QuaverSeries: A Live Coding Environment for Music Performance Using Web Technologies

Qichao Lan
RITMO
Department of Musicology
University of Oslo
qichao.lan@imv.uio.no

Alexander Refsum Jensenius
RITMO
Department of Musicology
University of Oslo
a.r.jensenius@imv.uio.no

ABSTRACT

QuaverSeries consists of a domain-specific language and a single-page web application for collaborative live coding in music performances. Its domain-specific language borrows principles from both programming and digital interface design in its syntax rules, and hence adopts the paradigm of functional programming. The collaborative environment features the concept of 'virtual rooms', in which performers can collaborate from different locations, and the audience can watch the collaboration at the same time. Not only is the code synchronised among all the performers and online audience connected to the server, but the code executing command is also broadcast. This communication strategy, achieved by the integration of the language design and the environment design, provides a new form of interaction for web-based live coding performances.

1. INTRODUCTION

Live coding, when used in a musical context, refers to a form of performance in which the performers produce music by writing program code rather than playing physical instruments [4]. During the past decade, dozens of live coding languages have emerged.[1] These languages run in various environments, such as the desktop, the browser, and embedded systems (Raspberry Pi, BeagleBone, etc.). The number of programming languages developed for live coding can, in some ways, indicate that performers want to develop their subjective language syntaxes tailored to their musical expressions. Some examples of such new syntaxes are TidalCycles [18], ixi lang[11], Lich.js [6], and Mercury [17].

There have been some discussions about how live coding languages relate to musical instruments [3], but relatively little attention has been devoted to analysing how it is possible to 'transduce' electronic instrument knowledge to the syntax design itself. That is, what types of symbols should be used for what musical purposes, how should different elements be connected, and so on. Instead, the syntax in most live coding languages is mainly borrowed from other programming languages. For example, the use of parentheses is ubiquitous in programming languages, and it is adopted in almost every live coding language. That is the case even though live coding without the parentheses is (more) readable for humans [12].

Inheriting standard programming syntax may create difficulties for non-programmers who want to get started with live coding. We have therefore been exploring how it is possible to design a live coding syntax based on the design principles of digital musical interface design. This includes elements such as audio effect chains, sequencers, patches, and so on. The aim has been to only create a rule by 1) borrowing from digital musicians' familiarity with the digital interfaces mentioned above, or 2) reusing the syntax from existing programming languages to help the parser to work. The design principle is also aligned with the concept of *ergomimesis*, namely the 'application of work processes from one domain to another' [13]. The goal of this principle is to lower the learning curve of our language syntax, especially for non-programmers.

The second aim of our current exploration is to develop a live coding language that is usable for a larger group of people. This can be seen as part of the trend of 'musical democratisation' [8]. Our experience with running workshops for larger groups of university students or pupils in schools is that software that needs to be installed locally makes it much more difficult to get started making music quickly. We, therefore, see browser-based interfaces as the best solution for minimal setup time.

Finally, as part of our interest in exploring the blurring of roles between performers and perceivers, we have also looked at how it is possible to include the 'audience' in live coding. An online deployment makes it possible to not only share the code among performers, but it also makes it possible for the audience to easily join into the online live coding performance. This requires a delicate organisation of a stack of web technologies.

In the current paper, our main research question is:

- How can web technologies influence the music interaction between performers and the audience?

From this two sub-questions emerge:

- How can we design a live coding environment that makes the audience part of the performance? How should the environment be modified to meet this requirement?

[1] See, for example, the TOPLAP overview here: https://github.com/toplap/awesome-livecoding

Web Audio Conference WAC-2019, December 4–6, 2019, Trondheim, Norway.

- How can we use knowledge from digital musical instrument design when developing the syntax of a live coding language? What are the trade-offs that we have to make to achieve this goal?

The paper starts with a background section in which we introduce some related work. Section 3 presents the new domain-specific language, elaborating on how the parser is designed and its semantics. In Section 4, we describe the interface design and explain the communication strategies based on the Firebase real-time database. Finally, in Section 5, we discuss how the environment fits the language design, and how it experimentally changes the relation between performers and audience.

2. BACKGROUND

Many existing live coding environments are installed locally and use the SuperCollider music programming language as the sound engine [15]. With the advent of the Web Audio API, there has been a shift towards developing live coding environments with web technologies. In the following, we will reflect on these two approaches to live coding.

2.1 Live coding with SuperCollider

SuperCollider consists of a programming language called *sclang* and an integrated development environment (IDE). In the IDE, users can boot an audio server (*scsynth*) in the background, and write the code following the syntax of *sclang*. The code, when executed, will be compiled into Open Sound Control (OSC) messages, and sent to the *scsynth* server to control the music sequence. One typical workflow in SuperCollider is to define the synthesiser architecture with the keyword `SynthDef`, and then play the *Synth* in a SuperCollider music sequence (*Pattern*).

Several live coders have chosen to design their syntaxes on top of SuperCollider. For instance, TidalCycles (Tidal) is a domain-specific language written in the Haskell programming language [18]. During live coding, the Tidal code will be interpreted and sent as OSC messages to control the sound engine called *SuperDirt* running in SuperCollider. FoxDot follows a similar architecture but uses Python as the host programming language [9]. Additionally, Troop is a collaborative environment developed for both Tidal and FoxDot, which allows users at the same network to co-edit and share the code on the screen [10].

An inconvenience with the above-mentioned environments, relying on one or more programming languages in addition to SuperCollider, is that it requires several steps of installation. A more user-friendly solution, then, is Sonic Pi, which is also built on SuperCollider audio engine, but offers a single, complete installation package [1]. Even though several OSes are supported, it does not currently run on desktop Linux or Chrome OS. In our experience, this makes it less ideal for schools. And for quick introductory workshops to live coding, we find that having to rely on software installs is less than ideal. For such situations, a web-based solution is more feasible and scalable.

2.2 Web-based live coding

Web-based or browser-based live coding environments only require an up-to-date browser to get started with live coding. With the rapid progress of the Web Audio API, the sound synthesis possibilities and timing capabilities for browser-based live coding have matured quickly. Two good examples of this are the JavaScript-based Gibber environment [20], and the Lisp-style language Slang.js [21]. Although the latter currently does not support collaboration, its parser, written in Ohm.js, provides a valuable example for our development. Another inspiration for us is from EarSketch [16], a music producing environment mainly designed for normal programming education, and its use of the Firebase real-time database pointed us in the direction of a collaborative live coding solution [23].

Some other web-based environments serve as interfaces for other languages. Estuary is a system built for live coding with Tidal in browsers [19]. It has several unique features: collaboration in four different text fields, the support for both SuperCollider and the Web Audio API, and so on. Estuary makes it possible to live code together from different locations, and has been shown to work reliably in cross-continental live coding.

As can be summarised from the brief review of existing live coding environments, the programming languages, syntaxes, and interfaces are diversified. In our exploration, we have been borrowing parts from many of these when designing our syntax and environment.

3. LANGUAGE DESIGN

The syntax design of QuaverSeries is based on a functional programming paradigm.[2] The following sections will describe its syntax and how Ohm.js and Tone.js have been used to implement the parsing and semantics.

3.1 Note representation

The note representation is probably the element that is varied the most among live coding languages. QuaverSeries is based on the idea of a music sequencer. Our prototype syntax looks like this:

```
60  _62  63_64_65_  66_67_68_69
```

The *sequence* has only three elements: numbers, underscores and blank spaces. The numbers refer to MIDI notes, with 60 being 'middle C'. A blank space indicates a separation of individual *notes*, while an underscore denotes a musical *rest*.

A *sequence* will always occupy the duration of a whole note, and all the *notes* will be divided equally. To illustrate, the one-line *sequence* above will be divided into four *notes*: `60`, `_62`, `63_64_65_`, and `66_67_68_69_`, with each of them occupying a quarter note length. Each *note* can be further (equally) divided by the total number of MIDI notes and rests. On the example above, `_62` means that an eighth MIDI note 62 will be played on the off-beat, after an eighth rest. Likewise, `63_64_65_` means eighth note triplets.

As can be seen from the examples above, we create the syntax rule by referring to the musical sequencer, and add extra programmability to the syntax using the dividing algorithm invented in TidalCycles [18]. One direct influence here is that we form a left-to-right typing flow. For the sake of consistency, this flow is kept in other parts of the syntax design as well. Hence, there is no pairing symbol such as parentheses and quotation marks in the syntax.

The sequence can then be connected to a sound generating module using the double greater-than sign (`»`) which is

[2]https://github.com/chaosprint/QuaverSeries

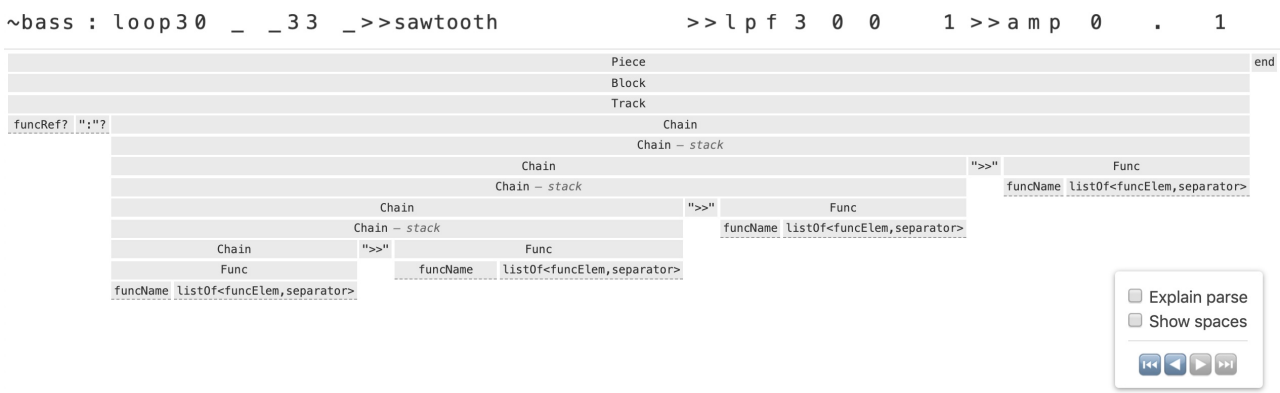

Figure 1: An example of an Ohm.js parsing tree. The hierarchy from top to bottom mainly includes: *Piece*, *Block*, *Chain*, *Func*, etc. In this example, the one-line code makes a *Piece*. This *Piece* contains one *Block*, and the *Block* is a *Track* rather than a *comment*. This *Track* can be further divided into a function reference name (*funcRef*) and a function chain (*Chain*). The function chain will be parsed using left recursion, and the semantics definition is written in JavaScript using Tone.js.

prevalent in programming languages, e.g. C++. In QuaverSeries, it indicates a signal chain flow, from left to right:

```
loop 20 20 20 20 >> membrane >> amp 0.8
```

The one-line code above will create a 'flat four' kick drum sequence using the oscillator function (`membrane`), and with an amplitude of 0.8 (`amp`). This syntax style naturally leads to the choice of using a functional programming paradigm.

3.2 Functional programming

Functional programming languages are prevalent in musical applications, such as the above mentioned Tidal. Overtone is another functional live coding language, following the syntax of Clojure, a dialect of Lisp language [2]. In a functional programming language, everything is a function in the mathematical sense. For example, for the function `y = f(x)`, `x` refers to the *independent variable*, `y` refers to the *dependent variable*, and `f` refers to the transformation. In a similar manner, to express an electronic music signal flow, we can, for example, write the pseudo code:

```
(lpf (square 440 1) 1000 1)
```

Here the `square` function, followed by `440` and `1` means a square wave oscillator function with two parameters as independent variables (inputs): frequency and amplitude. In this example, `lfp` is the abbreviation of *low-pass filter*, and refers to another function that receives three parameters: the audio signal for filtering, the cut-off frequency, and the Q-value. A pair of parentheses is here used to wrap the function that is passed to the next function as its input. This is typical in functional programming languages such as Lisp.

The same synthesis architecture can be rewritten in an object-oriented programming style as:

```
osc = Square(440, 0.8)
osc.connect(LPF(30, 1500))
```

Here `Square` refers to a *class* with a *constructor*. When the constructor is called, an *instance* will be created, and we can save it as in a variable called `osc`. The instance can call its *methods* predefined in the class.

Both functional and object-oriented programming paradigms have both pros and cons. But since we have chosen to start our syntax with a sequencer as the main defining element, we have found it most practical to use functional programming in the syntax design.

In the one-line functional programming code mentioned above, `loop`, followed by a sequence of MIDI numbers is the first function. When followed by `membrane`, the function on the left should become the frequency parameter of `membrane`, with an implicit conversion from MIDI notes to frequency. The `amp` is a function that sends the audio signal generated by the function chain to the audio interface, with a sound level scaling of 0.8. The equivalent Lisp-style would be:

```
(amp (membrane (loop 20 20 20 20) 0.8))
```

3.3 Parsing

The parser in QuaverSeries is built from scratch with the help of Ohm.js. This requires to first program in its domain-specific language, describing how the parser should act (see Appendix B). The parser will generate a parsing tree, identifying the structure of the code (see Figure 1). An example of the QuaverSeries syntax may help to explain how it works:

```
bpm 120

~bass: loop 30 _ _33 _
>> sawtooth >> adsr 0.04 0.3 0 _
>> lpf ~cutoff_freq 1
>> amp 0.1

~cutoff_freq: lfo 8 300 3000
```

The whole *Piece* in this example can be divided into different *Block(s)*, with each *Block* containing at least one function separated by an empty line. The first line is a function for setting the tempo of the piece (120 beats per minute). All the function names are typed in lower case, with an optional underscore in between.

Each function is typically followed by the function elements (*funcElem*). For example, the `adsr` function has four parameters, i.e. the *attack*, *decay*, *sustain* and *release* of an audio envelop. The usage of the underscore is flexible. Apart from its usage in the note representation (to denote a rest), an underscore can also be used as a Python-style placeholder

to keep the default value of a parameter. For instance, the `adsr` function has a value of zero for the *sustain* parameter, which means that there is no need to write the *release* value. Hence, we can use an underscore to represent the release.

The second block demonstrates a concept called *reference*. With a tilde-prefix (~), a function name becomes a reference that can link two signal chains with one signal chain modulating a parameter of the other. In the example above, the cut-off frequency of the low-pass filter is modulated by a low-frequency oscillator (`lfo`). Hence, to keep the consistency, it is suggested users add a reference at the beginning of every function chain.

3.4 Semantics

The semantics part of QuaverSeries defines how the code should be executed after being parsed. In Ohm.js, the parsing and semantics definitions are separated. Thus after the parser reads through the code and identifies several valid functions, Ohm.js needs further instructions on how to deal with these functions. For instance, when the parser detects a number, the parser will return the number as a string character. It is therefore necessary to write the semantic action as a JavaScript function that converts the string to a float in JavaScript, so that it can be used for numerical operations.

The semantics definition of QuaverSeries is written with Tone.js, a JavaScript audio and sound synthesis library based on the Web Audio API [14]. Currently, the functions are categorised into three parts: *control*, *effect* and *synth*. In the semantics definition, each function is organised into different tracks. Each track has its attributes, including *note*, *synth*, and *effect*.

Once a `run` message is received, the parser will read through the whole page, and convert every function to Tone.js code based on the semantics definition. For instance, when the `loop` function is detected, a Tone.js *Sequence* instance will be created. Likewise, if a `synth` function is identified, a Tone.js *Synth* instance will be created. If audio effects are found, the relevant Tone.js effect instances can be created. Finally when the `amp` is detected, the `connect` method of the *Synth* instance will be called to connect all the effects, with the amplifier (`Tone.Master`) at the end of the effect list.

As a summary, when the `run` command is given, the parser will read through the whole page and identify the functions. Next, semantics action will be executed by constructing Tone.js instances and calling their methods. The `update` command also reads the whole page, and updates each node that is playing, although it will first be effective at the beginning of the next bar.

4. ENVIRONMENT DESIGN

Collaboration has been an important motivation when developing the Quaverseries live coding environment.[3] The aim is to create a web application that live coders can use to collaborate in different *virtual rooms*, and where the audience can go to a particular room to watch an ongoing performance, albeit with a different level of access (see Figure 2).

4.1 Collaboration support

[3]https://quaverseries.web.app

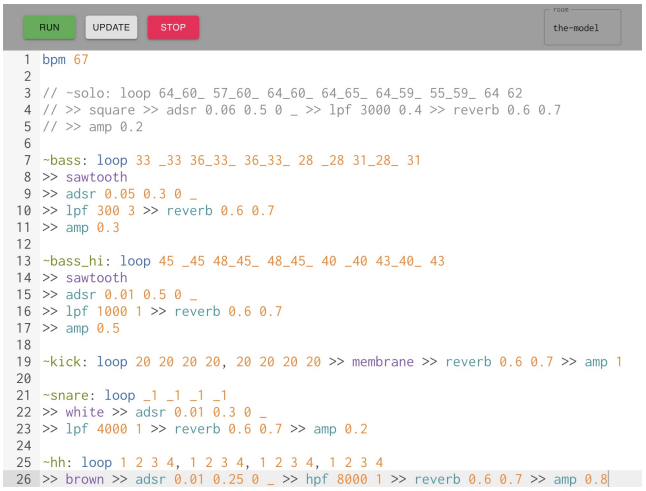

Figure 2: The QuaverSeries interface prototype. The syntax highlight has been implemented as a Ace editor theme. The buttons (`Run`, `Update` and `Stop`) are mapped to the keyboard shortcuts `Command/CTRL + Enter`, `Shift + Enter`, and `Command/CTRL + Period(.)`, respectively. The keyboard shortcut `Command/CTRL + Slash(/)` is for commenting out lines of code, which can be useful for muting a track during the performance.

Tools and algorithms such as Firebase and Operational Transformation have made the implementation of real-time code sharing much more approachable than it was only some years ago [5]. Firepad is an open-source tool that mainly uses Firebase realtime database and Operational Transformation algorithm. Thus, it provides a solution for synchronising code and sharing the cursor position between clients. In QuaverSeries, Firepad is used to share the code, while a customised strategy is designed to broadcast the related `run` and `update` commands to every client connecting to the database, including both the performers' and the audience members' clients. In this way, a live coder can control the sound running in all the browser clients. This is a similar strategy to what can be found in the Hydra synth, an environment developed for sharing visuals in the browser [7].

In the server database, two entries are storing the states of the `run` and `update`. Hence, once a user sends the `run` command by clicking the button or using the keyboard shortcut, the value of the entry `run` in the database will be set to the Boolean value `true`. As each client connecting to the database has its monitoring function for the value, once the Boolean value `true` is detected, each client will execute a relevant handling function. This function will do two things: 1) execute the code in the editor, 2) set the `run` entry back to the Boolean value `false`. Here is an example to illustrate this in pseudo code:

```
# the server
if Server get "run":
    send "run" to every client

# the client
if Client get "run":
    execute Music Code

# the interface
```

```
if Button "run" is pressed:
    sent "run" to Server
```

The principle of `update` is almost the same as `run`. The only difference is that `update` is used to renew the piece while the music is already on, that is, to calculate the current time and schedule what to play from the beginning of the next bar.

How does the system work in terms of stability? Fortunately, the transmission of only code between clients makes it possible to run the system over connections with very limited bandwidth. Furthermore, since the system is based on looped sequences, and an updating strategy per bar, allows for a considerable network delay without necessarily influencing the final musical result. This system design can, of course, be problematic if an `update` message is sent at the end of a bar. Still, the worst-case scenario is a one-bar offset among different locations. In our real-world testing so far, however, this has not been a problem in practice.

4.2 Live coding democratisation

Musical democratisation—in the sense of making music available to larger audiences—has been a growing trend ever since the invention of the phonograph [22]. Up until now, live coding has been an activity practised by relatively few. This is not only because it has been technically difficult to get started, but also because the community has been comparably small, and access to venues has been limited. Web-based live coding may help to address both of these problems, at the same time making it easier to provide access for online performances to get started.

QuaverSeries is a novel music streaming solution in that it does not stream audio or video but rather focuses on *code streaming*. Thus, instead of watching the audio/video of a performer's screen, the audience can enter a virtual room, watch new code appear on the screen, and have the musical sound rendered locally in the user's own browser. The audience can unidirectionally receive the `run` and `update` message from the server. This makes it possible to stop the rendering of music in the local browser at any time without influencing any other instances running on the machines of other performers or audience members.

The main difference between the performer and audience modes is that in the latter the code is not editable. Technologically speaking, though, every audience has the complete instrument locally, with the performers triggering the code. Thus every audience member could be seen as a collaborator and parttaker in the musicking.

5. CONCLUSION

In this paper we have presented the three parts of QuaverSeries: 1) a new domain-specific language, 2) an interface to edit and run the code, and 3) a new way of collaborating using 'virtual rooms.'

The environment draws extensively on new web technologies, utilising the power of local rendering in the user's browser thanks to the advance of the Web Audio API. This makes it possible to easily and quickly share live coding with lots of people, hence allowing the audience to become 'active participants' in a live coding session. Sharing only code between users makes it possible to create a collaborative environment with low network bandwidth. To minimise the risk of delay in the network, we have employed two strategies: 1) transmitting only code, and 2) calculating the updating based on musical bars. The trade-off of this approach, however, is that it also limits the musical possibilities of the system.

At the moment, the system is based on looped sequences only, resulting in beat-based music. Still, we have found that it is possible to create fairly complex musical results by adding multiple layers in the code. In future development, our first priority is to explore how it is possible for the audience to participate more in online performances. One approach is to build a chatting system in which the audience can write their own code, and propose this code to the performers. It would be up to the performer whether to accept the proposed code or not, similar to the way a *fork* works in the git version control system (such as used on GitHub).

QuaverSeries is currently at a prototype stage. It is fully functional, and we have explored it in a number of jam sessions. The aim now is to test it more in different performance contexts. We also plan to perform a more systematic user study, to understand more about how it works for beginning live coders.

6. ACKNOWLEDGMENTS

This work was partially supported by the Research Council of Norway through its Centres of Excellence scheme, project number 262762 and by NordForsk's Nordic University Hub Nordic Sound and Music Computing Network NordicSMC, project number 86892.

7. REFERENCES

[1] S. Aaron. Sonic pi–performance in education, technology and art. *International Journal of Performance Arts and Digital Media*, 12(2):171–178, 2016.

[2] S. Aaron and A. F. Blackwell. From sonic pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 35–46. ACM, 2013.

[3] A. F. Blackwell and N. Collins. The programming language as a musical instrument. In *PPIG*, page 11, 2005.

[4] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised sound*, 8(3):321–330, 2003.

[5] C. A. Ellis and C. Sun. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. Citeseer, 1998.

[6] T. Hoogland. Mercury: a live coding environment focussed on quick expression for composing, performing and communicating. 2019.

[7] O. Jack. Livecoding networked visuals in the browser. https://github.com/ojack/hydra, 2019.

[8] D. Kim-Boyle. Network musics: Play, engagement and the democratization of performance. *Contemporary Music Review*, 28(4-5):363–375, 2009.

[9] R. Kirkbride. Foxdot: Live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*, 2016.

[10] R. Kirkbride. Troop: A collaborative tool for live coding. In *Proceedings of the 14th Sound and Music Computing Conference*, pages 104–9, 2017.

[11] T. Magnusson. ixi lang: a supercollider parasite for live coding. In *Proceedings of International Computer Music Conference 2011*, pages 503–506. Michigan Publishing, 2011.

[12] T. Magnusson. Code scores in live coding practice. In *TENOR 2015: International Conference on Technologies for Music Notation and Representation*, volume 1, pages 134–139. Institut de Recherche en Musicologie, 2015.

[13] T. Magnusson. *Sonic writing: technologies of material, symbolic, and signal inscriptions*. Bloomsbury Academic, 2019.

[14] Y. Mann. Interactive music with tone. js. In *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

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

[16] S. McCoid, J. Freeman, B. Magerko, C. Michaud, T. Jenkins, T. Mcklin, and H. Kan. Earsketch: An integrated approach to teaching introductory computer music. *Organised Sound*, 18(2):146–160, 2013.

[17] C. McKinney. Quick live coding collaboration in the web browser. In *NIME*, pages 379–382, 2014.

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

[19] D. Ogborn, J. Beverley, L. N. del Angel, E. Tsabary, and A. McLean. Estuary: Browser-based collaborative projectional live coding of musical patterns. In *International Conference on Live Coding (ICLC) 2017*, 2017.

[20] C. Roberts, G. Wakefield, M. Wright, and J. Kuchera-Morin. Designing musical instruments for the browser. *Computer Music Journal*, 39(1):27–40, 2015.

[21] K. Stetz. Slang: An audio programming language built in js. https://github.com/kylestetz/slang, 2018.

[22] T. D. Taylor. The commodification of music at the dawn of the era of "mechanical music". *Ethnomusicology*, 51(2):281–305, 2007.

[23] A. Xambó, P. Shah, G. Roma, J. Freeman, and B. Magerko. Turn-taking and chatting in collaborative music live coding. In *Proceedings of the 12th International Audio Mostly Conference on Augmented and Participatory Sound and Music Experiences*, page 24. ACM, 2017.

APPENDIX

A. TECHNICAL STACK

The following table lists the main dependencies of QuaverSeries, and their current version number.

Dependencies	Version
React.js	16.2.8
Ohm.js	0.15.0
Tone.js	13.4.9
Material-ui	1.0.0
Firebase	7.0.0
Firepad	1.5.3
Ace	1.4.6

B. PARSER

This is how the parser of QuaverSeries is currently set up.

```
Quaver {

    Piece = Piece #"\\n"? #"\\n"? #"\\n"?
        Block --stack
    | Block

    Block = comment | Track

    comment = "//" c+

    c = ~"\\n" any

    Track = funcRef? ":"? Chain

    Chain = Chain ">>" Func -- stack
    | Func

    Func =  funcName listOf<funcElem,
        separator>

    funcElem = para | funcRef

    para = para subPara -- combine
    | subPara

    subPara = number | "_"

    number = "-"? digit* "." digit+ --
        fullFloat
    | "-"? digit "." -- dot
        | "-"? digit+ -- int

    funcRef = "~" validName

    funcName = validName

    validName = listOf<letter+, "_">

    separator = ","? space
}
```