

# LiveLily: An Expressive Live Sequencing and Live Scoring System Through Live Coding With the Lilypond Language

Alexandros Drymonitis  
PhD, Independent scholar  
alexdrymonitis@gmail.com

## ABSTRACT

LiveLily is an open-source system for live sequencing and live scoring through live coding in a subset of the Lilypond language. It is written in openFrameworks and consists of four distinct parts, the text editor, the language parser, the sequencer, and the music score. It supports the MIDI and OSC protocols to communicate the sequencer with other software or hardware, as LiveLily does not produce any sound. It can be combined with audio synthesis software that supports OSC, like Pure Data, SuperCollider, and others, or hardware synthesizers that support MIDI. This way, the users can create their sounds in another, audio-complete framework or device, and use LiveLily to control their music.

LiveLily can also be used as a live scoring system to write music scores for acoustic instruments live. This feature can be combined with its live sequencing capabilities, so acoustic instruments can be combined with live electronics. Both live scoring and live sequencing in LiveLily provide expressiveness to a great extent, as many musical gestures can be included either in the score or the sequencer. Such gestures include dynamics, articulation, and arbitrary text that can be interpreted in any desired way, much like the way Western-music notation scores are written.

## Author Keywords

live coding, live sequencing, live scoring, Lilypond

## CCS Concepts

• **Applied computing** → **Sound and music computing**; Performing arts;

## 1. INTRODUCTION

Expressiveness in live sequencing either through software or hardware is an issue that has been approached from various points of view. In a different strand, appreciation of computer code, and coding ease from the perspective of the coder in live coding sessions, have been addressed by

many languages built especially for this artistic practice. When creating loop-based music through live coding, these two issues can co-exist in a single practice. The notion of expressiveness differs among the various approaches to hardware and software sequencers and their developers. On the other hand, live coding language developers that target loop-based music seem to have a rather common understanding of the necessities that need to be addressed from the perspective of code appreciation and ease and speed of coding [18, 1, 19].

According to Chandra, programming languages are often regarded as esoteric and mystical [8]. Programming languages developed with live coding in mind, usually provide a high level of abstraction, so that coders can free themselves from the mundane activity of writing their synthesizers from scratch, and a musical jargon that connects the music produced by the coded algorithms to the code from its textual perspective. In such languages, the speed of loop definitions is closely tied to linguistic expression that closely relates to natural languages rather than computer code [18].

In the context of live sequencing loop-based music, the variety in approaches and what is recognised as a problem seems to be greater than that of live coding. From novel tunings [21] to stacking melodic lines [7], or from spinning sequencers [3] and non-linear dodecahedrons [14] to tangible sequencers based on ancient abacuses [15], both software and hardware sequencer developers seem to each have their own perspective and anxieties. It seems that expressiveness and flexibility are more subjective than the notion of computer code appreciation.

In an effort to create a tool for loop-based music through live coding, inspired by Csound's paradigm of the separation of the score from the "orchestra" [17], LiveLily was developed. This is a system written in openFrameworks (OF) that utilises a modified subset of the Lilypond language [20] with added syntactic sugar, to enable expressive and fast live sequencing and live scoring through live coding. It does not create sound, but communicates via MIDI or OSC with other software or hardware, to control sound created with audio software or hardware synthesizers. The Lilypond language was chosen for a few reasons. First of all, it uses the Dutch system for naming notes – C, D, E, F, G, A, B – and numbers for note durations expressed as denominators – 4 for a quarter note, 8 for an eighth note, etc. Western music jargon is used for other features like "clef" and "tempo". From a musical perspective, the nomenclature of this nature seems to me as a logical, yet rare approach in musical live coding.

On certain occasions, the visual aspect of a live music performance plays a role in the experience of the spectator. Although research in this field usually focuses on the performer as the visual aspect of the performance [4, 22], I believe that an animated score that follows the sequencer



Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). Copyright remains with the author(s).

```

1 \score show
2 \score animate showbeat
3
4 \insts synth bass cymbal beat
5 \bass \clef bass
6 \cymbal \clef perc
7 \beat \clef perc
8
9 \bar 1 {
10 \synth [c' ees'>8\fa^"no rev"*2 d'4-_- "rev"]*2
11 \bass e,2\fa> bes, \mp
12 \cymbal [c''16*2 r']*5 c''
13 \beat f'4\fa*4
14 }
15
16 \bar 2 {
17 \synth f'2( ees'4) d'
18 \bass e,2\fa> bes, \mp
19 \cymbal [c''16*2 r']*5 c''
20 \beat f'4\fa*4
21 }
22
23 \play
24
25 \loop aloop {1*2 2} \alooop
26

```



Figure 1: A LiveLily session.

can provide an extra element that contributes to the spectator’s experience. Adding to the appreciation of computer code mentioned above, an animated score can potentially provide a more approachable spectacle, where the spectator can focus on either the code or the music score, depending on what they feel more comfortable with.

The textual nature of this system serves the live coding approach well. In contrast to the belief of the computer keyboard not being great for [embodied] expression [16], the LiveLily language provides an extended array of expressive musical gestures, like dynamics, glissandi, articulation symbols, and arbitrary text. These gestures are embedded both in the score and the sequencer and are communicated to the audio-generating software or hardware. Their interpretation is up to the user of this software or hardware, or the acoustic instrument performer. LiveLily is designed in a way that enables the combination of live sequencing with live scoring, or the isolation of each, providing a flexible setup for electronic, acoustic, or electroacoustic performances. This is an open-source project, hosted on <https://github.com/alexdrymonitis/LiveLily>.

## 2. RELATED WORK

Precht et al. approach the MIDI sequencer from the perspective of novel tunings [21]. In this context, LiveLily provides quarter-tone tunings, using standard Lilypond syntax. The bespoke language of the Mondrian project bares similarities to LiveLily in the context of coding a sequencer [7]. The Mondrian syntax though seems to be much more abstract and cryptic, in a musical context. The author of this project states that Lilypond offers some of the structures of Mondrian, but it is geared toward typesetting. LiveLily though seems to be capable of offering structures developed with Mondrian, as it offers an easy way to reuse code chunks and build sequences employing stacking. From a live coding perspective, languages like Sonic Pi [1] and Tidal Cycles [19] are related to LiveLily in the context of fast music structuring and simplicity of the language.

LiveLily though is also a live scoring system. In this context, works like INScore [12], Maxscore [13], the bach family of Max objects [2], and Open Music [6] are related. Maxscore and the bach objects are part of the Max environment, so they are different from LiveLily by nature since they belong to the visual programming paradigm. They are also constrained to macOS and Windows, since Max does not

run on Linux. Open Music, a computer-aided composition and live scoring system, is also a visual programming environment, but it runs on all major Operating Systems (OS), including Linux. LiveLily is written in OF, so it also runs on all major OSes. Open Music and INScore are open-source projects, like LiveLily.

## 3. DEVELOPMENT STAGES AND CONSIDERATIONS

The initial stages of the development of LiveLily included researching existing live sequencing software. I had already decided that I wanted to use the Lilypond language, so this stage did not refer so much to the possible language intricacies and how to tackle them, but other software would help me decide how to approach the parser and the sequencing mechanism. Through this research, I realised that what needed to be decided was whether LiveLily would come as a plugin for an existing editor, or whether a bespoke editor would be embedded in the LiveLily software.

I opted for the second choice because programmers typically have their editor of choice, which means that I would have to build packages for many editors, if I wanted to share my project. Added to that, some editors like Vim or Emacs have a deep learning curve, and if I made a plugin for one of these editors only, many users would possibly get discouraged from using LiveLily.

The first framework I checked for building the editor was PyQt, a Python Qt module for building GUIs. An issue I encountered was PyQt’s behaviour with different keystrokes. Being inspired by the keyboard shortcuts used by the Hydrogen package for the Atom editor, which enables the user to write Python code interactively, I wanted to use Ctl+Return for executing a command, or Shift+Return for executing and moving the cursor one line below. PyQt was not updating the line numbering properly when hitting Shift+Return, so I had to look elsewhere. OF was the second choice, because of its intuitive set of commands, its flexibility, and its helpful community. It proved to be a good choice as it enabled me to write code for all the components of LiveLily, maintaining code integrity. As I started developing the LiveLily editor, the live scoring idea occurred to me. This strengthened my idea of developing an editor specifically for LiveLily and not developing a plugin for another editor, mainly because I wanted the editor and the score to be

integrated into the same GUI frame.

Once the bespoke editor started behaving closely to how I wanted it to behave, I started testing the sequencing mechanism. OF is designed mainly for visuals, so its main thread, where all the OpenGL drawing commands happen, is bound to the framerate. That posed a limitation to the sequencer, as high tempi combined with short note durations, like 32nds, would require a very high framerate for the program to be on time for every beat, plus jitter was very likely to occur, due to the slow refresh rate. Being a long-time Pure Data (Pd) user, I attempted to include Pd into OF with the ofxPd addon, as Pd has a sub-millisecond accuracy, and would therefore provide the necessary resolution for high tempi. Since I did not want to spread the project among many programming languages, I compared a Pd version of the sequencer with an OF version that run in a different thread, with a refresh rate of one millisecond. The results were similar, so I opted for the OF version.

Due to LiveLily using a modified version of a subset of the Lilypond language, an initial idea was to use the Lilypond compiler to create the scores. In the past, I have realised a project where Lilypond compiled the score that was loaded as an image in an OF program [10], but in a true live scoring context, this takes a substantial amount of time. Also, detecting specific notes, which is necessary for an animated score, is much more difficult with an imported image than it is with native OF primitive shapes and font symbols, loaded into the OF program. This led me to code the live scoring part from scratch. I first tried to create everything with primitive shapes like circles, lines, ellipses, and other native OF classes, but that proved to be too complicated. Instead, I used the *Sonata* font that includes Western notation symbols. I combined it with lines for the note stems, glissandi, and the staff, and curves for slurs.

## 4. SYSTEM ARCHITECTURE

The LiveLily system can be broken down into four components: the text editor, the language parser, the sequencer, and the interactive score. All these components are written in OF and come bundled in one OF app. The following subsections discuss each component separately.

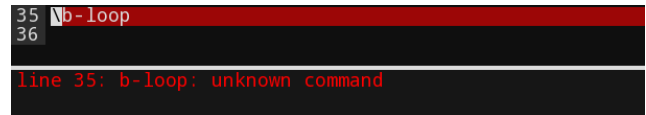
### 4.1 The Text Editor

The LiveLily text editor simulates a text editor running in a terminal window, like Vim or Nano. No mouse interactions are possible and the user navigates with the arrow keys. Line numbering is included, as well as code chunk selection for easier execution of music bars. It is possible to split the editor window both horizontally and vertically and end up with more than one editor. All editors though are in the same shell, and a command executed in any editor will affect the sequencer and the score.

It is possible to receive text via OSC, by typing the command `\fromosc` in an editor. This way, external programs can control LiveLily remotely. If more than one editor needs to receive OSC messages, the `\fromosc` command takes one argument, the OSC address to receive messages in, otherwise, the default `“/livelily”` address is used. Internally, this address is expanded to two addresses, namely, `“/livelily/press”` and `“/livelily/release”`, which are the addresses the client has to use, to separate a key press from a key release. The same applies to custom addresses, where the `“/press”` and `“/release”` part must not be included in the `\fromosc` command argument.

The text editor includes a traceback field, at the bottom of its window. In case an executed command produces an

error, the traceback is printed in that area and the line that produced the error is colour highlighted in red. If the line numbering changes by erasing or adding text above the colour highlighted line, the traceback line numbering is updated and the colour highlighting moves along with the rest of the editor. Figure 2 shows the traceback of an error that occurs when trying to call a loop that has not yet been defined.



```
35 \b-loop
36
Line 35: b-loop: unknown command
```

Figure 2: The traceback of an unknown command.

### 4.2 The Language and the Parser

Parsing LiveLily commands is based on the backslash, similar to Lilypond. All commands start with a backslash, and if an argument is required, the command is followed by it. The following line is the subcommand to animate the score and show the beat as a pulsating semi-transparent rectangle on top of the score.

```
\score animate showbeat
```

All instruments of a session must be defined before the user starts writing melodic lines. The lines below initialise four instruments and set the clef of one of them to bass, using the `“bass”` argument to the `\clef` command.

```
\inst synth bass cymbal beat
\bass \clef bass
```

A bar of music is defined with the `\bar` command, followed by a name given to the bar, and the bar contents inside curly brackets. The following chunk defines one bar named `“2”`, assuming a bar named `“1”` has already been defined.

```
\bar 2 {
  \beat f'8 c'' <f' a''> c'' f' c'' <f' a''> c''
  \rest 1
}
```

The `\rest` command states that all instruments that have not had a new melodic line defined in this bar, will copy their lines from the bar with the name of the argument to this command. To loop two or more bars, the `\loop` command must be invoked. The following line loops bars 1 and 2.

```
\loop a-loop {1 2}
```

Loops must also be named. The names for bars and loops are arbitrary. If the sequencer has not started yet, the last defined bar or loop is the one that will be played, as soon as the sequencer starts. Once it has started, to play a bar or loop, after defining it, it must be called with its name as a command. To play the loop in the line above, the user must type `\a-loop`. To play the bar with the name `“2”`, the user must type `\2`. The bar or loop chosen to be played will start looping once the current bar or loop ends. It is also possible to create loops from other loops, or with combinations of loops and bars, like in the line below. It is also possible to call a new loop in the same line that defines it, as in the line below

```
\loop b-loop {a-loop 2} \b-loop
```

To send a line or chunk to the parser so that it is executed, the user must hit Ctl+Return or Shift+Return. In the latter case, the line or chunk will be executed, and the cursor will move to the next line with text, or one line below if there is no more text after the line or chunk that is executed. When the cursor is placed on top of any of two paired curly brackets of a chunk, or inside such a chunk, the user can execute the whole chunk in a single shot.

### 4.2.1 Syntactic Sugar

LiveLily adds syntactic sugar to enable fast typing of bars and loops. The following bar defines a line with four quarter-note Fs played fortissimo for the “beat” instrument, and a chord played forte with an eight-note middle C and an E flat above played twice, and then a single quarter-note D above middle C played once. This whole pattern will be repeated twice. A version of this bar for the “synth” instrument, including text and articulation symbols is shown in figure 3.

```
\bar 3 {
  \beat f'4\ff*4
  \synth [<c' ees'>8\f*2 d'4]*2
}
```

In Lilypond, the square brackets are used for manual beaming of notes, but in LiveLily they are used to group notes so they can be repeated with the multiplication feature, as it is shown above. This syntactic sugar applies to loop definitions too. The following line creates a loop where bar “2” will be played three times and bar “3” once.

```
\loop c-loop {2*3 3}
```

## 4.3 The Sequencer

As with all components of this system, the sequencer is also written in OF. To escape from being bound to the framerate, as the main OF thread and all of the OpenGL drawing commands happen on a framerate basis, the sequencer runs in a different thread using OF’s `ofThread()` class. By calling the `waitNext()` method of the `ofTimer()` class, the subthread of the sequencer saves CPU since this method sleeps the thread it runs in, for a given time interval. The clock of this thread is on a one-millisecond frequency, for timing accuracy. It is possible to include the sequencer in the main thread of the program but to avoid jitter, a very high framerate is required, and that raises the CPU significantly.

### 4.3.1 The OSC Version

The sequencer sends OSC messages with the following information: dynamics, articulation, arbitrary text, pitch, and duration. The OSC addresses consist of the name of the instrument and one of the following strings: “dynamics”, “articulation”, “text”, “note”, and “duration”. For the example above, the OSC address for the duration of the “beat” instrument would be “/beat/duration”. If no IP address for an instrument is specified, the OSC messages will be sent to the local host, 127.0.0.1. To set an IP for an instrument, so its messages are sent to a remote computer, the following command must be executed, with the correct IP address as the argument to the `\ip` command. If a port other than the default needs to be set, it is written in the same line after the IP address, or in a separate line, as an argument to the `\ip` command. The default port is 1234.

```
\beat \ip 192.168.100.20
```

The dynamics are expressed in sheet music terms, from pianississisimo (*ppp*) to fortississisimo (*fff*). These symbols are translated to decibel values in the range from 60 (*ppp*) to 100 (*fff*). The following dynamics are possible: *ppp*, *pp*, *p*, *mp*, *mf*, *f*, *ff*, *fff*. These symbols are mapped to the following dB values: 60, 66.6, 73.3, 77.7, 82.2, 86.6, 93.3, 100.

The available articulations are marcato, trill, tenuto, staccatissimo, accented note, and staccato. These are symbolised with standard Western sheet music notation in the score, and in the LiveLily language, they are expressed with the same symbols used in Lilypond. They are transferred over OSC as strings. The same applies to text written above or below a note, using the caret or underscore symbol, borrowed from the Lilypond syntax. The user is free to interpret the articulation and arbitrary text in any way (s)he likes. The lines below creates the score shown in figure 3. Repeating dynamics symbols are omitted within the same bar, hence, the score in figure 3 displays the *f* symbol only once. In this example, “rev” stands for “reverb”, and “no rev” for “no reverb”.

```
\bar 1 {
  \synth [<c' ees'>8\f^'no rev'*2 d'4-.'rev']*2
}
```



Figure 3: A bar with text and the staccato symbol.

Pitch is sent as MIDI note values, with quarter-tone accidentals expressed as half values, for example, 60.5 for middle C a quarter-tone high. The duration is expressed in milliseconds. Notes can also be slurred, in which case the total duration of all slurred notes is sent once, and the rest of the durations are zeroed and dropped from the OSC stream. Both the pitch and dynamics are sent as lists so that glissandi and crescendi and diminuendi are possible. For glissandi, the list includes the MIDI note value and the current duration. It is possible to play chords, in which case the list will contain as many MIDI note values as the notes in the current step, and the current duration once, since all notes in a chord can only have the same duration. To express crescendi and diminuendi, the dynamics are also sent as lists, with the target dynamic and the ramp duration. The user must take care to create a portamento with these values in the software that creates the sound.

### 4.3.2 The MIDI version

Almost all information that is transferred via OSC can be communicated with the MIDI protocol too. To choose an output MIDI port, the user must call the following command.

```
\listmidiports
```

The available MIDI ports will be printed below this line. A MIDI port can be chosen with the following command, with the correct port number, depending on the output of the previous command.

```
\openmidiport 0
```

Once a MIDI port is open, each instrument that will send MIDI values instead of OSC, should be assigned its own MIDI channel. This is done with the following command, assuming an instrument named “synth” has been created.

```
\synth \midichan 1
```

The command above will set an instrument to MIDI mode. When in this mode, the dynamics of this instrument are translated to MIDI velocity values and are paired with the note values. Quarter-tone tunings are communicated as Pitch Bend MIDI messages. The duration of a note is not transferred in any way through MIDI, but determines the duration of a NoteOn message, before the equivalent NoteOff message is sent. The default duration of a NoteOn message is 75% of the duration of a note. It can be set manually for each instrument separately with the following command.

```
\synth \mididur 85
```

Articulation symbols are sent as Program Change values from 10 for marcato to 15 for staccato. In case Program Change messages are not an option in the receiving hardware, the staccatissimo, staccato, and tenuto articulations can get a percentage value of the total duration of the NoteOn message. For example, a staccato can be set to 50%, which means it will last half of the 75% default NoteOn duration (or another percentage that might have been set manually with the \mididur command). Arbitrary text cannot be communicated with the MIDI protocol, so it is omitted. Crescendi, diminuendi, and glissandi durations are sent as Control Change values and it is up to the user to handle this with a slew limiter or some other way. These values are internally mapped to the MIDI range from a range from 0 to the duration of one beat.

## 4.4 The Interactive Score

The score is written with a combination of primitive shapes like lines, vertices, and curves, with strings in the *Sonata* font. The latter provides shapes for the clefs, note heads, single beams, time signatures, dynamics, and articulation symbols. When the score is in animation mode, it is clocked by the sequencer and colour highlights the current note(s) of every staff. If the \showbeat command has been invoked, a semi-transparent rectangle pulsates on every beat. Figure 1 illustrates a still from a LiveLily session where the pulsating rectangle and the highlighted notes are visible.

An accompanying OF program that includes only one score part can be used when LiveLily is used in a live scoring session, so acoustic instrumentalists can sight-read the score. The sequencer runs only in the main LiveLily program so that all score-part programs are synchronised.

## 5. LIVELILY ENDEAVOURS

The development of LiveLily started as a personal project, leveraged by a personal approach to live coding [9, 11]. Up to now, this system has been used in live performances in small venues in Athens, Greece. LiveLily has been well received by the audience, with positive comments being received after the end of each performance. Even at times of malfunctions -due to the state of the software, currently being at a beta stage- the audience members enjoy the performance with this system.

The performances realised with LiveLily either follow a more traditional approach, where live coding is done through the computer hardware, or more unconventionally, where a

guitar is used as a computer keyboard to write LiveLily commands. In the latter case, the melodies that occur while typing commands are introduced to the melodic patterns written in LiveLily. Writing notes in the Western-notation and displaying the score serves this approach well, as it is centered around an instrument that often depends on reading music scores in this notation format. Future plans include character-based text generators based on Recurrent Neural Networks (RNN) with Long Short-Term Memory (LSTM), to provide melodic and rhythmic pattern suggestions in the LiveLily language, when live coding for an acoustic ensemble.

## 6. FUTURE WORK

One feature that is worth adding in the future is support for some DSP languages. Python with the Pyo module [5] is one of the first languages that will probably be added to the parser, since it is a language and module frequently used by the author of this paper and developer of LiveLily. SuperCollider is another option, but for that, contribution by other developers would be at least welcome, if not necessary. The idea behind this is to not have to launch any other software and run both the score and the “orchestra” in LiveLily.

Colour highlighting depending on the syntax is something that is common among programming languages. This can be found in Lilypond too, when written in the Frescobaldi software. This feature has not yet been implemented, but it is included in the near future plans, especially if support for other programming languages is to be added.

Another possible feature is to add the capability of creating more than one bar with the \bar command, by using the vertical bar character, similar to how bar endings are written in Lilypond, but slightly different, to enable fast typing. The way the system is currently designed, the distinction between the \bar and \loop commands is clear and logical. Being able though to define single bars only, poses a limitation to rhythm and dynamics, as currently, it is not possible to slur notes between different bars or create a crescendo or diminuendo that extends to more than one bar. If this feature is added, the \bar and \loop commands might need to change. Otherwise, LiveLily’s syntax should be extended to allow open-ended slurs and crescendi and diminuendi, so they can start in one bar and end in another, where these bars will be defined separately.

Finally, conducting short surveys either with audience members of LiveLily performances or with LiveLily workshop participants, is part of the future plans. Such workshops have already started being booked, so conducting a survey is feasible. These surveys will help in further developing and refining this system. The workshops aim at making other live coders aware of this system, so more people start to use it.

## 7. CONCLUSIONS

LiveLily is a flexible and intuitive live sequencing and live scoring system that provides a wide range of capabilities. The possible combinations of software, hardware, and acoustic instruments are many. Among others, it is possible to combine acoustic instruments with analog modular synthesizers, sync software audio with hardware electronic instruments, or connect computers in a local network and control them through this interface.

From the sequencer perspective, LiveLily provides many expressive musical gestures, like dynamics, glissandi, crescendi, articulations, and arbitrary text that can be interpreted at

will. From the live coding perspective, it uses a simple language that makes use of Western-music nomenclature, providing a coding paradigm that can be approached by non-coders too, given that they possess some Western-music knowledge. Its versatile bar and loop definition structure provides an easy way to define music segments fast. Its syntactic sugar adds on top of that, offering a shortcut to defining melodic or rhythmic patterns.

## 8. ETHICAL STANDARDS

This research has not received any funding. It also has no potential conflicts of interest. There were no participants included or animals used throughout this research.

## 9. REFERENCES

- [1] S. Aaron and A. F. Blackwell. From sonic pi to overtone: Creative musical experiences with domain-specific and functional languages. FARM '13, page 35–46, New York, NY, USA, 2013. Association for Computing Machinery.
- [2] A. Agostini and D. Ghisi. Real-Time Computer-Aided Composition with bach. *Contemporary Music Review*, 32, 02 2013.
- [3] D. G. Arellano and A. McPherson. Radear: A tangible spinning music sequencer. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 84–85, London, United Kingdom, jun 2014. Goldsmiths, University of London.
- [4] M. A. Baytas, T. Goksun, and O. Ozcan. The perception of live-sequenced electronic music via hearing and sight. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 194–199, Brisbane, Australia, 2016. Queensland Conservatorium Griffith University.
- [5] O. Bélanger. Pyo, the Python DSP Toolbox. In *Proceedings of the 24th ACM International Conference on Multimedia*, MM '16, page 1214–1217, New York, NY, USA, 2016. Association for Computing Machinery.
- [6] J. Bresson. Reactive visual programs for computer-aided music composition. pages 141–144, 07 2014.
- [7] P. Brinkmann. Mondrian music description language and sequencer. In *International Conference on Mathematics and Computing*, 2006.
- [8] V. Chandra. *Geek Sublime: The Beauty of Code, the Code of Beauty*. Graywolf Press, Minneapolis, Minnesota, USA, 2014.
- [9] A. Drymonitis. Live coding on a modular synthesizer. In *International Conference on Live Coding*, Valdivia, Chile, 2022.
- [10] A. Drymonitis and N. Chatzopoulou. Data Mining / Live Scoring – A Live Performance of a Computer-Aided Composition Based on Twitter. In *Proceedings of the 2nd Joint Conference on AI Music Creativity*, page 10, Online, July 2021. AIMC.
- [11] A. Drymonitis and M. Manousakis. Echo and narcissus: Live coding and code poetry in the opera. In *Proceedings of the International Computer Music Conference, ICMC*, pages 16–21, Limerick, Ireland, 2022.
- [12] D. Fober, Y. Orlarey, and S. Letz. Inscore an environment for the design of live music scores. In *Proceedings of the 2021 Linux Audio Conference, LAC*, California, USA, 05 2012.
- [13] G. Hajdu and N. Didkovsky. Maxscore - current state of the art. In *Proceedings of the International Computer Music Conference, ICMC*, pages 156–162, Ljubljana, Slovenia, 2012.
- [14] T. Hayes. Neurohedron : A nonlinear sequencer interface. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 23–25, Sydney, Australia, 2010.
- [15] C. Hinojosa and L. Patricia. Kanchay\_yupana//: Tangible rhythm sequencer inspired by ancestral andean technologies. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, The University of Auckland, New Zealand, jun 2022.
- [16] S. Holland and R. Fiebrink. *Machine Learning, Music and Creativity: An Interview with Rebecca Fiebrink*, pages 259–267. Springer International Publishing, Cham, 2019.
- [17] V. Lazzarini, S. Yi, J. ffitich, J. Heintz, Ø. Brandtsegg, and I. McCurdy. *Csound*. 01 2016.
- [18] T. Magnusson. The IXI Lang: A Supercollider Parasite for Live Coding. In *Proceedings of the International Computer Music Conference, ICMC*, pages 503–506, Huddersfield, UK, 2011.
- [19] A. McLean. Making programming languages to dance to: Live coding with tidal. FARM '14, New York, NY, USA, 2014. Association for Computing Machinery.
- [20] H.-W. Nienhuys and J. Nieuwenhuizen. Lilypond, a System for Automated Music Engraving. In *In Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, Firenze, Italy, 05 2003.
- [21] A. Prechtl, A. Milne, S. Holland, R. Laney, and D. Sharp. A midi sequencer that widens access to the compositional possibilities of novel tunings. *Computer Music Journal*, 36:42–54, 03 2012.
- [22] B. W. Vines, C. L. Krumhansl, M. M. Wanderley, I. M. Dalca, and D. J. Levitin. Music to my eyes: Cross-modal interactions in the perception of emotions in musical performance. *Cognition*, 118(2):157–170, 2011.