An Analysis of Live-Coding Systems

Mert Toka
Dept. of Media Arts and Technology
University of California, Santa Barbara
merttoka@ucsb.edu

Jennifer Jacobs
Dept. of Media Arts and Technology
University of California, Santa Barbara
jmjacobs@ucsb.edu

I. Introduction

In its broader definition, live-coding relates to any coding activity which is written and executed almost simultaneously. It is not bound to a specific domain; it refers to any real-time software development. However, in recent years, the term started to appear in the context of improvisational sound and visual production through real-time programming, and this document concerns the latter definition.

Over the years, there have been many programming languages and systems that support different affordances for improvisational creative programming. There are currently 43 programming languages and 50 publicly accessible libraries/tools developed for live-coding purposes [1]. This document focuses on a subset of these systems and analyzes them on two perspectives: **(1)** the affordances of each system and their support for various modalities; **(2)** an investigation of their support for an adapted subset of cognitive dimensions of notation [2] (see Sec. II-C). The results of the analysis are also used to determine requirements for a new technology that aims to provide greater musical expression and learnability.

I selected four open-source systems for analysis based on my familiarity with the environments:

- *SuperCollider* is a server-client system for high-fidelity audio synthesis [3]
- *TidalCycles* (*Tidal*, in short) is a pattern programming language built on a functional programming paradigm for robustness and easy pattern dispatching [4]
- *Siren* is a hybrid system that merges textual programming with additional interface level features for live-coding and algorithmic composition [5]
- *Gibber* is a web-based live-coding language that supports native web audio synthesis and visual elements [6]

II. Background

A. Brief History

Music production takes many forms and shapes, ranging from acoustic instruments to digital synthesis using mathematical procedures. Iannis Xenakis used several mathematical procedures and created a generative piece, Metastasis, with a large amount of data in the 1950s [7]. Later on, the advent of dedicated digital signal processing (DSP) hardware and audio-focused software like *Pure Data* and *Max* enabled logical processes to produce sounds that were previously challenging to achieve with physical instruments.

In the 1990s, algorithms began to play an important role in computer music and live-coding has emerged as a sub-genre of computer music around the 2000s. Various artists started experimenting with real-time music production through programming [8], and in 2004, Alex McLean started TOPLAP organization to gather live-coders around a community and to promote live-coding. TOPLAP started a culture of organizing algorithmic dance events, called *Algorave*s, where live-coders perform on stage, often accompanied by live-coded visuals with source-code projected as an overlay. These events present a rich experience for the audience as they observe the changes in the source code of the music that they are dancing to.

B. Rhythm, Repetition, and Patterns

Concepts relating to time are often difficult to represent digitally. A precise characterization of temporal structures has been intentionally ignored by early computer scientists [9]. Therefore, digital systems mimic the continuity of analog reality with discrete structures of sufficient resolution. Extracting the core parameters of this continuity, i.e. rhythm, the number of repetitions, a sequence of patterns, etc., enables us to have a common vocabulary in representing arbitrary notations with ease.

Rhythm arguably is the most important parameter for our understanding of musical temporality. Even though it is possible to generate arbitrary signals using computers, musicians tend to apply certain rhythmic elements in their gestures. Some computer music systems accommodate greater flexibility to transition between different tempo and beat structures. Polyrhythms, overlapping rhythmic structures of different divisions, are also commonly used in live-coding performances. Rhythm is often coupled with repetition. Repeating a certain structure over time both ensures the delivery of its message (as in the information theory) and introduces an inherent risk of being redundant. Musicians find creative ways to balance this risk and turn it into an opportunity.

Some live-coding systems rely on *pattern*s, rhythmic structures repeating over time. When we see the sequence `abcabcabcabc...` we can infer the pattern `repeat abc` [4]. These structures make improvisational live-coding less cognitively challenging and allow programmers to focus on higher-level structures in their composition.

C. Notation

Music and programming are two creative mediums mediated through notation [2]. In both scenarios, the notation is a way to describe the execution of a system or a succession of notes. It serves as the representation of a domain and directly affects how the medium is perceived and interacted with. In textual programming, notation defines the legitimate actions, i.e. the syntax, and the design of notation directly influences the affordances of the final system. In the case of hybrid systems that support both the textual and visual programming styles, the definition of notation expands and includes any interaction method that enables a distinct creative output, i.e. the interface.

Therefore, the design of notation for creative applications can both encourage and discourage users from certain actions and formulations. Building on Green's *Cognitive Dimensions Framework* [10] developed to analyze visual programming environments, Nash investigates [2] what factors play an important role in notation design in the context of traditional paper scores, Max/MSP, and digital audio workstations (DAWs). In the following sections, I adapt a subset of Nash's inquiries to evaluate four systems (see Sec. IV) for live-coding and attempt to find an opportunity zone for prospective technology by taking the best aspects from each system.

III. Methodology

The systems in this analysis are selected due to their strengths. I have had prior experience with some; as for the rest, I reserved enough time (around 1 hour) to familiarize myself with the premise and its affordances. It is also useful to consider the fact that these programming interfaces have a relatively steep learning curve, and it is practically impossible to reveal all their features. However, I believe my prior experience combined with the targeted exploration for this analysis would prove sufficient data for the analysis. The comparison is performed on two levels. First is the identification of features and affordances of each system that yield preferred engagement with its audience (Table I). These features vary across the systems, yet I believe monitoring different properties and selecting desirable features would inform the design of a more engaging and novel system. This comparison may not result in an exhaustive study due to the extensive scope of functions of each system. Yet again, the prior knowledge of the field helps identify their respective expressivity. I chose to rank features and affordances in binary as finding a common scale for different rankings is not practical.

Second is the investigation of cognitive dimensions of notations [2] that informs the design of a new system (Table II). Nash uses sixteen core dimensions through three scenarios of notation-mediated music interaction. I chose the six most applicable dimensions, adapted the direction of each dimension so that the higher numbers are always desirable, and ranked the systems based on their compliance with the prompts. I used the five-point Likert scale for these rankings.

	SuperColl.	Tidal	Siren	Gibber
Audio Programming	+	+	-	+
Visual Component	-	-	+	+
Concise Syntax	-	+	+	-
User Interface	+	-	+	+
Output Visualization	-	-	+	+
Recording & Editing	+	-	+	-
Web Support	-	-	-	+
Collaboration	-	-	-	+
Share/ Browse	-	-	-	+

TABLE I
AFFORDANCES AND MODALITIES OF EACH SYSTEM

IV. Evaluation

An individual investigation of each system is presented in the following sub-sections:

A. SuperCollider

SuperCollider contains an environment and a programming language for real-time audio synthesis and algorithmic composition. It has been developed by James McCartney and released in 1996 [3]. It uses a server-client architecture that separates the language and audio synthesis. The server, *SCserver*, provides a powerful and robust audio generation and communicates the programming language, *SClang*. SuperCollider is not concerned with visual components.

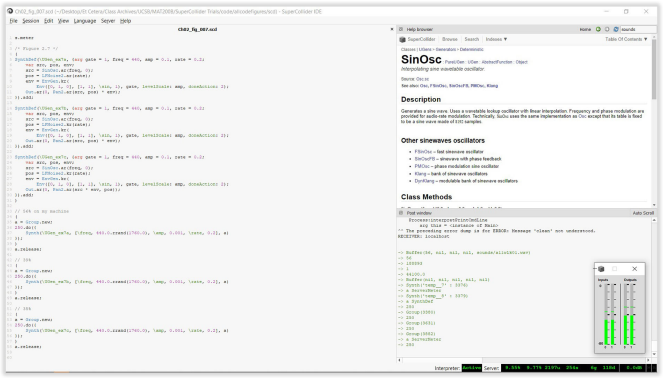

Fig. 1. Screenshot of SuperCollider environment. Code window, help browser, console, sound meter, and server status bar are visible.

SuperCollider is not developed as a live-coding environment. However, along with powerful algorithmic composition capabilities, its robust synthesis engine supports improvisational audio generation just as well. Specifically `PBind` class makes pattern generation and event dispatching easier by enabling repetitions and discrete event dispatching.

Due to its complicated yet powerful syntax, it requires hard mental operations and expert command of both programming and the music. Even a minimal audio example requires a serious time investment. The interface only displays minimal information to the user (like server-status and CPU load). Figure 1 shows an audiometer with two-channel input and two-channel output, which can be accessed through coding (`s.meter` command). However once mastered, the sky becomes the limit; it looks like any imaginable sound can be

Cognitive Dimensions	*Explanation*	**SuperCollider**	**Tidal**	**Siren**	**Gibber**
Offload Cognitive Load	*When making music, to what extend the notation helps with hard mental operations?*	1	3	5	4
Conciseness	*How concise is the notation?*	1	4	4	3
Provisionality	*Is it possible to sketch things out and play with ideas without being too precise about the exact result?*	2	4	3	3
Viscosity	*Is it easy to go back and make changes to the music?*	3	4	5	4
Learnability	*Does the notation allow newcomers to experiment with the system easily?*	1	2	3	3
Virtuosity/Expressiveness	*How expressive is the notation for experts?*	5	4	4	4

TABLE II
SUPPORT FOR ADAPTED COGNITIVE DIMENSIONS OF NOTATIONS

produced by the effective methods of the system. Since it is developed algorithmic composition in mind, it contains code level features for recording and editing the output.

Below is an example snippet that defines a simple stereo-panned sample player called `\bplay`, and `Pbind` object plays the sample player as an instrument every second with the provided arguments [11].

```
SynthDef(\bplay,
{arg out = 0, buf = 0, rate = 1,
 amp = 0.5, pan = 0, pos = 0, rel=15;
    var sig, env;
    sig = Pan2.ar(
        PlayBuf.ar( 2, buf,
            BufRateScale.ir(buf)*rate,
            1, BufDur.kr(buf)*pos*44100,
            doneAction:2),
        pan);
    env = EnvGen.ar(Env.linen(0.0,rel,0),
                    doneAction:2);
    sig = sig * env;
    sig = sig * amp;
    Out.ar(out,sig);
}).add;

(
    ~k = Pbind(\instrument,\bplay,
               \buf,d["k"][0],
               \dur,1);
    ~k.play;
)
```

B. TidalCycles

One of the prominent figures in live-coding history (see Sec. II-A), Alex McLean developed and released Tidal in 2009 as a pattern language [4]. Even though it establishes a unique and concise language powered by Haskell functional programming language, it does not synthesize the audio by itself. It communicates with *SCserver* through Open Sound Control (OSC) and triggers samples or synths. The syntax of the Tidal, however, enables economic use of characters and would result in powerful musical phrases with minimal changes. By default, Tidal does not support visual programming, yet it is fairly easy to reroute its OSC messages to a custom-made visual programming system [12].

Tidal does not have a dedicated user interface; instead, it supports text editors like Atom or Visual Studio Code with extensions (Figure 2). In one sense, this decision removes the need for learning a new interface for users; in the other sense, it deprives the system of using specific user interface features like visualizations, dedicated menu items, etc.

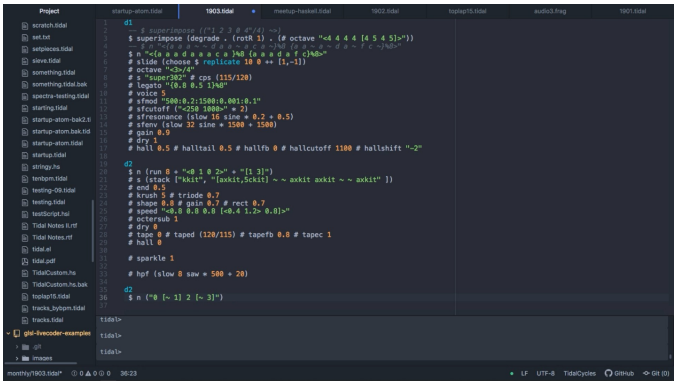

Fig. 2. Screenshot of TidalCycles running in Atom text editor.

Two example patterns are provided below. The first pattern (`d1`) plays a single `bd` sample once every cycle (defined as one second by default). This is the minimalist sound example Tidal provides. The second example (`d2`) generates a sequence of pitches for the sample `bd` and applies various effects to each of them.

```
d1 $ sound "bd"

d2 $ n "[c2 g4 c5 c3]*4" # s "bd"
    # sustain "0.1"
    # pitch2 "[1.2 3]"
    # pitch3 "[1.44 2.25 4 9]"
    # voice (slow 4 "0.25 0.5")
    # slide "[0 0.1]/8"
    # speed "-4"
```

The syntax of Tidal provides a good balance between the real estate it takes and the expressiveness it provides. Once the structure of pattern syntax is understood, the musical expression increases rapidly. Also since it uses the SuperCollider sound engine, any synths written in SuperCollider can be easily invoked by Tidal.

C. Siren

Siren is a hybrid environment for live-coding and algorithmic composition, which brings textual and visual programming together under its interface [5]. It has been developed by myself and Can Ince in 2018 to leverage live-coding traditions in offline composition with additional interface features. It depends on SuperCollider and Tidal for the audio synthesis and programming language, and both languages can be used in harmony at the same time. Various audio parameters feed its preliminary visual component that has been implemented using OpenGL Shading Language (GLSL) and ray-marching methods.

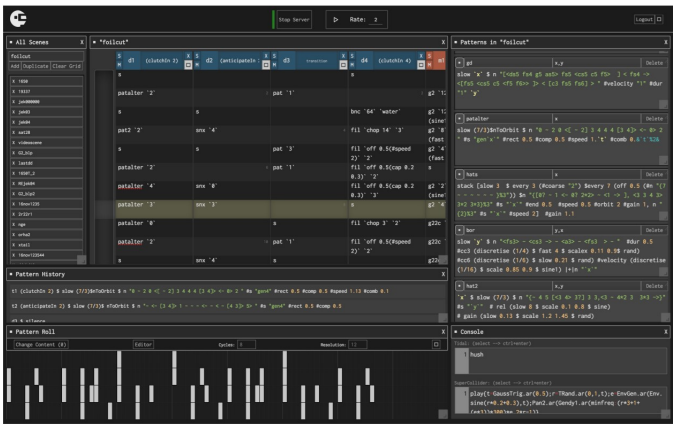

Fig. 3. Screenshot of Siren. Tracker interface, *Tidal* code blocks, history component, and a data visualization of the output are visible.

It does not have a textual programming language by itself. It interfaces with the Haskell compiler and SuperCollider server to execute code snippets. Its interface, however, extends the textual notation with various features. It visualizes the pattern outputs in a piano-roll inspired module. Live-recording can record a whole session and save it for future refinements. These additional features ease the cognitive load of the user, especially in live settings as it offloads hard calculations and relationships onto the interface.

As opposed to other systems discussed here, Siren also provides a step-tracker interface with global time controls, which makes sketching fine details and bigger compositional ideas easier. The interface would be more familiar to newcomers as the information is not hidden behind textual syntax. Besides, since it fully supports the textual input of both SuperCollider and Tidal, the mastery would yield greater musical expression, arguably even faster than the other systems.

D. Gibber

Gibber is an audiovisual live-coding environment for web browsers developed by Charlie Roberts in 2012 [6]. It supports Javascript and GLSL as main programming languages for audio and visual programming, respectively. It synthesizes audio using Web Audio API and renders visual components in WebGL.

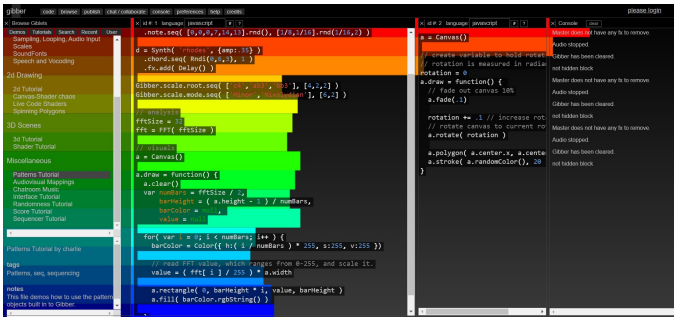

Fig. 4. Screenshot of Gibber interface. Menu bar, navigation panel, two code views, console, and transparent visual canvas are visible.

It has a well-designed interface with many functionalities. It contains a hub view that holds demos, examples, and tutorials. As opposed to the other systems, Gibber fully leverages its online nature by providing a repository of Giblets (code snippets in Gibber) shared by other live-coders. This feature motivates users to experiment with different methods, allows newcomers to test out ideas and learn the codebase faster. Networked live-coding sessions are also possible by connecting with other users [13].

The interface of Gibber accommodates an interesting visualization method for displaying the current progress of the musical output. Apart from the metronome on the top-left, `Notation.on()` command enables inline identifiers right on the executed line that shows the current state of the pattern. Figure 5 demonstrates one example on the example code.

```
Notation.on()

a = Drums('x*o*x*o-')
a.pitch = Mouse.y
```

Fig. 5. Annotation that shows the active beat in the pattern. The box moves forward in each tick of the metronome.

All these interface level features ultimately take the cognitive load off of the user. Its syntax is relatively concise compared to SuperCollider, but I find Tidal's syntax more intuitive and easier to develop on. It inherits the pattern structure of Tidal for creation rhythms and applies an object-oriented paradigm for the manipulation and variations of those patterns.

Below snippet is the main example that shows up when the user navigates to gibber.cc on the web browser. Connecting the user input, like the mouse positions, as parameters to other objects is as easy as typing `Mouse.y`.

```
a = Drums('x*o*x*o-')
a.pitch = Mouse.y

b = FM({ attack: ms(1) })
b.index = a.out
b.cmRatio = Mouse.x
```

```
b.fx.add(
  Delay({
    time:     Mouse.x,
    feedback: Mouse.y
  })
)

b.note.seq(
  ['c2','c2','c2','c3','c4'].random(),
  [1/4,1/8,1/16].random(1/16,2)
)
```

Judging by the quality and variety of the examples in its hub, mastery of the system could yield excellent musical expression.

V. Discussion

Previously discussed systems rely on textual programming for sound production. Textual programming offers greater expressivity, yet it requires a good grasp on how the syntax of a specific language works and what are how the expressiveness is embodied. It hides the possible options that could be used in the music production behind API references or language dictionaries and does not help with complex structures. Tidal takes the most similar approach to natural-languages (specifically English) with its functional programming paradigm and its choice of keywords that defines the language.

In contrast, visual programming acts as the opposite; it brings functionality to the foreground by placing them on the interface. Different types of windows, views, or modules hint their respective use cases to the user. Mastery of these components would still require training, however, it offers a lower entry threshold for newcomers and inexperienced programmers. The interface not only makes the system more user friendly but also provides useful information that would help the user make a better decision in their musical output. Visualizations and annotations employed by Siren and Gibber provide perfect examples.

Live-coding systems are powerful for real-time audio synthesis and playback. However, Tidal and Gibber are not designed to be used in non-live scenarios. In contrast, SuperCollider and Siren are designed algorithmic composition in mind. The live-coding capable features in SuperCollider are a small subset of what's possible with the environment. Siren employs various structures that resemble a DAW interface to help compose music that is built with programming primitives.

One major difference in Gibber is its use of web technologies cleverly. Anyone with access to a contemporary web browser and the internet can access the system online (gibber.cc). Running such an elaborate system on the web would have been unimaginable when SuperCollider is first released. Compared to other systems, this removes any barrier for setup and installing the software on personal computers. Chat, collaboration and network music are also possible in this modality. It also hosts a hub for sharing works and browsing for other people's code snippets. These approaches create a community around the system without relying on other methods like forums, gatherings, etc.

In the next sub-section, I briefly investigate a different type of programming environment, *Scratch*. It is not a live-coding environment, yet it contains audiovisual components and an event-driven block-based programming paradigm. Findings from this system will help formulate a new type of live-coding system in sub-section V-B.

A. Different perspective for programming: Scratch

Scratch is a block-based programming language served online (scratch.mit.edu). It represents components in textual programming with blocks of different colors that visually connect each other in various ways. The style and the use case of these blocks are highly intuitive and the system is primarily targeted at children or non-programmers [14].

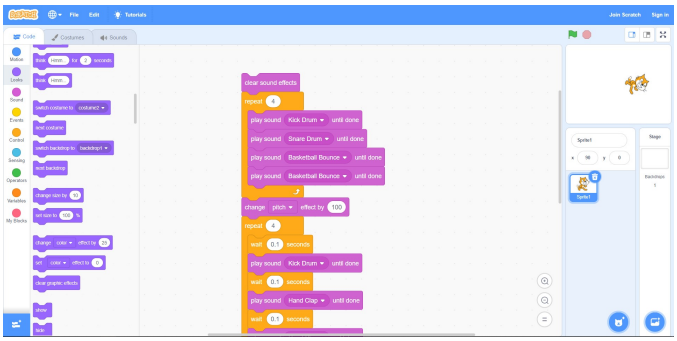

Fig. 6. Screenshot of Scratch. A simple pitch-shifted musical pattern is created with visual programming blocks.

Scratch contains blocks for events, control parameters, audio controls, and visual animations and manipulations. It also allows extensions for connecting various devices to the system.

Just like Gibber, Scratch is also built around a creative community of hobbyists. It provides a common place for sharing work and allows the remixes of projects developed by other people.

B. Proposed Technology

As a result of the analysis, I borrow the best parts of each system to propose new software for live-coding that would both be welcoming to new users and maintains virtuosity. The opportunity zone of the new system is highlighted in Figure 7:

- Robust and powerful audio synthesis for musical expression
- Support for built-in visual components
- Multilayered user interface [15] that gradually unlocks advanced features as the user gains more experience with the system
- Incorporating block-based programming for new users until they feel comfortable in switching more abstract methods of textual programming
- Economical syntax for coding that would resemble natural-languages

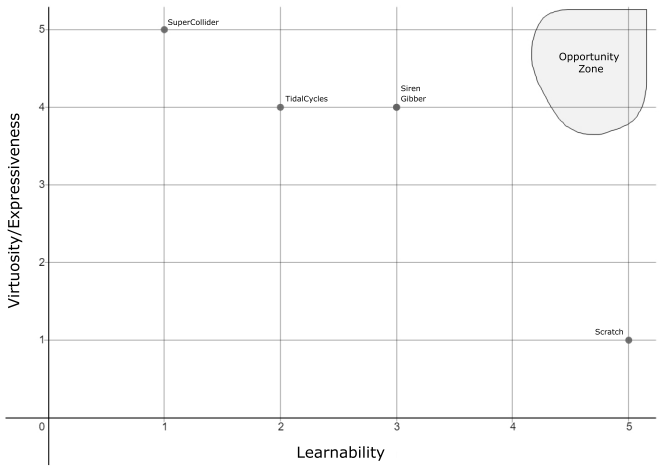

Fig. 7. Comparison of learnability and expressiveness and designation of an opportunity zone for proposed system

- Additional interface level features for helping users figuring out cognitively challenging aspects of the creative production
- Support for non-live setting by providing DAW-like features (global timers, recording, editing, etc.)
- A web interface for removing the need for installation and providing easier access
- Connecting a community of people with the ability to share works and collaborate on projects

VI. Conclusion

Live-coding is an interdisciplinary field for programmers, musicians, and visual artists. The abstract and algorithmic nature of the discipline enables rich textures in improvisational output. In this paper, I analyzed four live-coding systems (SuperCollider, TidalCycles, Siren, and Gibber) by identifying the strengths and weaknesses of each system. Inferring from block-based programming interfaces (Scratch), I propose a new live-coding system that could be beneficial for both novices and experts in music production.

References

[1] TOPLAP. (2019) All things live coding. [Online]. Available: https://github.com/toplap/awesome-livecoding

[2] C. Nash, "The cognitive dimensions of music notations," 2015.

[3] J. McCartney, "Supercollider: a new real time synthesis language," in *Proc. International Computer Music Conference (ICMC'96)*, 1996.

[4] A. McLean and G. Wiggins, "Tidal–pattern language for the live coding of music," in *Proceedings of the 7th sound and music computing conference*, 2010.

[5] M. Toka, C. Ince, and M. A. Baytas, "Siren: Interface for pattern languages," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)(Blacksburg, Virginia, USA, 2018), TM Luke Dahl, Douglas Bowman, Ed., Virginia Tech*, 2018, pp. 53–58.

[6] C. Roberts and J. Kuchera-Morin, "Gibber: Live coding audio in the browser," in *ICMC*, 2012.

[7] C. Rostand, "Metastasis," *Iannis Xenakis: Metastasis, Pithoprakta, Eonta, recording (New York, NY: Vanguard Recoding Society)*, 1967.

[8] A. McLean. (2013) A prehistory of live coding. [Online]. Available: https://toplap.org/wiki/TOPLAP_CDs

[9] A. Mclean and G. Wiggins, "Bricolage programming in the creative arts," *22nd Psychology of Programming Interest Group*, 12 2010.

[10] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131–174, 1996.

[11] coï¿¥ï¾¡pt. (2020) howto_co34pt_livecode. [Online]. Available: https://theseanco.github.io/howto_co34pt_liveCode/2-4-Pbinds-and-Patterns/

[12] 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*, 2014, pp. 63–70.

[13] G. Wakefield, C. Roberts, M. Wright, T. Wood, and K. Yerkes, "Collaborative live-coding virtual worlds with an immersive instrument." 06 2014.

[14] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010. [Online]. Available: https://doi.org/10.1145/1868358.1868363

[15] B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs, N. Elmqvist, and N. Diakopoulos, "Grand challenges for hci researchers," *interactions*, vol. 23, no. 5, pp. 24–25, 2016.