Beyond Editing: Extended Interaction with Textual Code Fragments

Charles Roberts
Media Arts & Technology Program
charlie@charlie-roberts.com

Matthew Wright
Media Arts & Technology Program
matt@create.ucsb.edu

JoAnn Kuchera-Morin
Media Arts & Technology Program
jkm@create.ucsb.edu

ABSTRACT

We describe research extending the interactive affordances of textual code fragments in creative coding environments. In particular we examine the potential of source code both to display the state of running processes and also to alter state using means other than traditional text editing. In contrast to previous research that has focused on the inclusion of additional interactive widgets inside or alongside text editors, our research adds a parsing stage to the runtime evaluation of code fragments and imparts additional interactive capabilities on the source code itself. After implementing various techniques in the creative coding environment *Gibber*, we evaluate our research through a survey on the various methods of visual feedback provided by our research. In addition to results quantifying preferences for certain techniques over others, we found near unanimous support among survey respondents for including similar techniques in other live coding environments.

Author Keywords

live coding, visualization, text editors, state, human-computer interaction

ACM Classification

H.5.5 [Information Interfaces and Presentation] Sound and Music Computing – Methodologies and techniques, D.2.3 [Software Engineering] Coding Tools and Techniques – Program editors, H.5.2 [Information Interfaces and Presentation] User Interfaces – Screen design

1. INTRODUCTION

We posit that, when engaging in creative coding, it is preferable to do so in a code editor that itself provides creative affordances. This is particularly true in *live coding*, where performers code audiovisual works on stage and often project their programs for audience members to see [1, 7]. For many performers, projecting source code is a critical component of the performance, as it reveals the activity taking place and potentially gives insight into the algorithmic processes at work. As the audience is typically watching the performer's text editor for a significant part of any given performance, the editor itself is an important performative element. In many live coding environments that support graphical performances, text editors are composited on top of the graphical output generated by the code performers write (examples include Fluxus [3], Lich.js [9], and LivecodeLab [2]), enabling performers and audience members to easily view both source code and generated visuals concurrently.

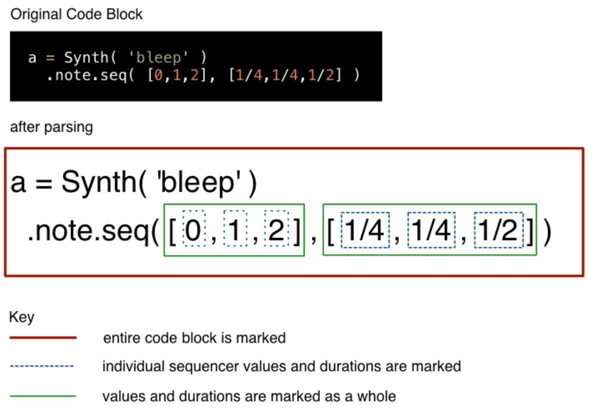
The research described here specifically examines how interactive affordances, including techniques for manipulating state and graphical annotations providing both continuous and discrete feedback, can be added to source code fragments in textual programming environments. As one constraint on our research, we use the source code of performances as a starting point for this line of inquiry as opposed to additional graphical widgets. However, a number of active research projects have examined the use of graphical widgets for these purposes in conjunction with text editors. For example, Lee and Essl provided a separate viewing area to display system state alongside the text editor in urMus [4], while Swift et al. used graphical widgets overlaid on top of the Emacs editor to display aspects of state [17]. As a second constraint, we limit our discussion to textual programming as opposed to visual programming languages.

The research presented here extends the browser-based creative coding environment *Gibber* [12]. After documenting the various interactive affordances we have added to Gibber's code fragments we will conclude with results of a survey examining the impact of the visual annotations we describe. Related work will be discussed in the context of the research we present.

2. METHODOLOGY

There are various techniques for executing code during live coding performances. Some live coding platforms, such as LuaAV [19], ChucK [20], and LiveCodeLab [2], allow users to seamlessly replace parts of running programs with new code, and can include mechanisms for managing state across versions. Other platforms, such as *Impromptu* [14] and SuperCollider [8], allow execution of individual lines or blocks of code. Gibber uses the second model, providing keystroke commands for executing an individual line of code, the block enclosing the current cursor position, or selected blocks of code.

For purposes of this research we introduced an extra parsing stage into Gibber. Gibber uses the syntax tree generated by this parser to assign code that calls constructors to the objects they create. More specifically, this parser gives the object a property named `text` containing the associated source code (e.g., invocation of a constructor) that created the object. For example, evaluating the code fragment `mysynth = Synth()` results in an object named `mysynth` with a property named `text` whose value is the string 'mysynth = Synth()'. The position of the code fragment in the text editor is also marked so that it can be programatically found


Original Code Block
after parsing
Key
entire code block is marked
individual sequencer values and durations are marked
values and durations are marked as a whole

```
a = Synth( 'bleep' )
  .note.seq( [0,1,2], [1/4,1/4,1/2] )
```

```
a = Synth( 'bleep' )
  .note.seq( [ 0 , 1 , 2 ] , [ 1/4 , 1/4 , 1/2 ] )
```

Figure 1: After source code is parsed, markup is added to textual components of constructor calls so that each component can be easily referenced.

in the future as needed; this is an important component of creating the continuous representations of state described in Section 3.3.2.

Gibber also looks for the creation of musical sequences in code blocks. A sequence in Gibber consists of a list of *values* and a list of *durations* defining the output messages and scheduling of the sequencer. Each element of these lists is marked so that its position in the text editor can easily be found no matter how code shifts during the editing of a program. The markers enable many of the annotations that will be described shortly.

Figure 1 shows how the text of a representative code fragment (creating a `Synth` object and sequencing its `note` method) is annotated upon evaluation.

3. INTERACTIVE AFFORDANCES FOR CODE FRAGMENTS

Providing audiovisual objects information about the source code used to create them enables a variety of annotations and interactive affordances. In this section we discuss the capabilities we have added to Gibber to date, and, when appropriate, also discuss previous research informing their development.

3.1 Editing Literals via Mouse Movements

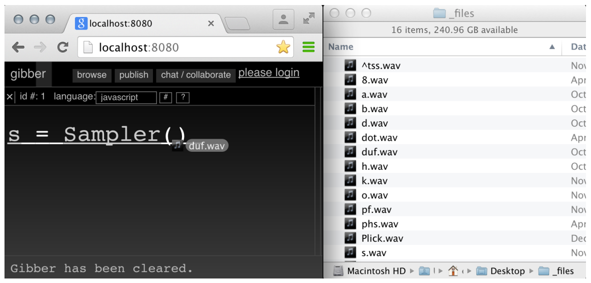
Using an idea adapted from the essays of Bret Victor [18], this augmentation turns literals (e.g., numbers) in calls to constructors and in sequencing calls into interactive sliders. Consider the two lines of code below:

```
mysine = Sine( 440, .25 )  // freq, amp
mysynth = Synth().note.seq( [ 0, 1, 2, 3 ], 1/4 )
```

In the first, the frequency argument (440) and the amplitude argument (.25) both become interactive widgets. Clicking on the literals and dragging horizontally will modify their text and concurrently modify the `frequency` or `amp` property of the `mysine` object. In a similar fashion, the arguments to the `note.seq` method will also be converted into interactive widgets when the last line is evaluated. Modifying these values via mouse interaction does not immediately affect the `mysynth` object, but instead changes the internal elements of the `values` and `durations` lists used by the object's sequencer.

When sequencing calls to the `note` method in Gibber, users may pass either numbers or strings representing note name and octave (such as 'c5', 'db4', 'd#2'). When these literals are turned into interactive elements, they advance through the chromatic scale. Thus, clicking on 'c5' and dragging to the right would yield values of 'db5', 'd5', 'eb5' etc. By default accidentals displayed are flats.

There is metadata in Gibber for every audiovisual property, describing a canonical range of values to be used and defining whether each property's output is perceived linearly or logarithmically. This metadata is used to both constrain the minimum and maximum values available to mouse gestures and to map the output curve of mouse movements to a linear or logarithmic scale as appropriate.



```
s = Sampler()
```

```
Gibber has been cleared.
```

^tss.wav
8.wav
a.wav
b.wav
d.wav
dot.wav
duf.wav
h.wav
k.wav
o.wav
pf.wav
phs.wav
Plick.wav
s.wav

Figure 2: The Sampler object's code fragment is underlined when a user has correctly dragged a resource above it that can be loaded if it is dropped.

3.2 Drag and Drop Operations for Resource Loading

Creative coding practice often takes advantage of audiovisual resources that exist as files on a programmer's computer. Importing these can be tedious, as the paths to individual files often must be manually entered by the programmer / performer; this is potentially problematic in the course of a live coding performance.

Our solution allows researchers to drag resource files directly onto certain code fragments that support the notion of loading files, automatically loading and presenting the file in an appropriate way. For example, if a `Sampler` object is playing back an audio file, dragging a different audio file onto its constructor's code fragment will immediately load and trigger playback of the new file and stop playback of the old one. Similarly, if a 3D model is presented on the screen, dragging a new model onto the code fragment that generated the object will immediately replace the existing model in the scene graph with the new resource. The code fragment is underlined when a file is correctly positioned above it to perform a drop, as shown in Figure 2.

The simplicity of loading files via drag and drop onto code fragments (as opposed to manually typing individual file paths) yielded inspiration for new functionalities in Gibber objects that rely on external file resources. When testing the added drag and drop interaction, we found ourselves wanting to load multiple files concurrently, with affordances for easily selecting the file currently used for audiovisual display. This would enable, for example, rapidly interspersing different audio files into a musical performance, or quickly applying randomized combinations of generative fragment shaders to a graphical scene. Accordingly, in Gibber users can drag single files, multiple files, or entire directories of files onto code fragments to load them all at once. Methods are provided to select which audiovisual resource is currently displayed by an object, including one that chooses a resource at random. Loading an audio Sampler with dozens of files and then randomly triggering their playback to form rhythmic patterns has yielded satisfying musical results that have impressed Gibber users.

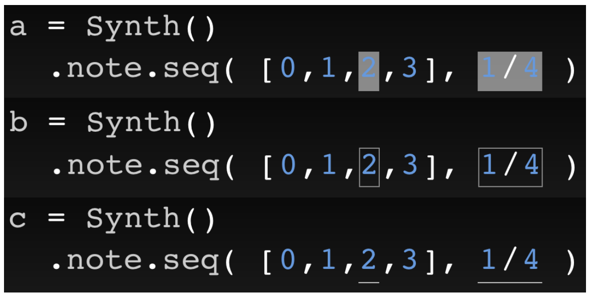
3.3 Continuous Display of System State

Performers capitalize on the visual aspect of live coding in different ways. Code comments can send messages to audience members. The naming of variables and methods can convey both humor and functionality. In many of performances by Andrew Sorensen using his Impromptu software, various code fragments are highlighted as the cursor moves between them to indicate scope; in our opinion this provides a dramatic sense of movement for the audience in addition to pragmatic information for the performer. As mentioned in the introduction, recent additions to both Impromptu and another environment authored by Sorensen, *Extempore* [16], have introduced graphical annotations to code that link the “State of the World” to the “State of the Code” [17]; they provide affordances to the code editor for graphically displaying the state of system output in conjunction with the code itself. In this paper, we describe techniques that use the font characteristics and background of code (as opposed to additional graphical widgets such as those found in [17]) to display insights on system state with the hope of providing both informative and performative affordances to Gibber.

3.3.1 Sequencer Phase and Output

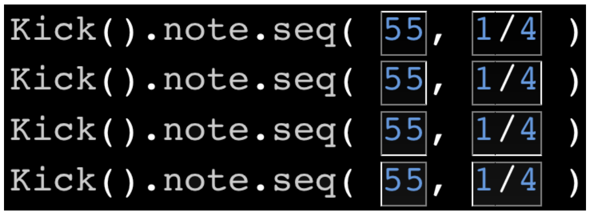
The use of sequencer objects in Gibber is the most common way of creating repeating patterns. As mentioned in Section 2, all sequencer objects in Gibber consist of two underlying lists of data, one determining timing and the other determining output values. These lists are wrapped in functions; by default, each time the function is called it outputs the next element of its corresponding list. Any time these functions select and output an element, Gibber visually highlights the corresponding text in the source code, revealing both the value that is outputted as well as the timing of when the output occurs. A number of different methods for highlighting the source code text corresponding to the output are provided, including flashing the background of the text representation, creating a border around it, and placing an underscore beneath it; the initial methods we created are shown in Figure 3. In a survey conducted with creative coders (discussed more thoroughly in Section 4), a strong preference was indicated for using a border to highlight as opposed to flashing or underscores; however, a number of commentors also suggested a combination of flashing and borders would provide the best results. We agree with this assessment; the borders do not obscure code while the flashing provides a more urgent sense of time. Accordingly, we implemented this combination and made it the default method of revealing phase and timing. Additionally, if the same output is triggered multiple times in succession individual sides of the border are highlighted in a clockwise pattern to indicate repetition. This clockwise progression also shows how many times the repetition has occurred (modulo 4). Fig. 4 shows a snapshot of this animation effect. Users can select which annotation they would like to use and further manipulate various properties such as the colors used by the notation or the duration of flashes.

Although by default sequencers advance linearly through their values and durations lists, elements can also be randomly selected for every output. The step size for each list can also be changed, enabling sequencers to skip elements when moving through a given list, and also to move in reverse. Highlighting the current elements output by the sequencer makes these processes more transparent. But even when sequencers are advancing linearly it can be useful to



```
a = Synth()
  .note.seq( [0,1,2,3], 1/4 )
b = Synth()
  .note.seq( [0,1,2,3], 1/4 )
c = Synth()
  .note.seq( [0,1,2,3], 1/4 )
```

Figure 3: Three modes of highlighting sequencer phase in Gibber: a) flashing, b) border, and c) underscore. Survey participants expressed a strong preference for the bordered indication.



```
Kick().note.seq( 55, 1/4 )
Kick().note.seq( 55, 1/4 )
Kick().note.seq( 55, 1/4 )
Kick().note.seq( 55, 1/4 )
```

Figure 4: Repeated quarter note output of a kick drum, with the borders indicating whether the current phase of each sequence is on beat 1,2,3 or 4.

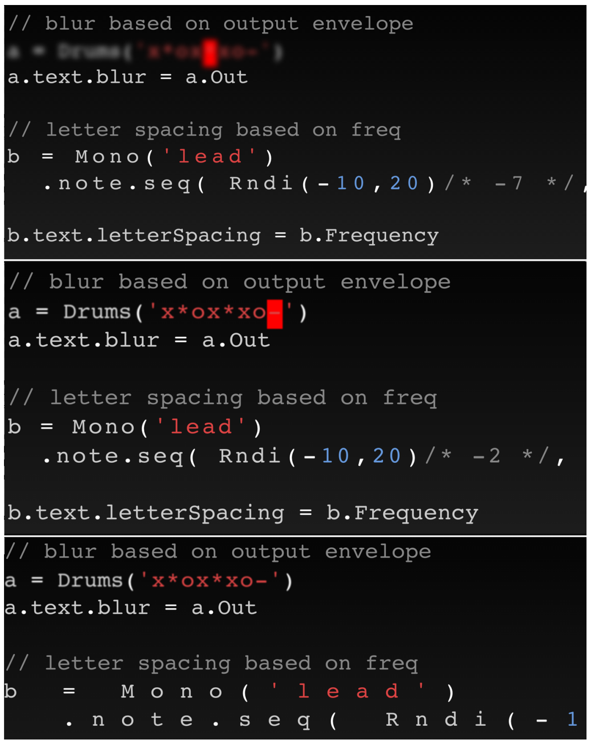
see which notes and durations are firing at any moment in time; this shows progression through each sequence and has the potential to increase audience understanding of events that are occurring.

Although we are unaware of any other textual programming environments that provide source code annotations for phase and timing, the visual programming environment *SchemeBricks*, by Dave Griffiths [11], also provides annotations visually indicating scheduling and control flow. In SchemeBricks, performers use drag and drop techniques to create modular lisp programs for generative music creation. The visual programming components (aka ‘bricks’), flash when signal travels through them; as such events often trigger audio output, the synchronization of the flash with the audio event helps show both performer and audience which part of the running program is responsible for each sound that is played.

3.3.2 Continuous Representation of Audiovisual Properties

Gibber provides a mapping abstraction that enables users to quickly create mappings across modalities [13]. Using simple assignment, programmers can create continuous mappings instead of instantaneous ones simply by capitalizing the name of the property in the righthand value of the assignment. The `text` property of each audiovisual object (discussed in Section 2) has various properties that can be assigned using this mapping abstraction. In effect, this enables font characteristics of code fragments to become continuous displays of audiovisual information. A few sample effects are shown in Figure 5.

The range of values used to determine each font characteristic is adjustable on a per-object basis. If a performer wants to create subtle shifts in font size based on the amplitude of a unit generator, they can simply lower the default maximum value for that property and raise the minimum to create a narrower range of activity. For example:



```
// blur based on output envelope
a = Drums('x*ox*xo-')
a.text.blur = a.Out

// letter spacing based on freq
b = Mono('lead')
  .note.seq( Rndi(-10,20)/* -7 */,

b.text.letterSpacing = b.Frequency
```

```
// blur based on output envelope
a = Drums('x*ox*xo-')
a.text.blur = a.Out

// letter spacing based on freq
b = Mono('lead')
  .note.seq( Rndi(-10,20)/* -2 */,

b.text.letterSpacing = b.Frequency
```

```
// blur based on output envelope
a = Drums('x*ox*xo-')
a.text.blur = a.Out

// letter spacing based on freq
b   =   M o n o ( ' l e a d ' )
    . n o t e . s e q (   R n d i ( - 1
```

Figure 5: Three frames showing text properties (blur and letter spacing) being controlled by audio parameters.

```
// font size is measured in ems
a = Drums('xoxo')
a.text.fontSize = a.Out
a.text.fontSize.min = .8
a.text.fontSize.max = 1.2
```

An open question is whether or not the effects generated are useful from either the perspective of the audience or the perspective of human-computer interaction. In the survey discussed in Section 5, we showed two examples of these effects in action. In the first example, the frequency of a monosynth controlled the font size of the code fragment that generated it. In the second, the output envelope of a drum loop was used to drive blurring of text. Reactions to these visual elements were varied, as shown by the responses of subject #8 and subject #9:

> Subject #8: 'Size change and blurring look like gimmicks rather than useful HCI features.'
>
> Subject #9: 'the blur effect looks dope.'

In our live coding practice, we have primarily used these notations with performative (as opposed to informative) intent. We do not feel they positively impact our understanding of the underlying processes at play in a given live coding session. However, we are intrigued by the rather dramatic visual effects they produce and the potential impact this could have on audience appreciation of live coding performance. Recent research by Lee and Essl [5] also explores this topic. In their work, text is rendered in a WebGL layer, imbuing its presentation with a variety of effects (via GLSL shaders) and 3D transformations. This provides more options for graphical manipulation of text than what is found in Gibber but currently removes some affordances for editing (such as syntax highlighting) typically included in code editors.

3.4 Updating Source Code to Display Data and Function Output

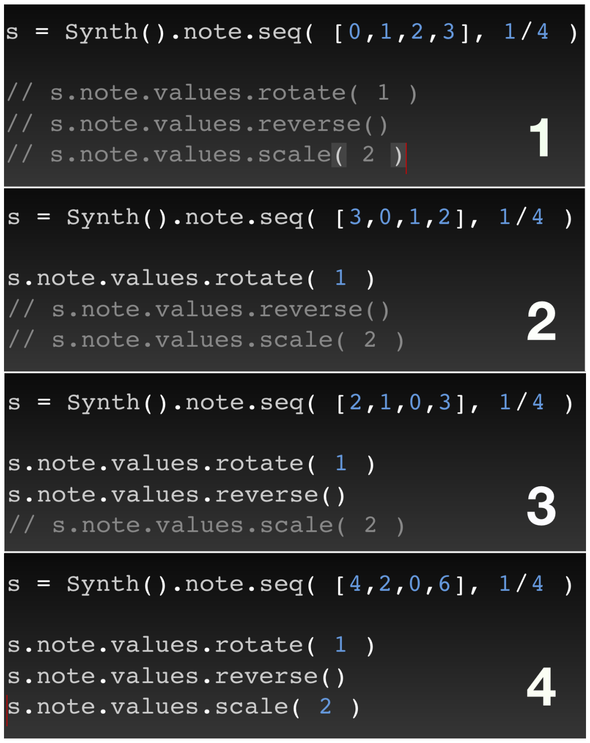
Gibber programmatically manipulates source code to display both the current values stored in pattern objects used by sequencers and the output of functions that are contained in these patterns. Patterns have a variety of methods for manipulating their contents, many of which are drawn from serialist techniques. In Figure 6 we show four frames of source code in Gibber. The last three frames show changes to the source code (as well as the underlying data structure) resulting from various pattern manipulations. This addition to Gibber was inspired by source code manipulation found in the live coding environment *ixi lang* [6], which also updates the source code of performances to reflect current values held in musical patterns.

In Section 3.3.1 we described how annotations are used to highlight elements that are selected for output by sequencer objects. In the case of literals, this highlighting is all that is required, as it provides both a temporal indication of when the output occurs as well as showing the value outputted. However, Gibber also enables functions to be members of patterns; whenever a sequencer selects a function from a pattern the function is evaluated to generate the final output. `Rndi` and `Rndf` are two examples of functions that are commonly used for this purpose; unlike their lowercase counterparts, `rndi` and `rndf`, which output numeric values, `Rndi` and `Rndf` output functions that generate random values each time they are called. Without extra consideration, the particular outputs of such functions would be opaque to audience members and performers.

With this in mind, we created two source code manipulations to display the output of any function embedded within a sequencer pattern. In the first, the original source code that generates or refers to the function member is replaced by its output and updated whenever the function is called. In the second, code comments are placed alongside the original source code; in this manner no code is ever hidden. You can see a comparison of these two methods in Figure 7. In our evaluation survey 71% of the participants indicated a preference for using comments to show the output of functions as opposed to changing the original source code.

All source code manipulations are removed when the Gibber engine is cleared (clearing removes all audiovisual objects from their respective graphs and stops output); in effect, the source code returns to its original state before any evaluation occurred. Thus, manipulations do not permanently alter the source code, enabling users to capture the original version so that performances can be re-created.

In addition to the ixi lang precursor, Alex McLean previously explored using source code comments to display state in his *feedback.pl* system [10, 21]. These comments both displayed underlying data structures and provided input mechanisms for changing them. Algorithms running in feedback.pl were also capable of modifying their source code; these modifications (which primarily consisted of changing the definition of variables due to the complexity of parsing Perl) were immediately evaluated to change the running program. In Gibber, source code modifications are not intended to be evaluated; ideally evaluation would produce no effect as the source code is always displaying current state. Instead the modifications are simply side effects for revealing algorithmic output and aspects of state that would otherwise remain hidden.


```
s = Synth().note.seq( [0,1,2,3], 1/4 )

// s.note.values.rotate( 1 )
// s.note.values.reverse()
// s.note.values.scale( 2 )
```
1

```
s = Synth().note.seq( [3,0,1,2], 1/4 )

s.note.values.rotate( 1 )
// s.note.values.reverse()
// s.note.values.scale( 2 )
```
2

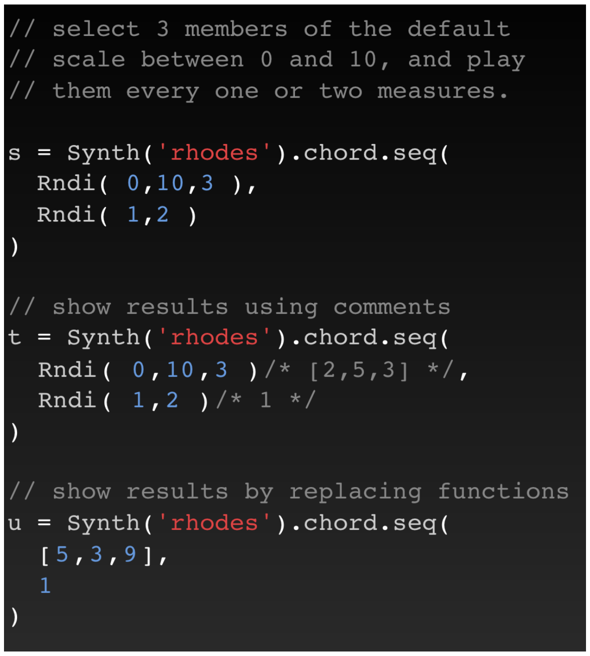
```
s = Synth().note.seq( [2,1,0,3], 1/4 )

s.note.values.rotate( 1 )
s.note.values.reverse()
// s.note.values.scale( 2 )
```
3

```
s = Synth().note.seq( [4,2,0,6], 1/4 )

s.note.values.rotate( 1 )
s.note.values.reverse()
s.note.values.scale( 2 )
```
4

Figure 6: Three list manipulations applied to the values (originally 0,1,2,3) of a sequencer assigned to a synth.


```
// select 3 members of the default
// scale between 0 and 10, and play
// them every one or two measures.

s = Synth('rhodes').chord.seq(
  Rndi( 0,10,3 ),
  Rndi( 1,2 )
)

// show results using comments
t = Synth('rhodes').chord.seq(
  Rndi( 0,10,3 )/* [2,5,3] */,
  Rndi( 1,2 )/* 1 */
)

// show results by replacing functions
u = Synth('rhodes').chord.seq(
  [5,3,9],
  1
)
```

Figure 7: Two methods of displaying function output as triggered by a sequencer.

4. EVALUATION

We conducted a web-based survey that collected basic information on demographics and programming experience, then asked opinions on videos demonstrating our various interactive affordances in action. The average time to complete the survey was just under ten minutes, and 102 responses were collected between January and February of 2015. The survey was directly emailed to the livecode mailing list[1], the Gibber mailing list[2] and also to students and composition faculty at UC Santa Barbara. The respondents were mostly males (93%), thirty to forty years of age (38%), and with over ten years of programming experience (56%). 75% of respondents had seen a live coding performance at some point in their life, and 67% had programmed in an environment that is typically used for live coding performance. 62% of respondents had experimented with Gibber in some capacity.

The survey was fairly simple; the primary intent was to expose subjects to the visual annotations described in this paper, collect data on their preferences, and provide a space for them to give comments and ideas for future work. Some of the results have been previously mentioned in Sections 3.4 and 3.3.1. After watching a video displaying the three methods of showing sequencer phase and output, 51% of respondents preferred the bordered indication, 26% preferred the flash indicator, and 11% preferred the underscore.

73% of respondents preferred visual indicators to use muted colors as opposed to bright ones. 82% of respondents preferred the use of code comments to indicate function output as compared to replacing the source code responsible for generating the function with its output values. Multiple subjects indicated that they wanted to be able to customize aspects of the notations to their personal preferences. Although it was not described in the survey, this capability is provided in Gibber and we plan on extending it in the future.

One particularly interesting result was that 96% of participants believed that other live coding environments could benefit from visualizations and notations similar to the ones displayed in the survey. This indicates a very promising research direction, and we look forward to future explorations. We are particularly interested in the effect of these notations on a more canonical ‘audience’, as opposed to the programmers and live coders who were the primary respondents to this survey, and are actively designing experiments with this goal in mind.

5. CONCLUSIONS AND FUTURE WORK

Our research extends Gibber with a variety of visual affordances for displaying both state and the output of functions. In addition, we added added interactive capabilities to source code fragments including manipulation of audiovisual properties and drag-and-drop loading of audiovisual resources.

We plan to continue our research on visual annotations while quantifying their effects for performers / programmers and audience members. Other research directions we are considering include:

- **Isolating animated sequences**. Sequences that are visually animated with source code changes yield a sense of dynamism that is comparatively lacking in static sections of source code. After watching a performance that used techniques described here, one audience member remarked that she became disappointed

[1]http://lurk.org/groups/livecode/
[2]http://lurk.org/groups/gibber/

when a shifting melodic motif was no longer visible in the editor due to scrolling; she was enjoying watching its evolution and associating the musical results with the changes in pattern she saw. Figuring out performance techniques to emphasize continuous display of patterns and algorithms could be one avenue for increasing audience enjoyment of performances.

- **Metaphor**. There are a variety of metaphors that could be used with the notational elements described here. As a simple example, consider tying a fade in amplitude of the master output bus to a fade in opacity of all source code text. We believe the use of metaphor could both aid audience understanding and provide a variety of dramatic effects.
- **End-user Design of Annotations**. Currently the system for defining annotations in Gibber is fairly opaque. How can we open exploration up to Gibber end-users? Providing control over existing notational properties is a good first step, but we believe we can do more in this area.
- **Annotations for Alternative Scheduling**. Although use of the sequencer object is currently Gibber's most idiomatic way of dealing with scheduling, Gibber does support other approaches of dealing with time, such as temporal recursion [15]. In a similar vein to providing end-user design of annotations, what abstractions will make our annotation system more generalizable so that it can accommodate these?

.

The results of our survey indicate a strong level of interest. In addition to the results previously reported, over 80% of respondents indicated that they believed the techniques described here were useful for performers, audience members, teachers, and/or students. The potential for illuminating algorithmic processes for all of these groups is promising, and we look forward to future experiments and future performances using these techniques.

6. ACKNOWLEDGMENTS

This research is graciously supported by a postdoctoral fellowship from the Robert W. Deutsch Foundation.

7. REFERENCES

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

[2] D. Della Casa and G. John. Livecodelab 2.0 and its language livecodelang. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 1–8. ACM, 2014.

[3] D. Griffiths. Fluxus. In *Collaboration and learning through live coding, Report from Dagstuhl Seminar*, volume 13382, pages 149–150, 2013.

[4] S. W. Lee and G. Essl. Communication, control, and state sharing in networked collaborative live coding. *Ann Arbor*, 1001:48109–2121, 2014.

[5] S. W. Lee and G. Essl. Web-based temporal typography for musical expression and performance. In *Proceedings of the New Interfaces of Musical Expression Conference*, 2015.

[6] T. Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of the International Computer Music Conference*. University of Huddersfield, 2011.

[7] T. Magnusson. Herding cats: Observing live coding in the wild. *Computer Music Journal*, 38(1):8–16, 2014.

[8] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[9] C. McKinney. Quick live coding collaboration in the web browser. In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*, pages 379–382, 2014.

[10] A. McLean. Hacking perl in nightclubs. `http://www.perl.com/pub/2004/08/31/livecode.html`, 2004.

[11] A. McLean, D. Griffiths, N. Collins, and G. Wiggins. Visualisation of live code. *Proceedings of Electronic Visualisation and the Arts 2010*, 2010.

[12] C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. *Proceedings of the International Computer Music Conference*, 2012.

[13] C. Roberts, M. Wright, J. Kuchera-Morin, and T. Höllerer. Gibber: Abstractions for creative multimedia programming. In *Proceedings of the ACM International Conference on Multimedia*, pages 67–76. ACM, 2014.

[14] A. Sorensen. Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2009*, 2005.

[15] A. Sorensen. The many faces of a temporal recursion. `http://extempore.moso.com.au/temporal_recursion.html`, 2013.

[16] A. Sorensen, B. Swift, and A. Riddell. The many meanings of live coding. *Computer Music Journal*, 38(1):65–76, 2014.

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

[18] B. Victor. Learnable programming. `http://worrydream.com/LearnableProgramming/`, 2012.

[19] G. Wakefield, W. Smith, and C. Roberts. LuaAV: Extensibility and Heterogeneity for Audiovisual Computing. *Proceedings of Linux Audio Conference*, 2010.

[20] G. Wang, P. R. Cook, et al. Chuck: A concurrent, on-the-fly audio programming language. In *Proceedings of the International Computer Music Conference*, pages 219–226. Singapore: International Computer Music Association (ICMA), 2003.

[21] A. Ward, J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander. Live algorithm programming and a temporary organisation for its promotion. In *Proceedings of the README Software Art Conference*, volume 289, page 290, 2004.