‘expression’ being a mapping across those controls to an interface we can use live in a performance setting.[1] The sea of VST plugins and max patches and eurorack modules are largely concerned with crafting systems and controls. It’s uncommon for them to be concerned with expression.

With all this considered, live coding, the act of programming music on stage with code as performance, is a rather strange case for a number of reasons. For the vast majority of use cases, code sits squarely in the systems domain. It’s a means of communicating to a computer what a system is, and in the context of music technology that system exposes a set controls for us to make music with. We don’t often think of it reaching the expressive domain, as code wasn’t really approached as an interface for a performer. But some noble souls dutifully ignored this, and tried anyway, first in the 80s with Ron Kuivila at STEIM (STudio for Electro Instrumental Music) coding in Forth until his system eventually crashed, on through the 90s with the development of the audio programming language Supercollider, and then the 2000s where it truly came into its own as a practice.[2] You had Adrian Ward and Alex McLean performing as *SLUB* in UK nightclubs with Perl and Basic, leading to the foundation of TOPLAP as an organization to explore and promote live coding, and the release of the ‘TOPLAP manifesto’: a short, opinionated document on live coding performance practice that would serve as a blueprint for next 20 years of exploration.[3] Out of this came papers, performance series, meet up groups, careers, dozens of live coding languages, and even some music.

But the conceptual anchor for exploration has always been algorithms, which is to say code and mental dexterity. Live coding was an experimental practice, born as a subversion of the ‘system/control/expression’ hierarchy I’ve alluded to. I’d like to assert that after decades of iteration, live coding performance is no longer an experimental practice. It is very well established as a community of practitioners, a set of tools, a school of thought, and as an approach to computer music. The rest of this paper will be spent advocating the following 2 points:

- that live coding stands to benefit from a shift towards focusing on expression and virtuosity, and away from the cognitive load of traditional programming
- that such a shift can and should be done by reasoning about the computer keyboard as an interface capable of being highly expressive.

Some quick notes on terminology. The term ‘live coding tools’ in this paper encompasses the full spectrum of specialized live coding languages and environments. For example, Strudel is a live coding language, but also an environment, in that it has a specialized IDE (integrated development

[1] Dobrian and Koppelman, The E in NIME.

[2] Blackwell and Collins, The Programming Language as a Musical Instrument; Roberts and Wakefield, Tensions and Techniques.

[3] Collins et al., Live Coding in Laptop Performance; Blackwell and Collins, The Programming Language as a Musical Instrument; TOPLAP, Manifesto Draft.

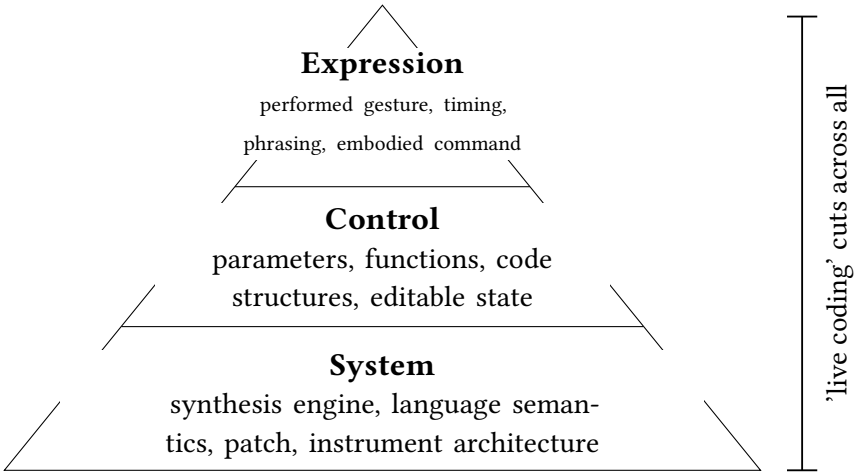


Figure 1: The system/control/expression hierarchy. The expression layer is names the performer's actionable, practiced relation to the system. Live coding is unusual because code can describe the system, expose controls, and become part of the performer's expressive interface.

environment).[4] The same goes for ChucK and the miniAudicle, as well as Supercollider and the Supercollider IDE.[5]

2 Towards Expression and Virtuosity

A live coding performer typically has a computer with an IDE open to one or more text buffers filled with code, the code denoting a program that produces music. The performance then consists of a fairly simple loop: conceive of an edit, edit the code, then evaluate it, updating the program and changing the sound it produces. This editing step actually takes the bulk of a performer's time and is a major barrier to expression. In traditional instrumental performance, there is no editing step. To perform with a guitar is to conceive of a sound or gesture, then to perform a physical gesture reflecting that sound (strumming the guitar with a specific fingering), the gesture itself producing the sound. This close coupling of gesture and sound is what affords expression. I've introduced it as a two-step process but it's really instantaneous. Ideation and physical action blur together in most live instrumental contexts. Georgios Diapoulis, who's laid groundwork for analyses of expression in live coding, termed this a 'pre-reflective processes'.[6] Diapoulis, among others, has observed that pre-reflective processes are both critically important to live expression and are lacking from most live coding tools.[7] Given the performance loop, a reasonable way to address this problem is to just remove the editing step, or make editing as fast as possible, to eliminate the time between the ideation and realization of ideas. The live coding language and IDE are equally relevant to this issue. It's a question of what IDE features afford rapid editing, and what features of a live coding language enable and encourage rapid editing.

[4]Roos and McLean, Strudel.
[5]Collins et al., Live Coding in Laptop Performance; Wang, The ChucK Audio Programming Language.
[6]Diapoulis and Dahlstedt, The Creative Act of Live Coding Practice; Diapoulis and Dahlstedt, An Analytical Framework.
[7]Diapoulis and Dahlstedt, An Analytical Framework; Diapoulis, Musical Live Coding.

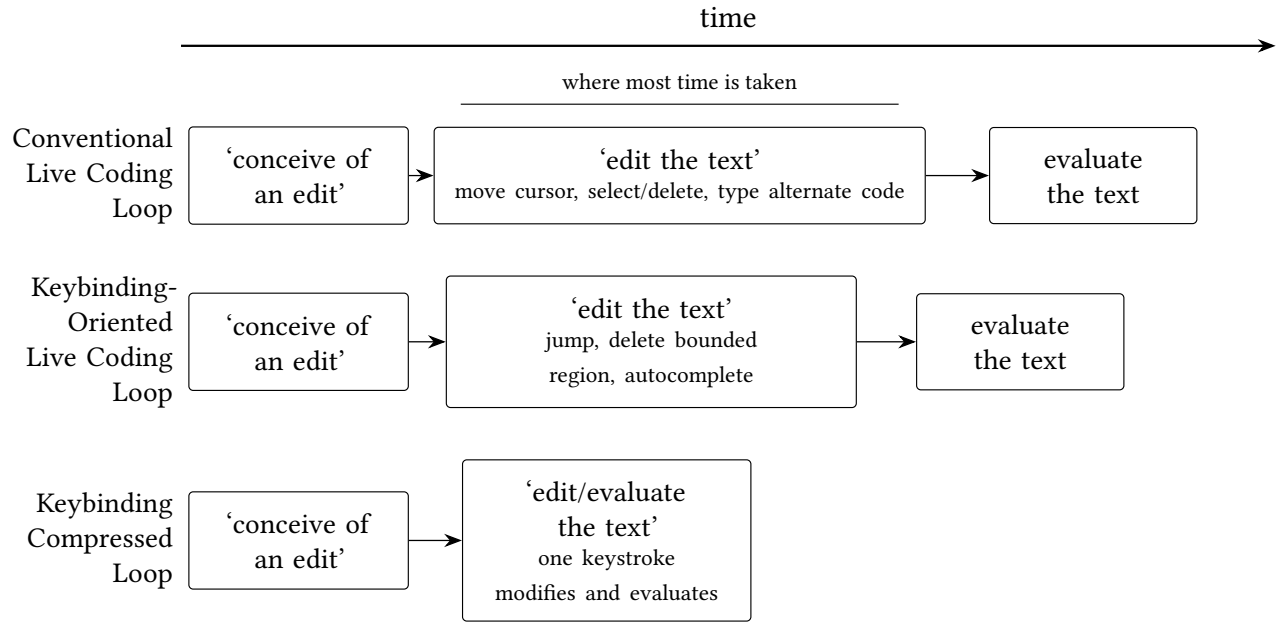


Figure 2: The conventional loop separates musical intention from sonic change through a text-editing phase. A compressed loop minimizes this text-editing phase, easing virtuosic expression.

I need to draw attention to my terminology: to 'conceive of an edit' in the loop of live coding performance is something more specific than conceiving of a musical idea. To conceive of an edit is to understand exactly the code you want to modify. It may take extra reasoning on the part of the performer to take an idea occurring as a series of notes or a rhythm or a specific set of audio effects and translate it to code that realizes the idea, but I'd like to stress that the idea can simply occur to the performer in their mind's eye as source code. Ideas can be conceived of directly as code without translation. This is important to establish this because all live music performance systems enable their own kinds of virtuosity, their own kinds of extremely high technical proficiency. It's this virtuosity, built through practice and understanding and expressed through muscle memory, that enables pre-reflective processes. For the purpose of this paper, my notion of virtuosity is very specific, and less speculative than Parkinson and Bell's.[8] I am unconcerned with any means through which virtuosity is telegraphed to an audience, like showing one's screen. Virtuosity is full understanding of, and command over a system. For virtuosic piano players, playing chords and navigating harmony is a pre-reflective process. For virtuosic live coders, the code should be pre-reflective, and the only barrier to expression should be the realization of that code in the IDE. Counter-intuitively, pre-reflective ideas that manifest in source code don't necessarily reflect anticipations of how exactly the code will sound once evaluated, but such is a part of live coding, and of electronic music in general. It's a symptom of how powerful and expansive the systems are.

To accommodate rapid editing in live coding language design is to enable virtuosity. And to enable virtuosity is to understand the relationship between pre-reflective processes and complexity. This is quite difficult, and something live coding discourse has approached well. Many different frameworks, checklists, research questions, and models have been constructed to approach live coding language design.[9] I won't go as far as to construct another, nor will I provide exhaustive analyses, as such discussion is extremely diverse and hasn't found much common ground. I'll simply assert that the majority of discussion doesn't explore live coding as a practice that affords pre-reflective process. This is to say that discussion of live coding performance is dominated by an understanding that live coding is an act of coding, and that the cognitive load associated with coding is inherent to the

[8]Parkinson and Bell, Deadmau5, Derek Bailey, and the Laptop Instrument.

[9]Kiefer and Magnusson, Live Coding Machine Learning and Machine Listening; Blackwell and Aaron, Craft Practices; Blackwell and Collins, The Programming Language as a Musical Instrument; Diapoulis and Dahlstedt, An Analytical Framework.

medium. Roberts and Wakefield's chapter on 'Tensions and Techniques in Live Coding Performance' is an excellent survey of live coding performance practice, interviewing many performers with mature, well-developed approaches. But both its frame and discussion lay firmly in approaching live coding as coding, and carry the baggage of thinking in algorithms as divorced from code, requiring the cognitive load of translation. One of its recurring tensions concerns how performers manage stability, risk, flow, and errors in real time.[10] This echoes Blackwell and Collins's earlier emphasis on live coding's challenge, cognitive load, error-proneness, and diffuse activity.[11] This frame of live coding requiring cognitive load has tremendous historical precedent since the field's inception, most critically in the oft-cited TOPLAP manifesto. In its principles for live coding performance, it explicitly advocates for 'mental dexterity' and 'insight into algorithms'.[12] From live coding's inception through decades of history and iteration, live coding has been framed a cognitive process, with the creative act working one level abstracted from the actual live coding tools. This frame creates an irreconcilable divide between live coding and pre-reflective process, and by extension a significant domain of exploration.

To clarify, I'm critical only of how live coding is discussed, not any specific tool or school of thought. It's an incredibly diverse array of practices without very consistent ideas about language construction or design. It's rejected definition from its inception, and this is something I firmly support.[13] This is why it's critical to conceive of a live coding practice that can exist outside the cognitive frame of programming, to conceive of tools and users who are truly virtuosic with them, to the extent where the text barrier is negligent and the only true barrier is text entry.

3 The keyboard interface

Expression and virtuosity in live coding tools are afforded in part by minimizing the time taken to edit code. Text entry, or rapid editing, is an under-explored domain of live coding performance. There is no shortage of novelty in live coding interface design, but much of that novelty comes from attempts to escape using the keyboard, or the IDE altogether and use another bespoke interface.[14] The tools that don't have fairly limited live coding tools have at least a small set of keyboard macros related to executing or stopping the execution of code, which are core to the performance loop. Many IDEs have built-in autocomplete features that can take a few input characters quickly expand them to function names or larger, pre-written chunks of code. These features, macros and autocomplete, are standard features in the modern IDE, and comprise most of what is brought up in live coding discourse.[15] However they're typically brought up in passing, whereas I believe there is substantial space for exploration here. To enable that exploration I propose a simple model for gesture.

Gesture in live coding is the act of editing text, which is to say using the computer keyboard and mouse. Diapoulis considers this, but I'd like to commit to it because of how cleanly it couples live coding as a practice to pre-reflective process.[16] We must consider every interaction with the computer, every keystroke and input received from the mouse or track pad, if we are to reason about what live coding can truly afford as a medium. We must think about how we can make every input as meaningful as possible.

In most contexts, macros and autocomplete are simply an IDE's ease-of-use features because

[10]Roberts and Wakefield, Tensions and Techniques, 293–318.

[11]Blackwell and Collins, The Programming Language as a Musical Instrument.

[12]TOPLAP, Manifesto Draft, "Principles".

[13]Blackwell et al., *Live Coding*.

[14]Diapoulis and Dahlstedt, An Analytical Framework; Salazar, Searching for Gesture; Fiebrink, Wang, and Cook, Don't Forget the Laptop; Armitage and McPherson, The Stenophone.

[15]Roberts and Wakefield, Tensions and Techniques; Blackwell, Patterns of User Experience; Kirkbride, *Collaborative Interfaces*; TidalCycles for VSCode; Strudel source repository.

[16]Diapoulis and Dahlstedt, An Analytical Framework; Diapoulis, Musical Live Coding.

programming isn't traditionally a domain where the timescale of keystrokes is useful to reason about. But in live coding, it fundamentally shifts what musical ideas a performer is comfortable exploring. Suppose a live coder conceives of a simple edit. Maybe they have a recorded audio sample of a drum being played back in a rhythm, and they'd like to change the sample being played. Assuming the simplest possible text editor, they'd have to use the mouse or arrow keys to move the cursor to the region of characters in the code representing that sample, delete that region, and replace it with new characters. This region may consist of a string that has to be typed out, or maybe it's an index into an array of samples, but this process takes time and conscious effort regardless of how the data is represented. With macros, you could assign the action of moving the cursor to a short series of keystrokes, for example `<ctrl-g> s`. If the region is a string, you have another macro to delete the region then use autocomplete to type the first few characters and select from a list of potential completions with the number keys 0 through 9 plus some modifier key. This approach does require visual feedback from the editor for a performer to modify code deterministically, but could cut an edit that takes 6 seconds down to an edit that takes 2 seconds, and expand what a performer can express throughout the finite time of a performance. If the region is just an indexing value, you could have another macro for incrementing that value, like `<ctrl-a>`, which is the default macro for incrementing an integer in the Vim text editor.[17] This would allow a near instantaneous edit, consisting of a rapid chain of 2 or 3 different macros for moving the cursor and editing the code, further cutting down on time, and expanding what can be expressed in a time-sensitive performance context.

Edit route	**Typical action chain**	**Cognitive Load**	**Syntax risk**	**Repeatable as gesture**
Manual text edit	locate region, select/delete text with mouse, type replacement, evaluate	high	medium/high	low
Vim or snippet edit	jump to text with key bind, highlight and delete text with keybind, type replacement, evaluate	medium	medium	medium
Macro edit	jump to text with key bind, change text with few-keystroke deterministic edit, evaluate	low	low	high

Figure 3: Input cost changes what edits are likely to enter performance. But at the macro/stage.

I'd argue this latter approach, editing entirely through macros, has the capacity to tightly couple gesture and sound. If a virtuosic live coder can pre-reflectively conceive of code, and the means of entering that code is via a deterministic set of keyboard macros, then a virtuosic live coder will pre-reflectively conceive of macros, which is to say gestures across the computer interface.

There exist tools that explore macros and autocomplete in minor ways, but very few papers approaching it.[18] Certain environments for TidalCycles have keybindings available for enabling/disabling regions of code.[19] Strudel has autocomplete for chords and scales, and a function called 'whenKey', that enables functions to be applied to a running process only when certain keys are pressed.[20] Fiebrink et al. assert that the laptop interface is expressive in 'Don't Forget the Laptop', but limit their exploration to direct mappings of input to sound.[21] The only project I've found that digs into macros as core to the act of live coding has been the Stenophone from Armitage and McPherson,

[17] This is a concrete editor example rather than a live-coding citation; Vim's normal-mode `CTRL-A` increments the number at or after the cursor.

[18] Roberts and Wakefield, Tensions and Techniques; Blackwell, Patterns of User Experience; Armitage and McPherson, The Stenophone.

[19] TidalCycles for VSCode.

[20] Strudel source repository.

[21] Fiebrink, Wang, and Cook, Don't Forget the Laptop.

which uses a special type of keyboard where keys are chorded together as an interface for the live coding language TidalCycles.[22] The project uses specialized hardware but mostly to leverage the stenograph's analog input for a system independent of the chording system.[23] Such a chording system could easily be replicated on a standard computer keyboard. Where it falls short is that it's primarily concerned with writing code and not editing written code, but the scope of the system is fairly small.

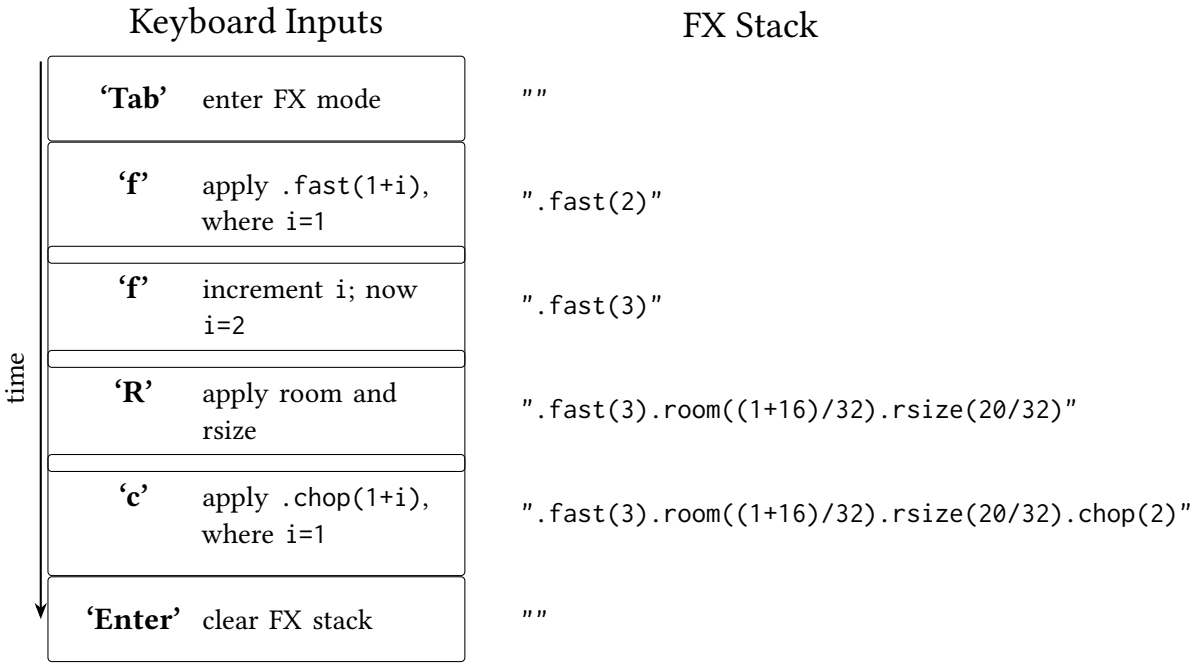
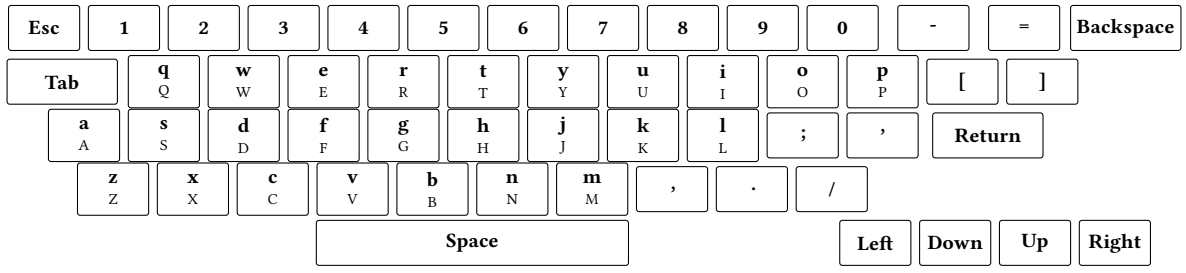


Figure 4: The FX binding system behaves like a small transformation language over the running performance state. One keypress updates a stack of code transformations, re-evaluates the result, and changes the sound without exposing the performer to ordinary syntax editing.



Esc: leave mode
Tab: enter/leave FX mode
Backspace: remove selected FX
Return: clear FX stack
Shift-Return: clear FX and solos
1–9: solo/unsolo d1–d9
0: clear solos
Shift-0: solo d1–d9
- / **=**: decrement/increment selected value
Alt– / **Alt-=**: halve/double selected value
[/ **]**: jump previous/next `//do` and evaluate
Alt-h / **Alt-l**: select previous/next FX
, / **.**: hold tempo nudge down/up
Shift-, / **Shift-.**: fine tempo nudge down/up
Ctrl-p/x/s/c: pin piano roll, XY, spectrum, scope
Ctrl-Return: clear pinned views
q: squiz
Q: no-op
w: echo offset
W: echo taps
e: modulated delay
E: wide modulated delay
r: reverb
R: large reverb
t: echo repeat
T: gain echo repeat
y: no-op
Y: no-op
u: moving delay
U: wide moving delay
i: bite slices
I: no-op
o: bit crush
O: coarse rate
p: ply and slow
P: ply 3/2 layer
a: no-op
A: no-op
s: slow
S: shape distortion
d: light distortion
D: full distortion
f: fast integer
F: fast fine
g: speed modulation
G: fine speed modulation
h: hurry integer
H: hurry fraction
j: fine hurry
J: degrade
k: swing
K: no-op
l: linger
L: short linger
z: add highpass
Z: clip
x: add lowpass
X: accelerate
c: chop
C: cut and orbit
v: no-op
V: reset global effects
b: early small
B: early large
n: add sample index
N: subtract sample index
m: compressor
M: smear

Figure 5: QWERTY map for the FX binding system. The legend maps each lower-case and upper-case key to the transformation or control action it performs.

[22]Armitage and McPherson, The Stenophone.
[23]Armitage and McPherson, The Stenophone.

I don't wish to champion any single approach to making keystrokes/inputs more meaningful, but as a concrete example, I must briefly describe a macro system I've been using for years as an extension of TidalCycles, and now of Strudel. I call it the 'FX binding system' because it's a means of rapidly working with global effects. Suppose I have Strudel code already running and producing sound. Pressing the 'Tab' key enters an alternate mode where every key is mapped to a function, and pressing a key adds that function to a stack of functions applied to all running code and immediately re-evaluates it. Currently, these functions encompass audio effects (delay, distortion, reverb, filters, etc), as well as pattern effects ('repeat the measure', 'double the speed of all patterns', 'for each note, add an extra note transposed up by a factor of 3/2', etc). Each function takes an integer argument that I can increment/decrement with '+' and '-', I can change which function in the stack I have selected with the arrow keys or with 'alt-h' and 'alt-l', I can delete functions with 'Backspace', and I can clear the entire stack with 'Return'. Functions can interact arithmetically as well, as I can apply a function that adds to a value by 'x', then apply another function that multiplies that same value by 'y' elsewhere in the stack. All changes to the stack are immediately applied to the running code, which I've found has enabled a considerable amount of pre-reflective improvisation. I can be intentional with my keystrokes, to help build to a moment in the composition, or I can simply enter them randomly (eg 'laksciusesh') and rapidly contort the running sound, then reset the transformation with a single keystroke. I do not think when I use this system, there is no cognitive-load. I intuitively understand how individual keys will modify the sound, and know that I won't suddenly produce errors in the code (as long as the computational load produced by the effects doesn't crash the computer). Some keys in some scenarios I cannot fully anticipate the effect of, such as one effect that changes the index of which samples are being played back across all running patterns of samples, but such is just a consequence of being able to explore a massive possibility space.

I conceptualize this system as a highly specific DSL where every single configuration of input characters will successfully compile. One could take this idea, 'a DSL that always compiles', and use it to reason about IDE features that afford improvisatory editing, where a performer doesn't need to worry about syntax issues or compiler errors. One might argue this places significant constraints on both what edits can be made and what languages would accommodate such a system. This might be true, but we are nowhere near understanding that limit. Systems related to such an inquiry have already been explored. Keyboard macro-heavy applications like Vim, Emacs, Kakoune, and Helix, as well as tiling window managers for linux including i3, dwm, and xmonad, are already quite mature and have communities and tools invested in the question of how fast we can use our computer.[24] There is a clear and fascinating area of exploration a marriage of these systems and live coding would enable.

4 Conclusion: Live Coding's value

We've come a long way from hacking perl in nightclubs. The development of new browser standards, like web assembly and audio worklets, have made browser based tools more powerful, and in turn enabled the development of powerful web-based live coding tools. The past year has seen a viral surge in interest around these tools (mainly Strudel and Hydra) and what can be accomplished with them. Anecdotally, when I chat with live coders who are a part of this new wave, none of them seem interested in 'the challenge' of live coding. Some are interested in exploring algorithms, some care about showing their screen, and many are interested in the notion of 'live coded music'. But mainly I think people are interested in live coding because the tools are just really good now. It's an excellent way to make live electronic music, in an age where most music is made by the computer

[24] These editor and window-manager examples are not live-coding systems; they are adjacent examples of mature keyboard-driven interaction cultures that could inform live-coding interface design. See the Vim, GNU Emacs, Kakoune, Helix, i3, dwm, and xmonad project documentation.

but very little music is performed using its interface. I first became interested in live coding not because I was interested in the challenge or the mental dexterity or really any understanding of live coding as 'coding', but because I wanted to perform with my laptop, and it seemed like with live coding you could do anything. Code is an interface for music with infinite complexity: It's where the 'system/control/expression' hierarchy breaks down, and we've only explored a little of what that means. To continue exploring, we need to consider live coding as a practice that's transcended the paradigm of 'coding'. That, or it's fundamentally expanded what 'coding' means, and what 'coding' can be.

References

Armitage, Jack, and Andrew McPherson. 2017. "The Stenophone: Live Coding on a Chorded Keyboard with Continuous Control." In *Proceedings of the International Conference on Live Coding*. Morelia, Mexico. https://iclc.toplap.org/2017/cameraReady/stenophone_camready.pdf.

Blackwell, Alan F. 2015. "Patterns of User Experience in Performance Programming." In *Proceedings of the First International Conference on Live Coding*, 12–22. Leeds: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.19315.

Blackwell, Alan F., and Sam Aaron. 2015. "Craft Practices of Live Coding Language Design." In *Proceedings of the First International Conference on Live Coding*, 41–52. Leeds: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.19318.

Blackwell, Alan F., Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual*. Cambridge, MA: MIT Press. https://doi.org/10.7551/mitpress/13770.001.0001.

Blackwell, Alan F., and Nick Collins. 2005. "The Programming Language as a Musical Instrument." In *Proceedings of PPIG 17*, edited by P. Romero, J. Good, E. Acosta Chaparro, and S. Bryant, 120–30. Sussex University. https://ppig.org/papers/2005-ppig-17th-blackwell/.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (3): 321–30. https://doi.org/10.1017/S135577180300030X.

Diapoulis, Georgios. 2023. "Musical Live Coding in Relation to Interactivity Variations." *Organised Sound* 28 (2): 149–61. https://doi.org/10.1017/S1355771823000444.

Diapoulis, Georgios, and Palle Dahlstedt. 2021. "An Analytical Framework for Musical Live Coding Systems Based on Gestural Interactions in Performance Practices." In *Proceedings of the International Conference on Live Coding*. Valdivia, Chile. https://doi.org/10.5281/zenodo.5801942.

Diapoulis, Georgios, and Palle Dahlstedt. 2021. "The Creative Act of Live Coding Practice in Music Performance." In *PPIG 2021: 32nd Annual Workshop, Doctoral Consortium*. https://www.ppig.org/papers/2021-ppig-32nd-dc-diapoulis/.

Dobrian, Christopher, and Daniel Koppelman. 2006. "The 'E' in NIME: Musical Expression with New Computer Interfaces." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, 277–82. Paris, France. https://doi.org/10.5281/zenodo.1176893.

Fiebrink, Rebecca, Ge Wang, and Perry R. Cook. 2007. "Don't Forget the Laptop: Using Native Input Capabilities for Expressive Musical Control." In *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*, 164–67. New York: ACM. https://doi.org/10.1145/1279740.1279771.

GNU Project. n.d. "GNU Emacs." https://www.gnu.org/software/emacs/. Accessed May 15, 2026.

Helix contributors. n.d. "Helix." https://helix-editor.com/. Accessed May 15, 2026.

i3 contributors. n.d. "i3: Improved Tiling X11 Window Manager." https://i3wm.org/. Accessed May 15, 2026.

Kakoune contributors. n.d. "Kakoune." https://kakoune.org/. Accessed May 15, 2026.

Kiefer, Chris, and Thor Magnusson. 2019. "Live Coding Machine Learning and Machine Listening: A Survey on the Design of Languages and Environments for Live Coding." In *Proceedings of the Fourth International Conference on Live Coding*, 353–58. Madrid: Medialab Prado / Madrid Destino. https://doi.org/10.5281/zenodo.3946188.

Kirkbride, Ryan Philip. 2020. *Collaborative Interfaces for Ensemble Live Coding Performance*. PhD diss., University of Leeds. https://etheses.whiterose.ac.uk/id/eprint/28901/.

Moolenaar, Bram, and Vim contributors. 2026. "Vim Reference Manual: change.txt." https://vimhelp.org/change.txt.html#CTRL-A. Accessed May 15, 2026.

Parkinson, Adam, and Renick Bell. 2015. "Deadmau5, Derek Bailey, and the Laptop Instrument – Improvi-

sation, Composition, and Liveness in Live Coding." In *Proceedings of the First International Conference on Live Coding*, 170–78. Leeds, UK: ICSRiM, University of Leeds. https://doi.org/10.5281/zenodo.19350.

Roberts, Charlie, and Graham Wakefield. 2018. "Tensions and Techniques in Live Coding Performance." In *The Oxford Handbook of Algorithmic Music*, edited by Roger T. Dean and Alex McLean, 293–318. Oxford: Oxford University Press. https://doi.org/10.1093/oxfordhb/9780190226992.013.20.

Roos, Felix, and Alex McLean. 2022. "Strudel: Algorithmic Patterns for the Web." In *Proceedings of the Web Audio Conference*. Cannes, France. https://doi.org/10.5281/zenodo.6768844.

Salazar, Spencer. 2017. "Searching for Gesture and Embodiment in Live Coding." In *Proceedings of the International Conference on Live Coding*. Morelia, Mexico. https://iclc.toplap.org/2017/cameraReady/SpencerSalazar_LongPaper-SearchingForGestureAndEmbodimentInLiveCoding-final.pdf.

Strudel contributors. n.d. "Strudel Source Repository." Codeberg. https://codeberg.org/uzu/strudel. Accessed May 15, 2026.

suckless.org. n.d. "dwm: Dynamic Window Manager." https://dwm.suckless.org/. Accessed May 15, 2026.

TidalCycles contributors. n.d. "TidalCycles for VSCode." GitHub. https://github.com/tidalcycles/vscode-tidalcycles. Accessed May 15, 2026.

TOPLAP. 2004. "Manifesto Draft." https://toplap.org/wiki/ManifestoDraft. Accessed May 15, 2026.

Wang, Ge. 2008. "The ChucK Audio Programming Language: A Strongly-Timed and On-the-Fly Environ/mentality." PhD diss., Princeton University. https://www.cs.princeton.edu/~gewang/thesis.html.

xmonad contributors. n.d. "xmonad." https://xmonad.org/. Accessed May 15, 2026.