SHARP: Supporting Exploration and Rapid State Navigation in Live Coding Music

Douglas Bowman Jr.
School of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA
drewb00@vt.edu

Daniel Manesh
School of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA
danielmanesh@vt.edu

Sang Won Lee
School of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA
sangwonlee@vt.edu

ABSTRACT

How do live coders simultaneously develop new creations and master previous ones? Using findings drawn from previous studies about exploratory programming and our experience practicing live coding, we identified a need to support creation and mastery in the live coding space — specifically in the realm of live coding music We developed a tool, SHARP, which attempts to empower live coders in both their exploration and performances. SHARP is a code editor extension that visualizes the history of each instrument that the live coder creates; the visualization can then be used to revisit the previous states of the algorithm and create new ones from it. We believe that this extension will support live coders' exploration in practice as well as enable novel musical aesthetics in performance contexts. We did an initial evaluation of SHARP using an auto-ethnographic approach where one researcher used the tool over multiple sessions to compose a piece. From our reflection, we saw that SHARP supported composition by making it easier to explore different musical ideas and revisit past states. Our analysis also hints at new possible features, such as being able to combine multiple previous states using SHARP.

Author Keywords

live coding, exploratory programming, creative support tools

CCS Concepts

•**Applied computing** → **Sound and music computing;** *Performing arts;* •**Human-centered computing** → Graph drawings;

1. INTRODUCTION

As the applications of computer programming have continued to expand and diversify, one particularly fascinating area of growth is live coding, which is any process of creating things such as music or graphics through code that is done live, usually in front of an audience [5, 1]. An example is a coding demo that is done live in front of an audience where graphics or music is created through code on the spot. Live coding in front of an audience can be time-consuming because precision is needed in every line of code and the time it takes to craft an algorithm is not as immediate as a gesture-based musical instrument [11].

Furthermore, while live coders perform, they need to keep track of multiple sounds that are being played at that moment and the code associated with each sound. However, due to the delay it takes for someone to write an algorithmic pattern, the sound and code may not necessarily match; the code that generated what is being played may not be on the screen anymore. Swift et al. labeled this as a discrepancy between the code that associated with the currently generated sound, *the State of World* (SoW), and the code that is currently on the screen, *the State of Code* (SoC), in their work [12]. It is important that a live coder keep track of the SoC and SoW during a performance; however, maintaining a mental map of each instrument's SoC and SoW in performance can be a tall task for a live coder.

When live coders are practicing, they are often exploring new sounds, techniques, instruments, and effects—trying to create something original or novel [11]. In live coding, *exploratory programming* is the process of live coding that involves experimentation to create something new. Exploratory programming in live coding can involve code that constitutes complex musical styles, i.e., adding or taking away instruments, adding effects, changing pitch or tempo. Live coders use certain techniques while practicing to remember their exploration, such as saving the resulting code or a snapshot of the code or taking a screen recording [9]. However, creating a variation of some code usually means overwriting previous versions of that code, therefore a live coder will lose the SoC unless you want to duplicate the code for every variation. These difficulties in saving exploratory programming make reusing code for future practices or performances more difficult.

Our new tool, SHARP(State-History Augmentation for Rapid Programming) aims to alleviate some of these pressures and allow live coders more freedom to experiment in and out of performances. It is built as an add-on package to the Atom (and Pulsar) editor and is specifically designed for the TidalCycles (or Tidal for short) live-coding language [10]. SHARP is a tool to support exploratory programming and to facilitate fast navigation between code states in a live-coding environment. SHARP is a code editor extension that visualizes the history of each instrument that the live coder creates; the visualization can then be used to revisit the previous states of the algorithm and create new ones from it. We believe that this extension will support live coders' exploration in practice as well as enable novel musical aesthetics in performance contexts. With SHARP, live

NIME'23, 31 May–2 June, 2023, Mexico City, Mexico.

coders will be able to freely explore without fear of losing any of their explorations and quickly move between states for more robust performances.

2. RELATED WORK

Researchers have worked on developing various tools that can support exploratory programming in general programming context [2]. Kery et al. [7][8] previously explored the improvisational ways that programmers perform informal version control on their exploratory code, such as commenting out old code or creating new files. Their suggested system, Variolite, explored local version control and used a version graph for programmers to have multiple versions of smaller chunks of code (e.g., multiple versions of one helper function). While the graph version control system in Variolite is similar to our proposed tool for live coding, SHARP focuses more on the needs of live coders: having to quickly navigate and understand the branch history in temporal dimensions. Specifically, our representation provides rapid ways to preview and navigate states and automatically create version histories for each instrument based on their execution history.

Practicing live coding can be a challenging task for a live coder. Nilson [11] reflected upon how live coders should practice their music. He suggested that only so much could be achieved through live coding performances starting from scratch, and that practice was a vital part of expanding what live coding could accomplish. Nilson's ideas pushed us to create a state-history management system that not only supports rapid changes needed for performance but also aids practice. Lee and Essl have created a real-time text recording in a text editor for the sake of reproducing a particular performance asynchronously [9]. SHARP differs from the previous work in that it maintains history per each track and does not store the keystroke-level data.

Swift et al. [12] have researched the idea of a distinction between the SoC and SoW. They argue that an automatic updating of the SoW to match the changing SoC is usually undesirable since it means any intermediate SoC, while being moved from a previous state to a new state, will also be executed. Swift et al., therefore, argue for manual control over code evaluation and execution, even if this results in a divergence of SoC and SoW. The authors attempt to create tools to help the live coder with this potential divergence. SHARP aims to address this concern by notifying the live coder when a split between a sound's SoC and SoW occurs.

3. SHARP SYSTEM OVERVIEW

SHARP aims to improve the exploratory process of the live coder by handling state-history management for them, which enables the live coder to focus more on exploration and rapid state changes. SHARP improves this process through its per-instrument version tree, its SoC indicator, and its node-tagging feature. We will now dive into these features in detail. In the following sections, state-history node and variation are used interchangeably.

3.1 Run code to create a version tree

When a user runs Tidal code, SHARP will create or update a state-history box corresponding to that Tidal pattern (i.e., instrument). These state-history boxes contain nodes representing different points in the history of the state of that block.

Nodes, as represented by a circle, will be automatically created, and a history tree is built as you continually run a pattern with differing elements. This tree lets the live coder focus on the piece without remembering the previous code state and the difference between the two code states. Nodes are automatically created and extended from the currently selected state. Branching nodes can be created by selecting the node, or variation, you want to branch from and running a new version of the code block (Figures 1 and 2). The temporal order of execution is also preserved based on how far a node is from the initial node; the rightmost node will be the most recent code state executed.

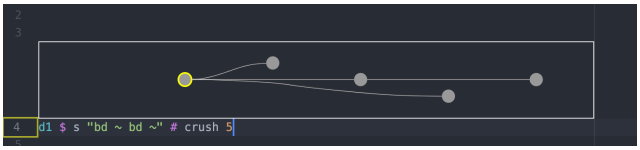

Figure 1: To branch from the first variation, first select it.

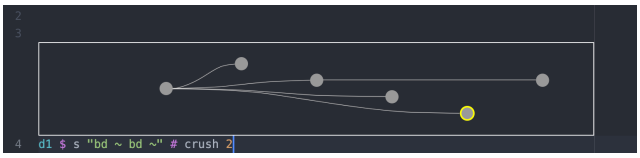

Figure 2: Once the variation you want to branch from is selected, you can run the code with the branching text to create a new branching node.

Hovering the mouse over a node will preview how the block of code below will change if the hovered-over variation is selected. Clicking that variation will permanently change that block of code to the selected state (Figure 3).

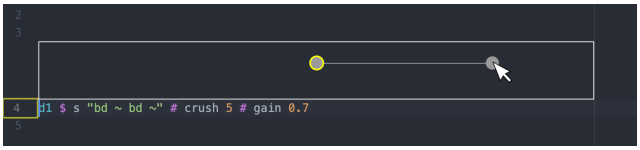

Figure 3: Hovering over a state-history node preview that state's text in the editor over top of the previous text.

3.2 Keep Track of SoC and SoW

Whenever a live coder modifies code after executing a line of code, there will be a difference between SoC and SoW. We created an indicator on the line number, which lights up when the SoC for a specific pattern differs from the SoW, which is code that generates sounds at the moment as part of the live-coding piece (Figure 4). In this way, the live coder always knows if the code in the editor for a pattern represents the code currently running or a variation thereof.

3.3 Explore Freely with State Navigation

The pressure of live-coding performances can often discourage exploratory programming and confine performers to previously practiced methods and sounds. Furthermore, exploratory programming in live-coding practices can be hindered by a lack of formal versioning and an inability to save sounds you want to perform later.

SHARP's state-history management and version trees enable live coders to freely explore pieces they are developing or performing by ensuring that all states of each instrument are saved and inserted into the version tree for quick access.

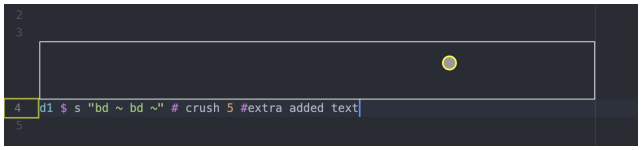

Figure 4: The yellow bordered container on line number 4 is lit here because the extra text has been typed since the last execution of the block, indicating a disparity between the SoC and SoW.

Live coders are then freed to focus on exploration instead of keeping track of all the potential states they may want to reuse.

Additionally, a node-tagging feature allows live coders to mark special states they are likely to refer back to with a custom color. This is useful for exploration and performance to make important states easy to locate, especially in a large tree with many branches containing multiple nodes (Figure 5).

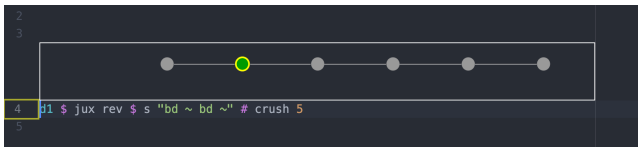

Figure 5: A state-history node tagged in green.

3.4 Implementation Details

SHARP is implemented in JavaScript and CSS, using the Atom editor package framework, and it relies on the existing TidalCycles package. Atom is an open-source code editor developed by GitHub, and it is designed to be very flexible by offering a package system to manipulate the editor to your needs. SHARP is meant to be used with TidalCycles, a live-coding language written in Haskell [10], and is currently not compatible with but certainly expandable to other live-coding languages.

4. INITIAL EVALUATION

4.1 Methodology

To evaluate SHARP, we took an auto-ethnographic approach [6] where one of the authors used SHARP every day over the course of 10 days and kept a diary of their reflections. The researcher had a goal of developing a 10-minute piece which they recorded via screen capture at the end of the study. The live coding sessions lasted a minimum of 30 minutes each. Afterward, we analyzed the diary entries for emergent themes.

While autoethnography does not necessarily yield generalizable results, it is still considered a useful starting point in HCI research, as it is a way to get in-depth insights [3]. Within the NIME community, it has been argued that autoethnography is a valuable evaluation tool for the so-called "researcher-practitioner" [4]. Because of the limited scope and duration of our diary study, we consider these results a preliminary stepping stone that will help orient our future work on the project.

The researcher in question had some basic experience with live coding using TidalCycles before the experiment, but they do not regularly perform live coding. Otherwise, they have a background both in software engineering and in music performance, primarily as a classical pianist.

4.2 Results

4.2.1 Facilitating Exploration

The researcher felt that SHARP proved beneficial for easily trying out different code. During the first few sessions, the researcher noticed they were hesitant to alter or delete code – without SHARP, they would have made a copy of the existing code block and edited the copy instead, preserving the older version. They felt that using SHARP was better, both because it was faster and it reduced clutter in the text editor.

On one occasion, the researcher had two different variations of a code block and wanted to experiment with how they could smoothly transition from one to the other. Using SHARP, the researcher tagged both variations to keep track of them, and so they could quickly switch back and forth between them and other new things they tried. On the other hand, they found that things got slightly confusing because, with SHARP, the order in which the variations were laid out in the version tree was the order in which they were explored during practice but not necessarily the order the researcher wanted them to unfold.

Finally, although SHARP is meant to add to the version tree every time a code block is run, for the duration of the study, there was a bug where sending the code to SHARP and sending the code to Tidal required two different commands. While this was somewhat inconvenient, the researcher sometimes found themselves purposefully running code without sending it to SHARP. This allowed them to compare several different numerical parameters to a function quickly without adding unnecessary clutter to SHARP's version tree. In practice, live coders may want to maintain the version history tree selectively based on their needs.

4.2.2 Changing Coding Strategies

The researcher reflected that using SHARP influenced how they went about live coding. As previously mentioned, one change was that they could rely on SHARP for versioning, and did not have to fall back to ad-hoc versioning methods like keeping multiple versions of a block of code around in the editor or undoing multiple times to restore the previous state of the code. In this case, the use of SHARP simply replaced an old coding behavior.

Beyond just simplifying the way they coded, the researcher realized they had adapted the way they coded in order to accommodate SHARP. For example, at the start of the study, they noticed they were combining several musical ideas into one block of code, i.e., one code block was responsible for a rhythm section, but there were three different types of sound (or 'instruments') defined in the code block. They found that it was better to break up the code into separate blocks so that they could use SHARP to keep track of the versions for each of the three rhythmic instruments separately. Because SHARP only does versioning at the code-block level, SHARP encouraged them to decompose code into modules.

4.2.3 Composition and Performance

While there are many ways to go about composing a live coding piece in terms of the improvisation level, the researcher opted for a planned approach where they created a blueprint for the code which would be executed and the or-

der it would be executed in. In one of the first few sessions, the researcher went back over the code they had written and tagged all the versions they thought might be good to use in their piece. Then, they copy-pasted the tagged versions into a separate file that had all the code changes written out in sequence. They reflected that it would be nice for SHARP to have a way to export tagged versions automatically, but also considered that there was no way to indicate what order they should go in. The researcher imagined one possible interaction where the versions could be used as a palette, and a sequence of versions could be assembled by dragging versions onto a timeline.

As part of their composition, the researcher wanted to combine several previous versions into one. Specifically, they wanted to write code that randomly alternated between running three different code blocks in SHARP's version tree. Using SHARP, this involved clicking on each node, copying most of the code, and then pasting it into a list, which could then be passed to another function in Tidal. While SHARP did make it easy to switch between nodes, the process was still somewhat slow, making it a costly operation for a performance setting.

Finally, as part of their composition, the researcher planned out a piece that involved periodically returning to the same section, i.e., program state. The piece's form was roughly ABACADA. The researcher found that using SHARP was particularly well-suited for quickly making the switch back to the 'A' section, which just involved clicking a tagged node on the version tree. The researcher's performance recording is submitted as supplementary material.

4.3 Future Considerations

From our preliminary study, SHARP appears to support exploration in a live coding setting. But what comes next after that exploration? SHARP does not currently offer any specific functionality to aid in synthesizing the results from a live coding practice session into a performance, a set of notes summarizing the session, or something similar. More work is needed to determine what types of functionality along these lines live coders would want, if any.

The preliminary study also points to possible new affordances for SHARP. For example, SHARP could offer a way to speed up the process of combining multiple previous program states. Additionally, SHARP could add a feature to help live coders try out running their code in different orderings. It is likely that running a study with multiple live coders may uncover the need for even more new features along these lines.

5. CONCLUSIONS

Live coders face the pressures of an audience during a performance and the pressures of maintaining mental maps of their exploration during practice. SHARP aims to aid the live coder in either scenario—helping them rapidly and efficiently manage a complex set of not necessarily linear exploratory ideas and keeping them privy to the SoW and SoC for each of their instruments. SHARP's ideas, while contained to TidalCycles for this research, can be expanded to any language for musical live coding. These ideas may also be useful for other types of live coding, such as live coding graphics.

Through our preliminary autoethnographic study, we have seen that SHARP can aid in code exploration and enable useful code switching affordances for performances. We also uncovered several possible new features for SHARP, such as the ability to merge multiple program states together into something new. In the future, we hope to run a larger study to get a better understanding of how SHARP can fit into the general live coding practice.

6. ETHICS STATEMENT

Our research was conducted under our university's computer science department. This work was not supported by any funding sources. No one was involved in the development and validation other than the authors themselves. The SHARP code is publicly available as open source at `https://web.pulsar-edit.dev/packages/tidal-sharp`.

7. REFERENCES

[1] U. Attanayake, B. Swift, H. Gardner, and A. Sorensen. Disruption and creativity in live coding. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5. IEEE, 2020.

[2] M. Beth Kery and B. A. Myers. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 25–29, 2017.

[3] A. Blandford, D. Furniss, and S. Makri. Qualitative hci research: Going behind the scenes. *Synthesis lectures on human-centered informatics*, 9(1):1–115, 2016.

[4] B. Carey and A. Johnston. Reflection on action in nime research: Two complementary perspectives. 07 2016.

[5] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003.

[6] N. K. Denzin and Y. S. Lincoln. *The Sage handbook of qualitative research*. sage, 2011.

[7] M. B. Kery, A. Horvath, and B. A. Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3–025, 2017.

[8] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[9] S. W. Lee and G. Essl. Live writing: Asynchronous playback of live coding and writing. In *Proceedings of International Conference on Live Coding*, Leeds, United Kingdom, 2015.

[10] A. McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70, 2014.

[11] C. Nilson. Live coding practice. In *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*, NIME '07, page 112–117, New York, NY, USA, 2007. Association for Computing Machinery.

[12] B. Swift, A. Sorensen, H. Gardner, and J. Hosking. Visual code annotations for cyberphysical programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 27–30. IEEE, 2013.