UNIVERSITY OF CALIFORNIA

Santa Barbara

Siren: General-Purpose Algorithmic Composition and Live-coding Environment

A Thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Media Arts & Technology

by

Mert Toka

Committee in charge:

Professor JoAnn Kuchera-Morin, Chair

Professor Marko Peljhan

Professor Curtis Roads

September 2018

The thesis of Mert Toka is approved.

_____

Marko Peljhan

_____

Curtis Roads

_____

JoAnn Kuchera-Morin, Committee Chair

September 2018

ACKNOWLEDGMENTS

I would like to thank my committee members for their support and effort in this project

Special thanks to my collaborator, *Can Ince*

Also, I would like to thank:

MAT Community

Fulbright Program for funding my master's education

Alex McLean for building TidalCycles and the live-coding community

Various artists and developers enabled and inspired this work

Joe the Cat for being the moral support in the time of need

My family and friends all around the world for being there and wishing for the best

ABSTRACT

Siren: General Purpose Algorithmic Composition and Live-coding Environment

by

Mert Toka

Siren is a software environment for exploring rhythm and time through the lenses of algorithmic composition and live-coding. It leverages the virtually unlimited creative potential of the algorithms by treating code snippets as the building blocks for audiovisual playback and synthesis. Employing the textual paradigm of programming as the software primitive allows the execution of patterns that would be either impossible or too laborious to create manually.

The system is designed to operate in a general-purpose manner by allowing multiple compilers to operate at the same time. Currently, it accommodates SuperCollider and TidalCycles as its primary programming languages due to their stable real-time audio generation and event dispatching capabilities.

Harnessing the complexity of the textual representation (i.e., code) might be cognitively challenging in an interactive real-time application. Siren tackles this by adopting a hybrid approach between the textual and visual paradigms. Its front-end interface is armed with various structural and visual components to organize, control and monitor the textual

building blocks: Its multi-channel tracker acts as a temporal canvas for organizing scenes, on which the code snippets could be parameterized and executed. It is built on a hierarchical structure that eases the control of complex phrases by propagating small modifications to lower levels with minimal effort for dramatic changes in the audiovisual output. It provides multiple tools for monitoring the current audio playback such as a piano-roll inspired visualization and history components.

TABLE OF CONTENTS

1 Introduction....1

1.1. Problem Statement....2

1.2. Motivations and Significance....3

2 Background Research and Related Concepts....4

2.1 Brief History of Interactive Audiovisual Systems....4

2.1.1 From Analog Synthesis to Computer Music....5

2.1.2 Demoscene....6

2.1.3 Live-Coding....7

2.2 Computational Concepts....10

2.2.1 Algorithms....10

2.2.2 Programming Languages....10

2.2.3 Programming Environments....11

2.3 Music Software....13

2.3.1 Musical Trackers....13

2.3.2 Digital Audio Workstations (DAWs)....13

3 System Design and Methods....15

3.1 Technical Overview....16

3.2 Modules....18

3.2.1 Scene....18

3.2.2 Multi-Channel Tracker....19

3.2.3 Pattern Dictionary....23

3.2.4 Console....25

3.2.5 Playback 26
3.2.6 Global Controls 28
3.2.7 History 28
3.2.8 Pattern Roll 29
3.2.9 Graphics 30
3.2.10 Debug Log 32
3.2.11 Settings 32
3.3 Interface-level Features 32
3.3.1 Layout 32
3.3.2 Save 33
3.3.3 Context Menu 33
3.4 Connectivity with External Hardware and Software 33
4 Results 36
4.1 Deliverables 36
4.2 Community Adoption 36
4.3 Milestones 36
4.3.1 sq 37
4.3.2 Siren 38
4.4 Evaluation 38
5 Conclusion 43
5.1 Future Directions 43
6 References 45

LIST OF FIGURES

Figure 1 Timeline of the early live-coding activity 7

Figure 2 A screenshot of the interface (top) and its simplified diagram (bottom) 15

Figure 3 Block diagram of Siren 16

Figure 4 Module blueprint 18

Figure 5 The breakdown of a channel header 20

Figure 6 An example of a function call on a channel cell 22

Figure 7 An example of a pattern function in the dictionary 23

Figure 8 Example of a pattern function with parameters and math. expressions 25

Figure 9 Console modules of TidalCycles (left) and SuperCollider (right) 26

Figure 10 Parsing the recorded file (left) for scene generation (right) 27

Figure 11 The pattern roll module with two active channels 29

Figure 12 Two examples of the visuals on the graphics module 31

Figure 13 Some examples of visuals in Processing controlled by Siren 34

Figure 14 Interface of the previous version of Siren, sq 37

1 Introduction

The last century has witnessed a rapid increase in the computational power, paving the way for a shift from previously tangible creative endeavors to their computational counterparts. Digital sonic explorations have become possible with the cutting-edge research by the universities and the industry, followed by the advent of image synthesis towards the end of the century. The emergence of abstractions in human-computer communication has helped programming languages become ubiquitous, and commanding the machines using these higher-level languages enabled translating artistic ideas into computational problems.

Live-coding is one of the most recent creative practices that benefited from the advancements in computer technology. Even though the naïve definition could be real-time software development for any domain (such as database systems, software engineering, and embedded systems), for this document, we will limit ourselves with the domains for creative endeavors only. Then the definition becomes “the improvisational creativity technique through interactive programming to create sound and image-based digital media.”

In its purest form, the medium is composed of two primary materials. *The algorithm* is the method of structuring information unambiguously, and *the programming languages* are the syntactical representation of this information that is used to communicate with the computational device. When mastered, both materials promise great flexibility and power in information processing, and the creative applications could quickly benefit from this expressive potential.

1.1. Problem Statement

Live-coding is a compelling way for media artists and engineers to be able to work with the interactive computation in real time. However, the tools adopted for the execution of the practice has not evolved much over the course of two decades.

Firstly, many programming languages are developed for live-coding music and visuals, and each is tailored to different problems in its making. Usually, live-coders prefer mastering one of these languages and persist with it, however, in some cases, the execution of one method could be more comfortable using another language. In this case, the performer needs to set up multiple compilers and make sure both systems are running correctly and in harmony. Setting up such a system is a tedious task that often requires advanced technical and mixing skills.

Secondly, the practice is only dedicated to the improvisational creativity. Although in some examples previously crafted code could be used, the flow of the performance is primarily determined by the impromptu programming. This spontaneity results in missing an editable recording of the performance for future manipulations and failure to make the best of the creative potential of such a robust method of execution in the offline composition.

Thirdly, there haven’t been nice hybrid systems that brings the textual representation of the code with the visual representation of an interface in the context of live-coding.

Lastly, algorithmic thinking and programming is a challenging task even when performed offline. Maintaining and expressing intricate sonic connections through a textual representation introduce a fair amount of cognitive load onto the performer.

1.2. Motivations and Significance

This Masters document describes *Siren*, a live-coding environment that aims to tackle abovementioned shortcomings in its design.

Rather than only allowing the improvisational live-coding, Siren treats algorithm and code as a compositional tool. It packs a tracker module that works with augmented code snippets, to allow for fine-tuned algorithmic compositions. The user could use both methods at the same time for exciting results.

It also accommodates multiple compilers for the execution of different languages at the same time. Currently, we support SuperCollider and TidalCycles since they cover most of the need in the live-coding community. Also, it is easy to record and play sessions using the recording feature in the system. The recordings can then be translated into full-fledged tracker-based compositions using the playback module for later editing.

Although the live-coding manifesto published by TOPLAP [1] demands to strip down the bare information, i.e., the code in its rawest form, we also believe that we need to explore the possibility of hybrid systems that allow performers to use the power of algorithms more efficiently. Siren combines the full creative power of the textual information, *the code*, or *the algorithm*, with a visual representation, *the interface*, to explore the articulative potential of hybrid systems in real-time audiovisual creation. It treats code as a creative primitive and allows for organizing complex structures additively on its interface.

Lastly, the system accommodates various interface-level modules that help mitigate the cognitive load of real-time coding such as providing visualization of recent playback and means for hierarchical control of multiple channels.

2 Background Research and Related Concepts

As mentioned in Chapter 1, the essence of creative programming deals significantly with *algorithms* and *programming language*s. The result of these processes reveals on the successful execution of the textual information using an appropriate compiler or interpreter. Writing the source code and the execution of the program could be performed either on separate software, i.e., *text-editors* and *Command-Line Interfaces (CLIs)* or together on an *Integrated Development Environment* (*IDE)*. Since the creative programming is not immune to these steps, Siren puts an effort in improving the potential of this field by bringing the teachings of live-coding to an interface with visual components that are heavily inspired by *IDEs, Musical Trackers,* and *Digital Audio Workstations* (*DAWs)*.

This chapter, first, briefly talks about the history of creative programming by touching upon the milestones in interactive audiovisual systems. Moreover, it highlights the critical concepts in the first paragraph by discussing algorithms, programming languages, text-editors, CLIs, IDEs, Musical Trackers, and DAWs.

2.1 Brief History of Interactive Audiovisual Systems

It would be incomplete to talk about interactive audiovisual systems without mentioning their precursors. When we consider interactive sonic systems, the earliest and most widespread use of the medium is the acoustic field. Musicians have been performing with acoustic instruments for thousands of years [2]. However, when we focus on the evolution of computer music, it would be feasible to analyze the analog systems first.

2.1.1 From Analog Synthesis to Computer Music

In the mid-20th century, the room-sized computers were reserved for number crunching and making calculations, and they were usually not powerful enough to perform real-time audio processing. In the meantime, the analog systems were getting robust enough to perform interactive audio synthesis. The development of first analog synthesizer (the RCA synthesizer) in 1959 [3] led to the golden age of analog synthesizers in the next decade. Buchla and Moog's analog synthesizers were capable of real-time audio processing and synthesis in the 1960s [2].

The discovery of microcontrollers in the 1970s has allowed for the rapid investment in small computational devices, especially for personal computers, and resulted in several commercially successful products like Xerox Alto (1973), Apple II (1977) and Commodore 64 (1982). They featured displays and input/output devices for the interaction with the machine [2].

In 1983, Yamaha released the DX-7 FM synthesizer, and it quickly became the first successful commercial digital synthesizer [3]. In the same year, the standardization of the *Musical Instrument Digital Interface* (MIDI) protocol created new possibilities in inter-device communications. At this point, computers could control the digital synthesizers over the MIDI in real time but not synthesize sonic material on their own. For example, in his work *Hungers* (1987) Morton Subotnick controls an entire studio worth of MIDI equipment from a Macintosh computer, and the computer responds to input from a MIDI keyboard [3].

Towards the end of the 1980s, real-time computer music on personal computers seemed possible due to the rapidly increasing microprocessor power. The NeXT computer is released in 1988 with two microprocessors—one reserved for floating-point arithmetic—

and a digital signal processor, which enabled the system to process audio in real time [2]. In addition to hardware developments, in the 1990s, audio-focused programming languages such as *SuperCollider* (1996), *Pure Data* (1996), *Max Signal Processing* (1997) have started to emerge.

2.1.2 Demoscene

While computer music keeps evolving, another subculture that brings code, sound, and graphics together in a challenging way has started to mature in the 1990s. The *demoscene* is a nerdy subculture for a worldwide network of creative people who come together to create *demo*s [4]. These demos are small executable programs that showcase the aesthetic and technical skills of their artists and developers. Usually, each demo is constrained within the file size of 4kb to 64kb, resulting in the need for procedural textures, geometry, animation, and audio, by employing as little assets as possible. The *demoparties* have been organized all around the world to allow groups to meet and compete for the most impressive demos. Each group live-develops one project throughout the party, and the audience votes to determine the winner.

Demoscene is a challenging field for everyone involved in the development, and it paved the way for advancements in multiple fields such as computer art and video game industry [4]. Even though people are still interested in this culture, it lost its popularity at the beginning of 21st century due to excessive computational power and higher-level application development opportunities [5], but it ultimately cultivated other subcultures and promoted the aesthetic coding.

2.1.3 Live-Coding

The history of the live-coding is newer than the previously mentioned fields. In the mid-1980s, some performers have experimented with programming on stage and shared the source code to the audience [6] [7]. However, the performers started focusing on the field only after the 2000s (see Figure 1) and the term “live-coding” has matured.

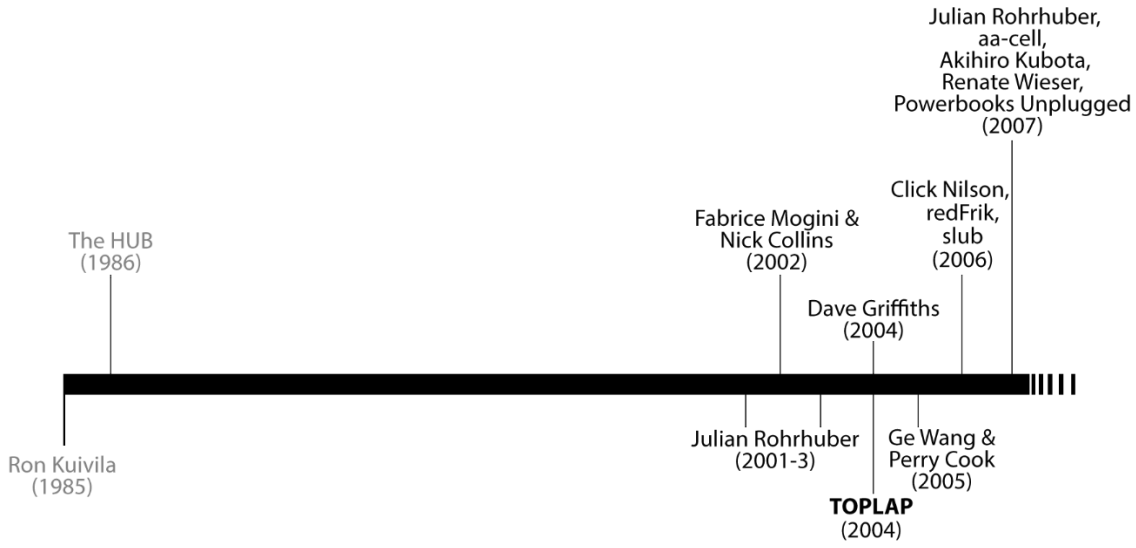


Figure 1 Timeline of the early live-coding activity

Above figure shows the distribution of early live-coding activities [8]. One of the most prominent milestones in this short timeline is the foundation of TOPLAP by Alex McLean in 2004. TOPLAP enabled live-coders to meet around a community to explore and promote live-coding. The main website of the organization includes various resources such as discussion forum, a chat room, numerous links to events, pictures, videos, and software.

In addition to providing a community for live-coders, TOPLAP organizes *Algorave*s all around the world. Algoraves are the algorithmic dance events where live-coders performs on stage and people dance to the sounds of algorithms. Performers all around the world come

together each year mid-March to celebrate Algorave's birthday with a 24-hour online stream of live-coding.

According to the list of live-coding languages compiled on TOPLAP GitHub repository [9], there exist 34 live-coding languages today. Judging by the Algorave performances, the community seems to be favoring SuperCollider and TidalCycles the most, with some exceptions of other systems here and there.

2.1.3.1 SuperCollider

SuperCollider is a platform for audio synthesis and algorithmic composition, initially developed by James McCartney and currently maintained by the community as an open source, cross-platform software [10]. It functions on a server-client architecture (i.e., *scserver - sclang*) that communicates through Open Sound Control (OSC) [11]. SuperCollider IDE provides a convenient editor for writing and executing sclang code with built-in help tools.

Due to its robust and verbose audio synthesis capabilities, SuperCollider plays a central role in most of the other live-coding languages like TidalCycles. It often requires a significant amount of typing compared to the other languages, and it does not have a standardized method of rhythm built inside. Instead, there are ways to introduce rhythmic elements like the *PBind* class, and often it is used in conjunction with *SynthDefs* which contain the description of the sound synthesis as a building block.

2.1.3.2 TidalCycles

In contrast to the SuperCollider's complicated syntax, in 2010, Alex McLean introduced a live-coding language—TidalCycles— that promotes writing patterns with a highly

economical syntax. The language is embedded in Haskell functional programming language, and it is compiled using the Interactive Glasgow Haskell Compiler (GHCi). It offers pattern representation, a library of pattern generators and combinators, and an event scheduler [12].

TidalCycles defines articulate operations on the patterns in its design; the patterns can be concatenated, merged, transformed, and combined with other patterns. Each pattern lives in a cyclical time structure that repeats itself indefinitely until either it is stopped or manipulated by the performer [13]. It also supports several channel-wise transition functions to be applied in between the pattern alterations.

The simple definition of the pattern does not immediately imply the sonic output; instead, it generates a series of triggers that could be bound to any action—drawing shapes or playing notes. For the synthesis and playback, it relies on SuperCollider with a custom SuperDirt sampler [14], and it can communicate with any other medium via OSC or MIDI protocols.

2.1.3.3 Others

Apart from SuperCollider and TidalCycles, there are numerous live-coding languages available to the performers, each of which with a different advantage. Gibber [15] enables performers to use browsers as a coding canvas by introducing a web-based synthesis engine. Conductive [16] offers a higher-level abstraction with density look-ups as its central pattern generation method. FoxDot [17] interactively abstracts SuperCollider in a fast and user-friendly manner. Finally, as a proprietary visual programming language, Max/MSP [18] promises different interaction technique to the textual representation of the code.

2.2 Computational Concepts

This section explains the concepts that have informed the design of the Siren from the standpoint of algorithmic reasoning and software development.

2.2.1 Algorithms

The *algorithm* could be defined as a procedure with an unambiguous set of rules that leads to the development of the desired output from a given input. It is a mathematical construct, or a recipe, that *abstracts* the procedure by implying as little as possible about the exact method of implementation [19]. An algorithm could be implemented in knitting, in solving mathematical formulas, or in coding software.

People have been using algorithms without even realizing since the dawn of early human history. However, the formal recognition of the term has started to emerge in the 20th century following the developments of mathematics and computing technology by key figures such as Alan Turing, David Hilbert, Charles Babbage, and Claude Shannon [19].

2.2.2 Programming Languages

Just like the natural languages, the programming languages have emerged from the need to communicate with the other party, i.e., the computational device. This *textual information* encodes the algorithm with user-friendly wording and transmits instructions to the device upon the successful compilation and translation.

After the device-specific assembly languages in the 1950s, higher-level languages like Fortran took off and introduced programming to a broader population (for example, the initiation of MUSIC-N series by Max Mathews [20]). Much more general-purpose languages, like C, have appeared around the 1970s and revolutionized programming. C has

offered the versatility by enabling programmers to code anything using this higher-level language. The advent of object-oriented languages, like C++, enabled structuring large programs as a combination of easily manageable sub-programs.

Towards the end of the century, different paradigms such as visual, functional, and scripting have emerged with the languages of Max, Haskell, and JavaScript, respectively. These new paradigms offered ways for approaching various domain-specific problems more naturally.

2.2.3 Programming Environments

Multiple types of publicly-available products are developed to write and execute the source code which translates the underlying algorithms to the programmer. While crafting the source code requires some writing tool, executing the code requires another tool for compiling and translating it into the machine language. Command-line interfaces are usually equipped with tools to call compilers or interpreters. However, an IDE is the software that brings these two together and accommodates for both the writing and execution of the code with helper tools such as debuggers, memory managers, and class creators.

2.2.3.1 Text Editors

The evolution of digital text editors started out with the simplest tools that merely translates the keystrokes into the characters on a text file. The built-in editors in popular operating systems such as *Notepad* in Windows, *TextEdit* in MacOSX, and *Gedit* in Linux offer nothing more than this simple interaction of typing. However, as the development process grows substantially, the programmers demand additional features from an editor.

Intermediate text-editors, like *Notepad++*, attract attention due to their lightweight build and support for simple features like syntax highlighting based on the language.

The contemporary ones such as *Atom, Visual Studio Code,* and *Vim* offers far more sophisticated features for software development. Additional code formatting shortcuts, interactive error checking, and collaboration integrations are some to mention. Usually, they maintain an open-source plug-in store for further custom features and allows viewing external programs like CLIs on the primary interface.

2.2.3.2 Command Line Interfaces (CLIs)

Before the graphical user interfaces, the principal means of interaction with computers was performed in command-line interfaces. Today, the end users of computers do not rely on CLIs for navigating around, yet many software developers still prefer to use it. The user interacts with the device by issuing texts to the command-line, and CLI executes the appropriate command accordingly. The source code that is stored in the main memory can be compiled and run using these tools. Programs with command-line interfaces are usually easier to automate and communicate through web sockets.

2.2.3.3 Integrated Development Environments (IDEs)

The integrated development environment provides elaborate facilities to the programmers for the software development. They contain advanced text editor, debugger, and compiler as its essential components. The various additional features such as debugging memory states, visual class generators, and intelligent code autocompletion maximize the productivity of the programmer by relieving the cognitive load of coding and transferring it into the tools on the interface.

Where some IDEs are tailored for a specific programming language, usually they support multiple compilers. Some of the most popular IDEs today are the *Visual Studio*, *Xcode*, *Eclipse*, and *PyCharm*.

2.3 Music Software

Below topics discuss the musical systems that have been influential on the design of Siren.

2.3.1 Musical Trackers

Trackers are text-based music composition tools which were popular in the early days of software with graphical user-interfaces. Initial implementations for AmigaOS at the end of the 1980s include Ultimate Soundtracker (1987) [21] and NoiseTracker (1990) [22].

The interface of the musical trackers resembles spreadsheet-like grid, where the columns represent channels or tracks, and rows denote discrete time steps. Each cell of the grid holds textual information that corresponds to the instrument, pitch, amplitude, or a control message for the resultant sound. The execution of the cell is either triggered by a timer or by the user who controls the interface using a computer keyboard.

The research of Nash and Blackwell suggests that the tracker paradigm offers greater virtuosity and flow when compared to traditional sequencers [23].

2.3.2 Digital Audio Workstations (DAWs)

The digital audio workstation is the most comprehensive solution for complex audio recording, editing, and mixing. It augments the music production process just like an IDE does to the software development. Some versions are integrated into hardware, but usually, they refer to the software with graphical user-interfaces. Built on top of the multitrack

recording functionality, contemporary DAWs allows operations on multiple channels for mixing and production purposes. They offer numerous components tailored for specific use cases—e.g., waveshape viewers for different tracks, digital piano roll, and external plug-in support.

3 System Design and Methods

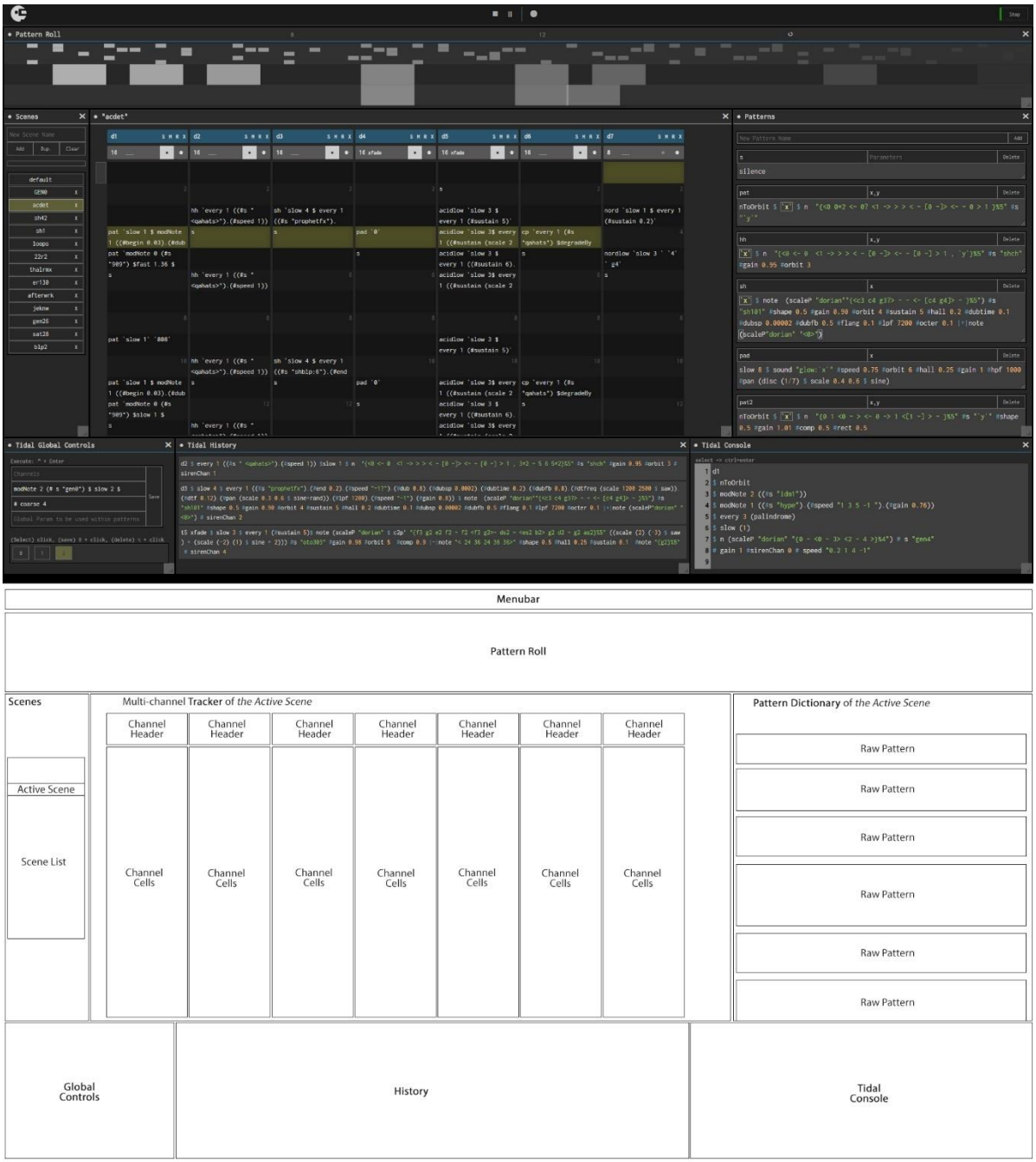


Figure 2 A screenshot of the interface (top) and its simplified diagram (bottom)

This chapter describes the details of Siren and articulates how the system design embodies its motivations. It, first, provides a technical explanation in Section 3.1 by

outlining tools and frameworks used in the implementation. Then in Section 3.2 below, it elaborates on how each module operates and contributes to the design objectives. After mentioning other interface-level features in Section 3.3, this chapter concludes with a brief discussion on the communication of Siren with other hardware and software in Section 3.4.

3.1 Technical Overview

Siren is realized using various open-source tools and subsystems. It consists of two main parts: the front-end (*interface*), and the back-end (*server*).

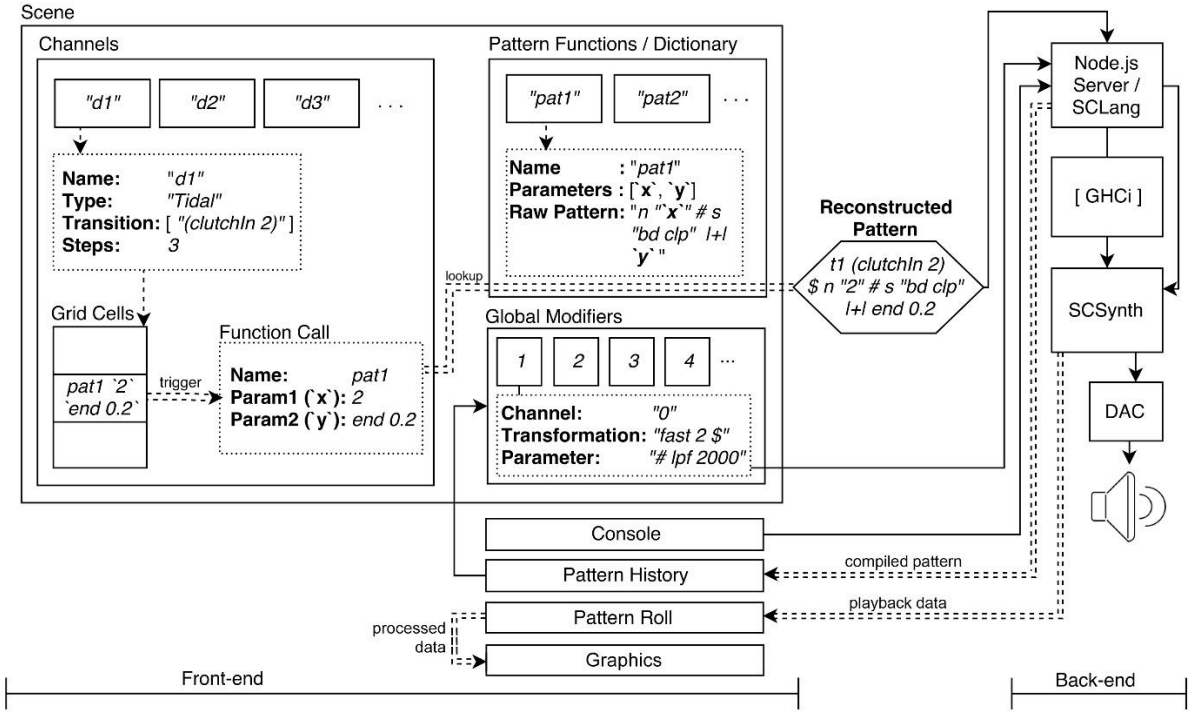


Figure 3 Block diagram of Siren

Figure 3 outlines the main components and their communication with the rest of the system. On the left, the front-end lays out the hierarchical structure [24], and on the right, the back-end reveals behind-the-scene components. In the figure, solid boxes denote prominent modules, and their placement reveals the hierarchy relations. Single-dashed boxes

and dashed arrows provide examples to these components. Solid arrows denote message propagation, and double-dashed lines represent labeled actions. Fields with square brackets (`[ ]`) are optional.

Both parts of the system are implemented using JavaScript-based web technologies due to their optimistic outlook and versatile nature. Web services provide means for reliable local execution, as well as hosting the system on world wide web. Currently, rather than hosting the system online, we chose to use local execution for robust audio generation and networking. We employ an *Electron* package that enables building cross-platform desktop applications using the web technologies [25] to convert the web-based system into a native software.

The interface is primarily implemented with *React.js* [26] for fast content rendering, and *MobX.js* [27] is used as a bidirectional state manager. We also employ other open-source libraries for various parts of the interface: *Codemirror* [28] is used for syntax highlighting and advanced text-editing features wherever any type of code is visible to the user. *react-grid-layout* [29] constructs the general modular layout of the interface with draggable components. *Supercollider.js* [30] communicates with and controls SuperCollider server. *Abletonlink* [31] and *TidaLink* [32] affords for precise time events in the tracker module (see Sec. 3.2.2.1). *gl-react* [33] provides GLSL shader support to the React.js environment in the graphics module (see Sec. 3.2.9).

The back-end acts as a core to the whole system and uses *Node.js* [34] for sustaining the communication between the front-end, the compilers, and other external systems (such as any other OSC-enabled software or hardware controllers). Node.js is an open-source, cross-platform JavaScript environment that can execute JavaScript code outside of the web

browsers. It hosts the front-end using *Express.js* [35] and enables Siren to work with separate systems like Haskell compiler and SuperCollider server. The back-end initializes the SuperCollider server and Interactive Glasgow Haskell Compiler (GHCi) for compiling reconstructed SuperCollider and TidalCycles messages, respectively.

3.2 Modules

Siren is built on a modular grid layout where different functionalities are embodied using the *modules* (see Figure 2). While a few works across all compilers, some modules are tailored for a specific one. We represent this information with a compatibility strip in between the module header and content area (see Figure 4). As per convention, we designed each module with the following blueprint:

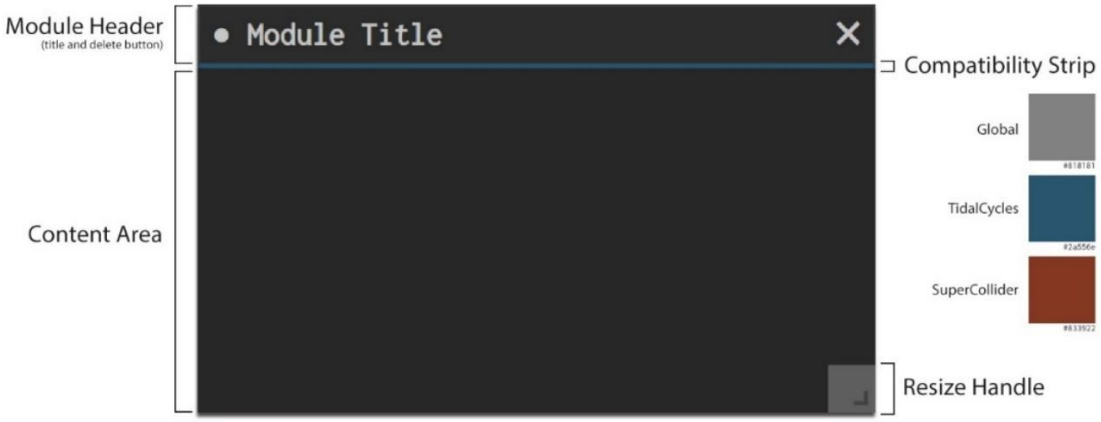


Figure 4 Module blueprint

3.2.1 Scene

The primary structure of Siren is the concept of the *scene*, which acts as a top-level container for *multi-channel tracker* and *pattern dictionary* modules that are explained in Sections 3.2.2 and 3.2.3, respectively.

Using the scene module, a user can create an empty scene, see the list of existing ones, duplicate or delete any one of them at her will. Each of the listed items is entirely independent of one and other and contains unique sets of channels and patterns. When selected from the scene list, its channels in the tracker and pattern functions in the dictionary loads.

The product of each scene –contents of its tracker and patterns– can be thought of as a musical phrase or a motive. Combining these structures back-to-back creates sonic explorations in the form of a song. Therefore, we designed a feature called the *song mode* where the tracker of each scene is played from top to bottom, and whenever its done the next scene gets activated. The actuator button for this feature is located on top of the scene list.

3.2.2 Multi-Channel Tracker

The primary tool for algorithmic composition in Siren is the module called *tracker*. It functions around a single time structure and accommodates any number of *channels* that are analogous to the *tracks* in traditional DAWs. Channels are composed of a *header* and an array of *cells*.

3.2.2.1 Time

One of the most crucial points in the system is its temporal structure. It forms the backbone of the compositional aspect of Siren. We implemented a precise *tick mechanism* built on top of the Abletonlink that is mainly shared by all channels on the tracker, as well as other time-dependent modules like Pattern Roll and Graphics (see Sec. 3.2.8 and 3.2.9, respectively). This mechanism allows all events to be triggered synchronously. We currently use two ticks per second, and the number could be adjusted at the user's will. The controls

of this timer can be found in the Menubar (see Figure 2) or can be toggled using *ctrl*[1]*+enter* while the focus is outside of a code block.

This timer scans the cells of each channel from top to bottom and triggers the content of the cell as it moves. In turn, the corresponding code in the cell gets reconstructed by Siren and compiled using the appropriate compiler. The compilation of the triggered event is synchronous with the timer in Siren. However, a new temporal progression is forked after the trigger depending on the type of the compiled code. TidalCycles uses *cycles* as its base rhythmic element, and SuperCollider operates on its temporal form depending on the code written. The time on the tracker functions as a control parameter by introducing modulations using variables and transitions.

3.2.2.2 Channel Header

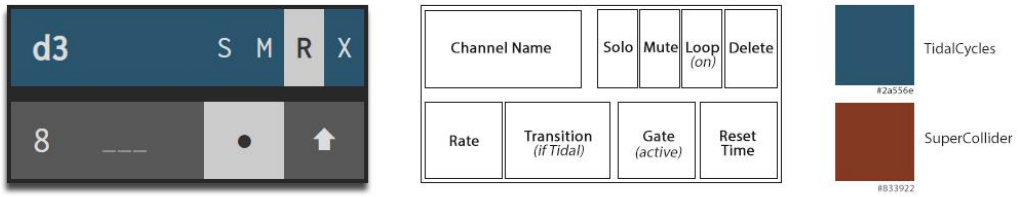


Figure 5 The breakdown of a channel header

The header field of each channel defines the properties and provides operational controls. Figure 5 explains the header of a TidalCycles channel. Each channel has a *type* property that determines which compiler to use when the content of its cells is triggered. This property is denoted by the color of the header, which can be seen on the right of Figure 5. If the type is TidalCycles, a *transition* field appears on the bottom row that allows using

[1] Control *(ctrl)* on Windows and Command *(cmd)* on Mac operating systems. Rest of the document will only use the former, but it is safe to assume the latter for Mac systems.

TidalCycles transitions when changing the active pattern (refer to Sec. 2.1.3.2 for an explanation of the transition in TidalCycles).

The *name* is a unique identifier of a channel in the scene and is used for correctly reconstructing the pattern. By default, TidalCycles expects channel names as **`d1...9`**, but it is trivial to add more channel names by manipulating the necessary files on the system. We do not impose a specific naming convention for the SuperCollider channels, however, since it is used to determine the correct audio bus, we enforce using a unique one for each scene.

The header also governs the time-based operations on a channel. The *gate* –which is active on Figure 5—is a toggle button, and when active, it listens to the ticks generated from the main timer. The speed of the time progression on a channel depends on the *rate* parameter. It uses modulo arithmetic on the tick numbers to advance the timer onto the next step. In the current case, a channel rate of 8 sustains 4-second intervals between consecutive cells. Lastly, the *reset* button resets the timer by bringing it to the first cell of the channel.

Several channel-wise operations are also available on the header. *Solo* button deactivates all other channels on the scene, and *mute* acts as the opposite by deactivating the channel itself. The *loop* button determines whether channel loops back to the beginning once the timer triggers the last cell of the channel. The song-mode described on page 18 disables the loop features of all channels to operate appropriately.

3.2.2.3 Channel Cells

In contrast to the traditional trackers that holds musical notes or control change messages, the tracker *cells* in Siren holds *function calls* of the raw patterns, which are defined in the *pattern dictionary* (see Sec. 3.2.3).

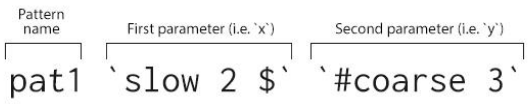


Figure 6 An example of a function call on a channel cell

Cells are the vertical list of text boxes into which the function calls are written. Figure 6 exemplifies a function call on the channel cells. A function call consists of the pattern name, which is used to determine correct item in the pattern dictionary, and a list of optional parameter instances, which are surrounded by grave accents (`). If the parameters defined in the pattern dictionary have default values, instantiating the corresponding parameter may be omitted.

We designed two methods of executing the cell. First, the execution on time trigger is briefly explained above. The other method is to use key bindings (*alt+enter*) on the highlighted cell (see next paragraph). When the cell is triggered using either method, its corresponding implementation is looked up from the dictionary, and the pattern gets reconstructed by replacing the parameters in dedicated places (see Figure 3). Then, depending on the channel type, the back-end receives the reconstructed pattern and compiles it using the appropriate compiler.

The combination of multiple vertical channels creates a spreadsheet-like grid, and useful operations on the grid are provided for convenience. While a cell is focused, pressing *Enter* initiates the selection mode, where the user can navigate the grid using the regular spreadsheet controls. *Arrows* (i.e., arrow keys) navigate the actively selected cell, *shift+arrows* selects multiple cells, *ctrl+arrows* jumps to the last cell on the direction of the arrow. Clipboard operations are also available–*ctrl+x* cuts selected cells, *ctrl+v* pastes copied cells, and *del*/*backspace* deletes the contents of the cells. While pasting, if the

number of vertical cells on the clipboard exceeds the number of available cells below the current one, it automatically inserts new cells by increasing the overall step size of the channel. The quick cell creation provides an easy way to scale-up the composition space on the tracker.

3.2.3 Pattern Dictionary

This module holds an array of augmented code snippets or patterns. The user can write regular code blocks that serve a specific purpose, augment it by parameterizing desired parts and instance it in one of the channel cells for execution.

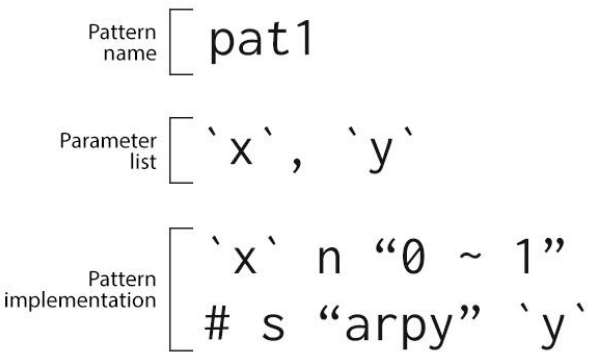


Figure 7 An example of a pattern function in the dictionary

Each item in this module consists of a unique name, an optional parameter list, and an implementation field (see Figure 7). The type of the code in the implementation does not matter since it is executed when triggered by the tracker and the error check is performed on the compiler.

The primary feature of this module is to augment patterns by using *parameters* or *mathematical expressions*. When combined with the multi-channel tracker, these tools offer improved control and modulation.

Siren offers three types of optional parameters to be defined in the pattern functions: *literal*, *random*, and *reserved parameters*. Anyone of these is defined by surrounding a substring by grave accents (`). The same parameter can be defined in multiple places in the pattern implementation for complex relations and modulations.

A literal parameter does not have any specific type and any input that reconstructs a syntactically correct output is accepted. Figure 7 demonstrates the definition of a pattern function with two parameters, located in the pattern dictionary, and Figure 6 exemplifies the function call of the same pattern function from one of the grid cells. For example, users can pass `` `slow 2 $` ``, `` `every 4 (slow 2) $` ``, or `` `slow 2 $ every 4 (slow 2) $` `` as the first parameter of **pat1**, without having a character limitation. If the substring defining the parameters includes a question mark followed by a value (e.g. `` `x?5` ``), this parameter is said to have a *default value*. In this case, a grid cell can omit instancing this parameter to utilize its default value.

A random parameter is defined in the same way as the literal parameter does, but since it produces a random number, it needs to be in place of a numerical field. Also, its corresponding function call needs to satisfy specific syntax—two numbers surrounded by vertical bars and grave accents and separated by a comma (e.g. `` `|0.1,3.2|` ``). When compiled, the random result will be calculated and replaced in the definition.

The third parameter type, the reserved parameter, is predefined strings that sustain the communication between the channels in the tracker and the pattern functions themselves. They do not have an explicit function call, and their values are implicitly calculated from the current state of the channel that they are referred. We dedicated the parameter `` `t` `` to pass the value of time, `` `rt` `` for the rate, and `` `st` `` for the step count of the channel. These

dynamic values create many opportunities in the modulation of the patterns from the tracker interface, and they are particularly useful when combined with the mathematical expressions (see Figure 8).

In addition to the parameters, Siren offer using a wide range of mathematical expressions that enhance the pattern creation process even further. They can be defined in the implementation of the pattern functions and can be combined with any parameter. Upon successful reconstruction, the expression is replaced by its results. They are defined by surrounding any mathematical operation by ampersand signs (**&**). It supports basic operation on number spaces, symbolic calculations, vector and matrix arithmetic, and trigonometry.

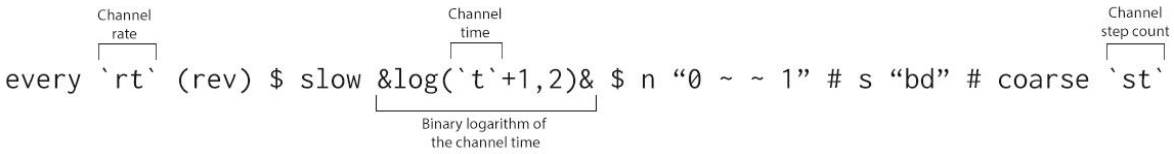


Figure 8 Example of a pattern function with parameters and math. expressions

The above figure demonstrates the use of parameters and mathematical expressions together in a single pattern function. The example pattern will reconstruct into different patterns depending on its location on the tracker. Here, the speed of the pattern is in logarithmic relation with the time, the sample rate of the playback depends on the number of steps, and the direction of the pattern changes based on the rate of the channel.

3.2.4 Console

Apart from the algorithmic composition described in the modules above, the interface of Siren is also suitable for conventional live-coding, and the *console* modules are the primary tools for impromptu programming. These modules function as a regular text-editor where

the performer types code in real time and compiles with the given key binding. She can either select the desired portion of code and compile it with *ctrl+enter* or use the same binding to evaluate a code block automatically.

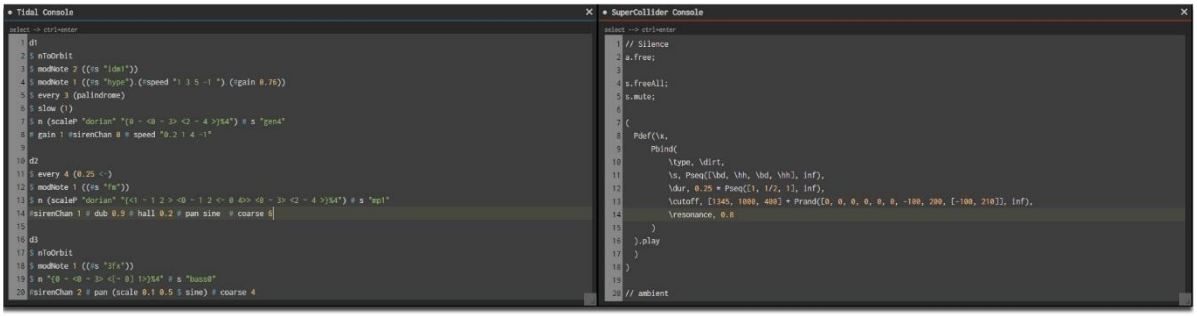


Figure 9 Console modules of TidalCycles (left) and SuperCollider (right)

Currently, Siren accommodates two versions of the console module, one for SuperCollider and another for TidalCycles (see Figure 9). Former sends the executed code block to the SuperCollider server, and the latter to the Haskell compiler.

It is developed keeping in mind the standards of contemporary text-editors, such as syntax highlighting and text editing capabilities. Codemirror and custom syntax descriptors provide correct highlighting for both versions of the consoles. They display line numbers, highlight currently active line, and implement smart block-code execution.

3.2.5 Playback

This module provides a convenient tool to record and save the playback of a live-coded session. It can facilitate both conventional live-coding on the console and algorithmic tools on the tracker interface. Moreover, it can cleverly convert regular live-coding sessions into Siren's tracker format for easier future manipulations and mastering. Figure 10 provides a simple example of this feature by showing the contents of the recorded file on the left and the generated scene on the right.

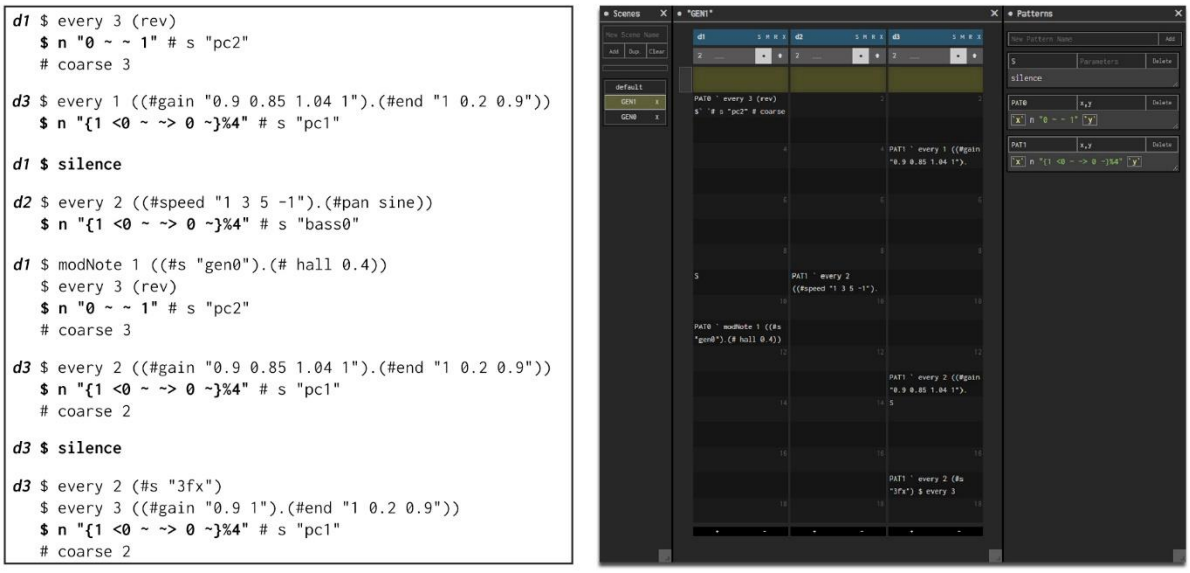


Figure 10 Parsing the recorded file (left) for scene generation (right)

The user starts recording the session with the record button on the Menubar. After this point, the system records every code compilation, and when the user stops recording the session, the back-end stores this information in a file. The playback module lists these files and allows taking notes for each file so that finding the desirable recording the next time would be more comfortable.

Once we save the recording, the module displays two options. First one is the playback of the file, which will not display any visual clues but will play the session entirely using the correct timestamps. The second feature is converting the save file into Siren's tracker format automatically.

To achieve this, we start by generating an empty scene. The idea here is to parameterize patterns and place them into the pattern dictionary with as little repetition as possible. We determine the core portion of the pattern (bold texts in Figure 10) and parameterize everything before and after the core portion with parameters. Once the channel name is determined (italic text in Figure 10), we either create a new channel or use one of the

existing ones. If the pattern core is already in the dictionary, we place the pattern onto the appropriate place in the tracker with different function calls.

3.2.6 Global Controls

This module is dedicated to modifying all active TidalCycles patterns. The component consists of three text inputs for channels selection, pattern transformation, and modification. When compiled with the key binding (*ctrl+enter),* this module looks up for the active patterns in the pattern history (see below) and applies the specified parameters to the patterns depending on the channel selector. Also, the channel selector, transformer, and modifier can be grouped and stored for rapid, dramatic changes in the musical output.

Channel input expects a list of space-separated channel numbers. For example, `1 2 3` means that only the first three channels will be affected by a successful compilation of the global controls. Alternatively, `0` can be used to affect the patterns in all active channels.

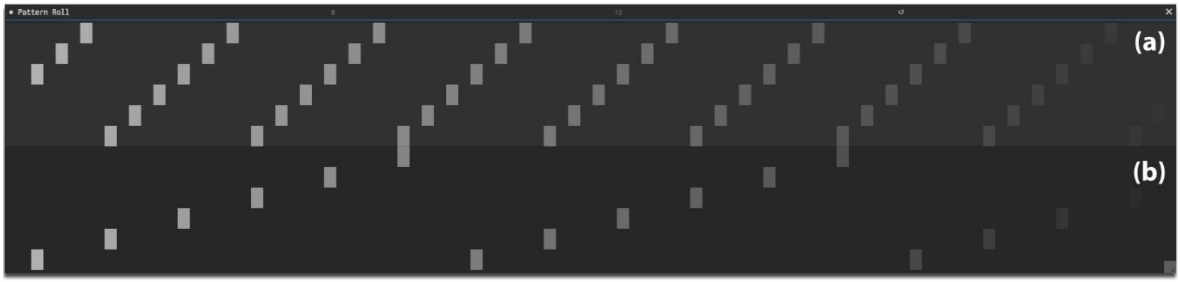
Transformer input accepts any function that transforms TidalCycles patterns. It requires a dollar sign (`$`) at the end of the text to correctly parse the patterns. Modifier input accepts all kinds of input that are relevant within the context of the language. Any input to this field needs to start with a hashtag (`#`) for correct parsing.

3.2.7 History

History module keeps track of the successfully compiled patterns on the channels. It serves as a tool to see the state of recently active channels and the results of pattern reconstructions after the evaluation of parameters and mathematical expressions. Patterns in this module serve as an input to the global controls module described above.

3.2.8 Pattern Roll

Inspired by the MIDI piano roll modules in traditional DAWs, the pattern roll module provides visual insight into active TidalCycles patterns. It operates as a playback visualization tool that ultimately relieves a portion of the cognitive load of the live-coding practice and transfers it onto the interface.




```
d1 $ slow 1 $ n (run 6) # s "bd" (a)
d2 $ slow 3 $ n (run 6) # s "bd" (b)
```

Figure 11 The pattern roll module with two active channels

It serves as a density map about the event dispatching and channel relations. The horizontal axis denotes quantized time bins that work with the global timer, and vertical shows channel relations and sample information of the patterns. The default length of the view is eight seconds and each second is quantized into twelve bins. However, both parameters are editable by the user.

Figure 11 exemplifies two active patterns visualized on the pattern roll module. The vertical axis is divided into two equal sections, one for each active channel, denoted by the differences in their background colors. In this example, we chose to visualize two similar patterns, with small variations on their speeds. The top pattern (a), visualized on the top row of the figure, is running with the regular speed, and the other (b) –on the bottom row— is running three times slower than the first one. The little white boxes denote individual note triggers of the sample **bd**, distributed over time.

The processing and storage of the individual triggers pose a computational challenge, especially with longer window durations and complicated multi-channel setups. To overcome this issue, we used a tree structure to exploit fast hierarchical data processing. From top to bottom, the levels of the tree hierarchy is as follows: the channel, sample file name, and the note number of the trigger. On the leaves, we store all trigger parameters (i.e., modifiers) and maintain statistics propagated to each node of the tree for later reference from the Graphics module.

3.2.9 Graphics

This module aims to provide a visual extension to the TidalCycles patterns by listening to the playback messages, analyzing them and feeding this information to the GLSL shaders to animate 3-dimensional figures in synchronous with the sonic tone.

The graphics engine implements the ray-marching technique in GLSL, which offers a fast rendering of mathematical surfaces in the 3-dimensional space. We chose to use the 3D version of Superformula [36], i.e., supershape, as our primary figure for its flexible parameters and vast visual outcomes. The seven parameters in the formula can result in a virtually unlimited number of unusual shapes. Combined with other mathematical structures and blending methods, it proposes a sturdy base for the visual generation. Figure 12 demonstrates two examples of visuals: Left one has only a single supershape, and right incorporates the supershape in a grander stage with more elements and textures.

TidalCycles playback data is discrete; it only reveals information about the most recent trigger. We needed to aggregate and process the data in a more sophisticated fashion to achieve a continuous animation. Therefore, we maintain what we call *evolution matrices* for each active channel and pass these four-by-four matrices to the shader. The matrix elements

are hand-picked from a larger pool of modifier list as they affect the sonic outcome of the pattern the most. Some examples of such modifiers are delta time (the time difference between the last two triggers), note number, the speed of sample playback, and the pan value.

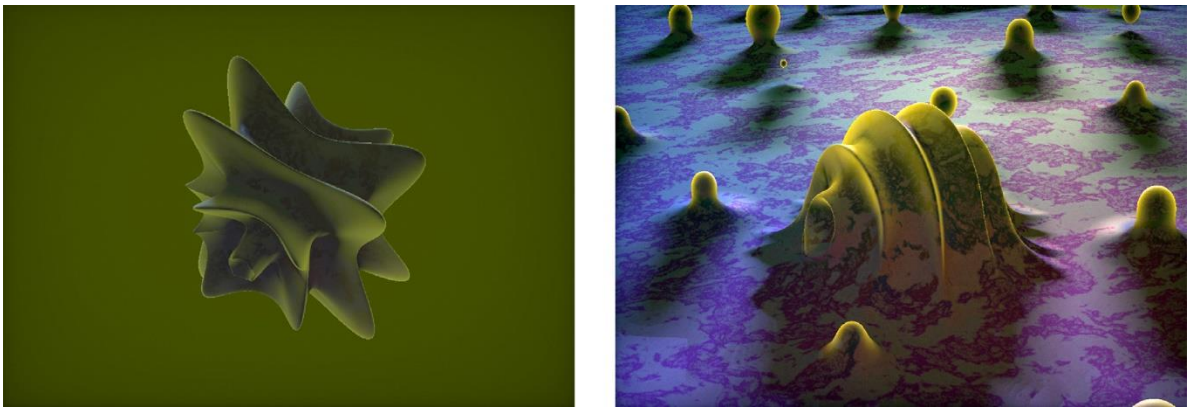


Figure 12 Two examples of the visuals on the graphics module

The matrices are initialized with the predefined default values of each element. When a sample is triggered, its modifiers are read from the processed tree (see the Pattern Roll module), and the equivalent value from the trigger multiplies the corresponding elements in the matrices. Concurrently, as the time progresses, each value continuously *decays* towards their default values. This tussle for the values of the matrix elements results in continuous modifications in initially discrete trigger parameters.

Once we pass the evolution matrices to the shader along with additional information about the current state of the scene, we utilize selected values of the first matrix for shape manipulations, the second one for determining the texture, the third one for camera movements, and the fourth one for applying screen-space effects onto the visual. The active scene information determines the external stage props that accompany the central supershape.

3.2.10 Debug Log

The nature of the live-coding practice involves the usual operations of writing the code, observing the outcomes, and desirably manipulating the existing code to evolve the result. However, sometimes this method halts with incorrect textual inputs either by mistake or some logical error, resulting in the need for debugging.

Each compiler is designed to provide human-readable error messages in case something does not work. The same thing also applies to the SuperCollider server and Haskell compilers. Siren continually listens to this information and displays them on the *log* module provided for each compiler. This way rather than digging through long messages dumped on other windows in the operating system, the performer can directly see what is wrong on the interface itself.

3.2.11 Settings

The initial setup requirements of any system can be tedious and might demand to build extended system structures behind-the-scenes. We provided a *settings* module where the user can input the config parameters and paths that are specific to their operating system to ease the process of manual path specification. This module aids the back-end to use necessary and readily-available resources on the operating system before the initial use of the environment.

3.3 Interface-level Features

3.3.1 Layout

We developed Siren with a modular layout system to provide maximum freedom to its users. Apart from the fixed Menubar on the top that holds Siren logo, timer, and server

operations, each module explained in Section 3.2 can be personalized at user's will. Every module is designed responsively–arranges its contents based on its width—to provide resizable window dimensions. The user is also free to place each module in the desired position on the screen and to change the visibility of the module when she does not need it.

We prepared multiple layouts that are designed for specific use-cases in Siren. For example; one layout holds the tracker interface and the pattern dictionary for algorithmic composition; the other maximizes console views to use the system as a regular text editor. In addition to readily available layouts, the user is free to create four custom layouts and save them for future reference.

3.3.2 Save

The system comes with two save functionalities: One is the save operations on the content level (*ctrl+s*), and the other is on the layout level (*ctrl+shift+s*). Saving the content writes all the information on the modules to local files, and the system loads these files on restart. Saving the layout works similarly on the layout of the system.

3.3.3 Context Menu

The context menu, in other words, the right-click menu, holds the abovementioned operations in a convenient location. It provides access to adding tracker channels, saving the contents and layout, and layout-based operations.

3.4 Connectivity with External Hardware and Software

We initially designed Siren to work with TidalCycles and SuperCollider, but we quickly realized it could support various other systems using communication protocols such as OSC and MIDI.

Currently, the interface of Siren is fully compatible with nanoKONTROL2 hardware controller released by Korg. It offers perfect control for Siren since it is targeted for controlling channel-wise music software. By using its eight-channel control layout, we can regulate up to eight channels on the tracker of Siren. The knob controls the rate of the channel, slider seeks the timer to a specific location, and buttons change the solo, mute and loop operations of the channel header.

Apart from the hardware connections, we experimented with using Siren in conjunction with other software that supports graphical output. By setting up an OSC connection between Siren and Processing and dedicating one channel of the tracker for the communication between the two, we governed a custom 2D shader engine written in GLSL and Processing (see Sec. 4.3.1). Figure 13 demonstrates some examples of controlling 2D shader engine using Siren.

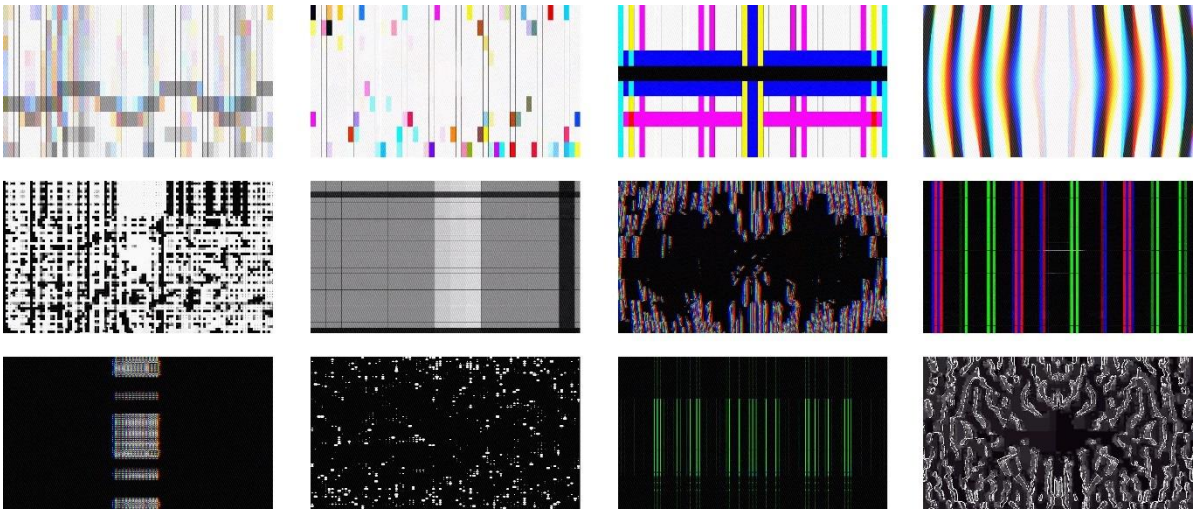


Figure 13 Some examples of visuals in Processing controlled by Siren

Another software integration was with Unity game engine to create and control virtual reality experiences. We used the OSC communication using the C# scripting. By analyzing and processing the audio playback described in Sections 3.2.8 and 3.2.9, we experimented

with 3D mesh creations and manipulation, and dynamic texturing and lighting for the virtual space.

Lastly, we have also experimented with using low-level digital signal processing chains for real-time video editing. Using the playback analysis from TidalCycles, we employed a custom Python script to work with Siren and manipulated footages in real-time based on the same signal processing methods used in TidalCycles patterns.

4 Results

4.1 Deliverables

The development of Siren resulted in several deliverables which are publicly available on the author's GitHub repository [37].

- Open-source code of Siren, along with the install and build instructions
- Various custom SuperCollider SynthDefs and additional Haskell functions for TidalCycles
- OSC connections for Processing, Unity, and Python

4.2 Community Adoption

Siren has debuted at the 5th birthday stream of the Algorave in March 2017, performed by Can Ince. The initial performance followed by several appearances in different Algoraves and a couple of performances in Turkey, United Kingdom, and the United States. The public opinion towards Siren is highly positive ever since the website of TOPLAP featured it and the crowd-funded book about electronic music instruments—*Push Turn Move* [38]—included it along with SuperCollider and TidalCycles. Several live-coders are currently experimenting with Siren, and we expect this number to grow once we publish a through documentation along with the source code.

4.3 Milestones

The implementation of Siren contains significant version changes. To eliminate performance issues along the way, we have entirely reprogrammed and renamed the software, which is previously known as *sq*.

4.3.1 sq

This early version of Siren set out as a sequencer tool for TidalCycles patterns and constructed the primary structure of the multi-channel tracker. Figure 14 demonstrates the interface of sq; the tracker on the left (9 TidalCycles channels and 6 Processing channels), a simpler version of the pattern functions in the middle, and Menubar on the right.

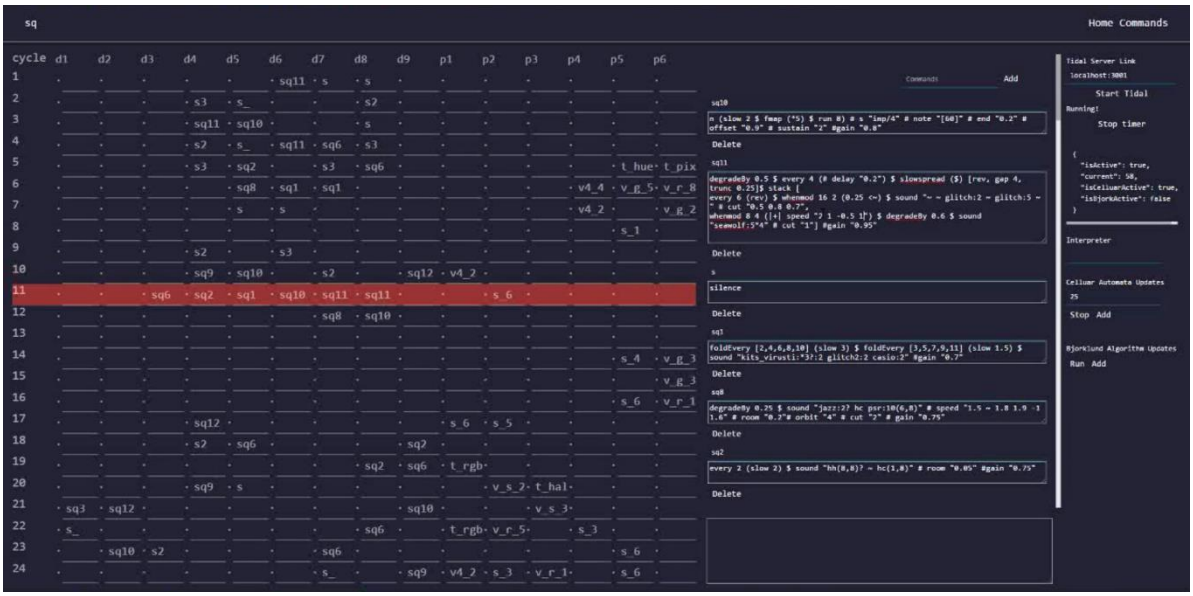


Figure 14 Interface of the previous version of Siren, sq

On top of the code compilation explained in Section 3.2.2, this version is also the first version where we had experimented with controlling an external visual system using the dedicated channels on the tracker. 6 Processing channels in the above figure (i.e., **`p1..6`**) enables controlling various parameters of a custom-made shader engine written with GLSL and Processing (see Figure 13 for some visual examples).

Another experiment in this version was to evolve contents of the tracker automatically based on some cellular automata rules. We chose Conway's Game of Life [39] for its simple logic and quick implementation. Whenever the timer loops back to the top of the tracker, we calculate the next state of the tracker by treating the function calls inside the cells as

predetermined blocks with minor randomization. Even though this process produced exciting outcomes, we abandoned this idea in the later releases since this progression of states implied a spatial relation between two adjacent channels and the channels on the tracker operate completely independent from each other.

4.3.2 Siren

We have renamed the system to Siren after reprogramming the interface from scratch. The state of the system in Chapter 3 is the latest version of the interface.

4.4 Evaluation

To formally evaluate Siren, we employ the *Cognitive Dimensions of Notations* framework which is initially introduced by Green to investigate the psychology of programming languages [40] and later adapted towards the musical user-interfaces [41] [42].

Building on Green's definitions, Nash translates the cognitive dimensions into a set of questions that can quickly inform the end-users. He uses these questions to investigate traditional paper scores, Max/MSP, and DAWs [42]. The author follows the investigation method of Nash to serve a personal evaluation of Siren in his NIME paper [43]:

- ***Virtuosity / Learnability****—How easy is it to master the notation? Where is the respective threshold for novices and ceiling for experts?*

The single first thing Siren assumes is that the performers are acquainted with programming skills, especially with TidalCycles or SuperCollider languages. This initial requirement introduces a high threshold for novices. The performers who are readily familiar with these languages will find Siren easy to learn and master. Once the user is also

familiar with the software, the creative ceiling of Siren is virtually unlimited as it augments the textual information even further whose creative possibilities are also virtually unlimited.

- ***Visibility****—How easy is it to view and find elements or parts of the music during editing?*

The transparency is one of the critical aspects in Siren as it interfaces the textual information. Channels lay out the temporal view (Sec. 3.2.2), and pattern roll serves as a visualization tool for the current playback that provides higher resolution on how each pattern unfolds over time (Sec. 3.2.8). Just like in DAWs, the distribution of the musical information over the layout may enhance the visibility if related modules are placed and resized efficiently (Sec. 3.3.1).

- ***Juxtaposability****—How easy is it to compare elements within the music?*

Among the modules in Siren, the major components that influence musical output are channels and patterns (Sec. 3.2.2 and 3.2.3, respectively). The possibility to reorder items inside these modules allows side-by-side comparisons of the musical elements. As the pattern dictionary over-populates, it may become harder to find and reorder a pattern of interest; however, the hierarchical scene structure could be used to prevent the dictionary clutter.

- ***Hard Mental Operations****—When writing music, are there difficult things to work out in your head?'*

Textual programming can be considered one-dimensional in the sense that the relationships are encoded only in text. In visual programming (as in Max/MSP), in addition to the textual input inside the elements, their placements and interconnections also matter. However, in the end, since the encoding is compiled into machine instructions regardless of

its visual representation, this dimensionality only pertains to the user experience, not to the workings of the process itself [44]. In the case of Siren, the introduction of a temporal structure, which is primarily textual but communicated as a visual component, eases the cognitive load on the user by visualizing the execution order. Also, pattern roll visualizes the playback and provides another sensory input that ultimately helps the perception of the pattern progression (Sec. 3.2.8).

- ***Hidden Dependencies****—How clear are the relationships between related elements in the notation?*

Siren uses channels and pattern dictionaries (Sec. 3.2.2 and 3.2.3, respectively) to communicate between sequencer calls and actual patterns. This simple communication presents a familiar approach to cross-reference different patterns across the system. In the case of crowded scenes, it may be harder to differentiate between similarly titled elements. However, following a clear and distinctive naming system could eventually overcome this issue.

- ***Conciseness / Diffuseness****—How concise is the notation? What is the balance between detail and overview?*

Sequencing the elements using the channel grid provides greater visibility on the progression of items in its cells (Sec. 3.2.2.3). The size of each step in the grid is virtually unlimited and thus may require for scrolling and re-scaling to peek at the bigger picture.

- ***Provisionality****—Is it possible to sketch things out and play with ideas without being too precise about the exact result?*

In Siren, every scene can be used as a sketchbook to incubate compositional ideas. As such, the volatile sketching of patterns can be executed very quickly by utilizing the console

module that acts as a traditional text editor (Sec. 3.2.4). Findings in the initial experimentation stage lay out the fundamental ideas, which are subsequently implemented in a more precise form in the composition.

- ***Secondary Notation****—How easy is it to make informal notes to capture ideas outside the formal rules of the notation?*

On the interface of Siren, each module serves a defined purpose and offers fixed tools to manipulate data. Regarding functionality, the system falls short on transforming the notation into an informal and improvisational tool yet rearranging the layout to meet specific needs yields several scenarios for experimentation (Sec. 3.3.1). Moreover, Siren supports secondary notations through a fusion of textual and visual approaches to the programming. Annotating the programming languages is still possible through detailed comments on the text body. It provides non-executable and convenient statements to realize quick notes, sketches, and prototyping.

- ***Role Expressiveness****—Is it easy to see what each part is for, in the overall format of the notation?*

Siren uses a single view that exposes desirable elements of a composition through a modular layout. Each module has a header that specifies its purpose (Sec. 3.2), however differentiating items at first glance might be puzzling. The hierarchical structure (Sec. 3.2.1) is not immediately visible on the interface but inherently constructed throughout the system.

- ***Premature Commitment****—Do edits have to be performed in a prescribed order, requiring you to plan or think ahead?*

Creation and manipulation of the composition can be performed in any order. This flexible workflow allows for musical accidents, which may ultimately turn out to be

sonically enjoyable. On the other hand, in conventional step sequencers, the edits need to be planned if the aim is to introduce, for instance, a significant breakdown in the middle of the composition. The breakdown requires precise adjustments before and after the event. Comparatively, the scene concept (Sec. 3.2.1) and global controls (Sec. 3.2.6) in Siren can be used to introduce a break or a verse. In a way, it is designed to allow mistakes which may end up fueling inspiration.

- ***Error Proneness***—*How easy is it to make annoying mistakes?*

The system, in its essence, depends on text input for naming scenes, patterns, parameters and many other aspects of the interface. It is possible to make mistakes in cross-referencing the grid entries and corresponding pattern function names. However, if the function call fails to reconstruct syntactically correct patterns, both the interface and back-end compensates the mistake without disturbing the musical output. Substantial actions, such as completely removing a scene, is guarded with a two-step action requirement: clicking the remove button and approving the selection. Also, the storage mechanism (Sec. 3.3.2) acts as a reliable safeguard for unexpected mistakes.

5 Conclusion

This document introduced Siren, a general-purpose tool for algorithmic composition and live-coding with SuperCollider and TidalCycles. It treats code as a creative primitive and augments the live-coding process with compositional tools such as a multi-channel tracker, visualization, and many more.

In Chapter 1, we list the problems and shortcomings of the field, followed by the motivations for the proposed system. Chapter 2 starts elaborating on the precursors of creative programming by briefly touching upon the history of computer music, demoscene culture, and the live-coding practice. The rest of the chapter depicts the inspirational concepts and software from the perspectives of computer science and music making. In Chapter 3, we illustrate how Siren operates by delving into the technical aspects and by explaining the inner-workings of each module on the interface. We highlight how the design objectives are embodied using these modules, and present external hardware and software that communicate with the interface. Following the methods, Chapter 4 mentions the community adoption and previous version, and the author presents a personal evaluation of the system guided by the cognitive dimensions framework.

5.1 Future Directions

There are a couple of paths that this project can lead from this state. The first feature, inspired from the modular synthesizers, would be to introduce a plug-in system where the users choose to load or offload specific functionality at will. This way, support for more live-coding languages could be downloaded as one of the plug-ins. Limiting the number of concurrently active components will prevent bloating the system.

The other improvement planned for the near future is to have more precise control on the graphics module. A system-wide integration requires providing a dedicated channel and means for messaging between the tracker and the graphics engine. Separating the interface and the graphics in two distinct computers could also be a viable solution since current configuration demands excessive computational power.

Lastly, the data which is continually produced by Siren could be used to train a neural network so that it can assist the user in the creation process. Training such a system might involve manual ranking of the data based on personal liking, or it could be performed entirely unsupervised.

6 References

[1] TOPLAP, "ManifestoDraft," 14 November 2010. [Online]. Available: https://toplap.org/wiki/ManifestoDraft. [Accessed 3 August 2018].

[2] R. Kirk and A. Hunt, Digital Sound Processing for Music and Multimedia, New York and London: CRC Press, 2013.

[3] E. S. Kuehnl, "Brief History of Computer Music," The Csound Community, 2015. [Online]. Available: http://www.csounds.com/community/computermusichistory/. [Accessed 2 09 2018].

[4] V. Scheib, T. Engell-Nielsen, S. Lehtinen, E. Haines, Taylor and Phil, "The Demo Scene," in *ACM SIGGRAPH 2002 Conference Abstracts and Applications*, San Antonio, 2002.

[5] M. Reunanen, Times of Change in the Demoscene: A Creative Community and Its Relationship with Technology, Turku: Juvenes Print, 2017.

[6] N. Collins, *Origins of live coding*, Durham: Presentation, 2014.

[7] C. Roads, "The Second STEIM Symposium on Interactive Composition in Live Electronic Music," *Computer Music Journal*, vol. 10, no. 2, pp. 44-50, 1986.

[8] TOPLAP, *A prehistory of live coding*, TOPLAP CD, 2008.

[9] TOPLAP, "TOPLAP GitHub Repository," 24 08 2018. [Online]. Available: https://github.com/toplap/awesome-livecoding. [Accessed 1 09 2018].

[10] S. Wilson, D. Cottle and N. Collins, The SuperCollider Book, Cambridge and London: MIT Press, 2011.

[11] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound,* vol. 10, no. 3, pp. 193-200, 2005.

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

[13] 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*, Gothenburg, 2014.

[14] J. Rohrhuber and A. McLean, "SuperDirt GitHub Repository," 2015. [Online]. Available: https://github.com/musikinformatik/SuperDirt. [Accessed 3 09 2018].

[15] C. Roberts and J. Kuchera-Morin, "Gibber: Live Coding Audio in the browser," in *Proceedings of 37th International Computer Music Conference*, Ljubljana, 2012.

[16] R. Bell, "An Approach to Live Algorithmic Composition using Conductive," in *Proceedings of LAC 2013*, 2013.

[17] R. Kirkbride, "FoxDot: Live coding with python and supercollider," in *Proceedings of the 2016 International Conference on Live Interfaces*, Brighton, 2016.

[18] Cycling 74, "Max Software Tools for Media," Cycling 74, [Online]. Available: https://cycling74.com/products/max/. [Accessed 3 09 2018].

[19] J. Gleick, The Information: A History, A Theory, A Flood, New York: Pantheon Books, 2011.

[20] G. Wang, "A history of programming and music," in *The Cambridge Companion to Electronic Music*, Cambridge, Cambridge University Press, 2007, pp. 55-86.

[21] K. Collins, Game sound: an introduction to the history, theory, and practice of video game music and sound design, London: MIT Press, 2008.

[22] C. Matsuoka, "Tracker History Graphing Project," 2007. [Online]. Available: http://helllabs.org/tracker-history. [Accessed 17 08 2018].

[23] C. Nash and A. Blackwell, "Liveness and Flow in Notation Use," in *New Interfaces for Musical Expression*, Ann Harbor - Michigan, 2012.

[24] C. Ince and M. Toka, "Siren: Hierarchical composition interface," in *Proceedings of 43rd International Computer Music Conference*, Shanghai, 2017.

[25] GitHub Inc., "Electron JS," GitHub Inc., [Online]. Available: https://electronjs.org. [Accessed 13 08 2018].

[26] Facebook Inc., "React JS," Facebook , [Online]. Available: https://reactjs.org. [Accessed 13 08 2018].

[27] "MobX," open source, [Online]. Available: https://mobx.js.org. [Accessed 13 08 2018].

[28] "Codemirror," [Online]. Available: https://codemirror.net. [Accessed 13 08 2018].

[29] S. Reed, "react-grid-layout," [Online]. Available: https://github.com/STRML/react-grid-layout. [Accessed 5 07 2018].

[30] C. Sattinger, "supercolliderjs," [Online]. Available: https://github.com/crucialfelix/supercolliderjs. [Accessed 5 07 2018].

[31] Ableton, "Ableton Link," [Online]. Available: https://www.ableton.com/en/link. [Accessed 5 07 2018].

[32] A. McLean, "TidaLink," [Online]. Available: https://github.com/tidalcycles/TidaLink. [Accessed 10 07 2018].

[33] G. Renaudeau, "gl-react," [Online]. Available: https://github.com/gre/gl-react. [Accessed 10 07 2018].

[34] L. Foundation, "Node.js," [Online]. Available: https://nodejs.org/en/. [Accessed 10 07 2018].

[35] "Express.js," [Online]. Available: https://expressjs.com/. [Accessed 11 07 2018].

[36] J. Gielis, "A generic geometric transformation that unifies a wide range of natural and abstract shapes," *American journal of botany,* vol. 90, no. 3, pp. 333-338, 2003.

[37] M. Toka, "Siren: GitHub Repository," 2018. [Online]. Available: https://github.com/merttoka/Siren.

[38] K. Bjørn, Push Turn Move, Copenhagen: Kim Bjørn , 2017.

[39] E. W. Weisstein, "Game of Life," MathWorld - A Wolfram Web Resource, [Online]. Available: http://mathworld.wolfram.com/GameofLife.html. [Accessed 20 08 2018].

[40] T. R. G. Green and P. M., "Usability analysis of visual programming environments: a `cognitive dimensions' framework.," *Visual Languages and Computing,* vol. 7, no. 2, pp. 131-174, 1996.

[41] A. F. Blackwell, R. T. Green and J. D. Nunn, "Cognitive dimensions and musical notation systems," in *Proceedings of International Computer Music Conference*, Berlin, 2000.

[42] C. Nash, "The cognitive dimensions of music notations," in *International Conference on Technologies for Music Notation and Representation*, Paris, 2015.

[43] M. Toka, C. Ince and M. A. Beytas, "Siren: Interface for Pattern Languages," in *New Interfaces for Musical Expression*, Blacksburg, 2018.

[44] A. McLean, "Artist-programmers and programming languages for the arts," Goldsmiths, University of London, London, 2011.