LIVELILY: THE EVOLUTION OF A LIVE SCORING / LIVE SEQUENCING ENVIRONMENT

Alexandros Drymonitis
Independent Scholar
alexdrymonitis@gmail.com

ABSTRACT

In this paper I focus on the development of the *LiveLily* Live Scoring and Live Coding system, since its initial release on 2023. This system has been developed for writing Western-music scores live. Its further development includes user defined functions, enhanced OSC communication, syntax highlighting and other features. Inspired by the *Lilypond* language, this system offers an intuitive textual language, simpler than *Lilypond*, for writing and playing music scores live. It provides the ability to use Python and the *Pyo* module to write DSP code to play scores or to control other software or hardware via OSC or MIDI communication, and communicating the score with human performers through an accompanying software that can display part of the full score. Being focused on Live Scoring, it is very responsive and takes minimal time to render a score, while it can play rendered scores in real-time.

1. INTRODUCTION

Live Scoring is an artistic practice of creating music scores in real-time. It is a field that has seen development through research and artistic practice since more than ten years, at the time of writing. Projects like *INScore* [1], *Maxscore* [2], the *bach* objects for *Max* [3] and *OpenMusic* [4] date back to 2012, 2013 and 2014, but most of them are still actively maintained. Live Coding too, is a field with active research. In this practice, an artist uses computer code to create audio or visuals and writes or edits this code live during a performance. Live Coding is characterized by its community of scholars and artists and an abundance in literature —which is demonstrated by its own conference [1] —, but also by a multitude of approaches, including combinations of live coders with other instruments, choreographers, performance artists and others [5].

LiveLily is a system for Live Scoring and Live Sequencing through Live Coding, based on a bespoke Domain Specific Language (DSL) [6]. Even though Live Coding can be characterized by curiosity, where live coders experiment with various inter-disciplinary projects, there is little to no research on the combination of Live Coding with Live Scoring for creating traditional Western-music scores. In this context, we should mention that live coders tend to see the code as a music score [7], leaving a gap between Live Coding and the traditional Western-notation music score. *Maxscore*, the *bach* objects for *Max* and *OpenMusic* can be considered as Live Scoring systems that can be used in a Live Coding context, but these projects follow the visual programming paradigm, as *Maxscore* and the *bach* objects are developed for *Max*, while *OpenMusic* is a visual programming system of its own.

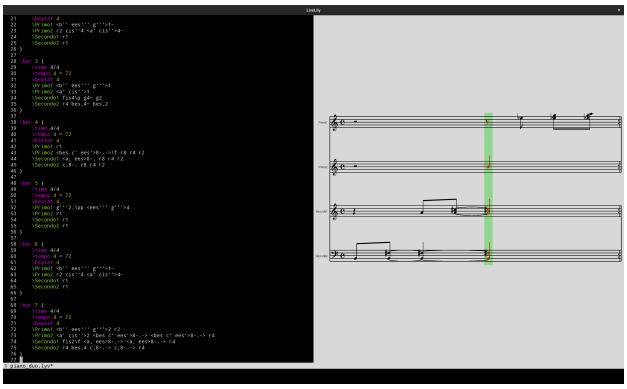

Figure 1. A *LiveLily* session with a composition by Yiannis Sfyris.

The *LiveLily* system utilizes textual code, inspired by the *Lilypond* language [8], therefore it fills the gap between textual Live Coding and Live Scoring in a traditional Western-music context. Software like *Frescobaldi* [2] might seem similar to *LiveLily*, in the context of writing code in the *Lilypond* language and rendering the score, but rendering takes a considerable amount of time in a Live Scoring context. *OpusModus* [3] utilizes textual code to render traditional Western-music scores too. Like *LiveLily*, this software features Live Coding too, but this is mainly used as a mechanism to provide feedback to composers on what they write while writing a score and it does not aim at Live Scoring.

LiveLily is an open-source [4] true Live Scoring system, as it renders the scores in real-time. It also includes a sequencer that sends either OSC or MIDI messages as a score plays. These messages can be received in other software or

[1] International Conference on Live Coding (ICLC) https://iclc.toplap.org/

[2] https://frescobaldi.org/

[3] https://www.opusmodus.com/

[4] https://github.com/alexdrymonitis/LiveLily

Figure 2. Live Coding an acoustic ensemble.

hardware, so a score can be played with any desired sound. Figure 1 illustrates a *LiveLily* session with part of a composition by Yiannis Sfyris, who collaborated with me during my Postdoc research [9].

In addition to the main system, an accompanying program that displays part of the score can be used with performers to either combine electronics with acoustic instruments, or to live code acoustic ensembles. Figure 2 illustrates one such Live Coding session, where the author uses *LiveLily* to live code a score which an acoustic ensemble sight-reads and follows to play techno. Prior research I have conducted has combined performers with hardware [9], with a Yamaha Disklavier robotic piano being controlled by *LiveLily* and at the same time a performer playing on the same instrument.

This paper is structured as follows: The next section analyzes the architecture of the *LiveLily* system. The section after that describes the basic usage of the *LiveLily* language. Then a section on the new features that are the main focus of this paper is laid out. A section on future work follows, with the conclusions closing the paper.

2. SYSTEM ARCHITECTURE

The *LiveLily* system is written in C++ with the openFrameworks toolkit [5] (OF). This toolkit was chosen for its flexibility, its support from its user and developer communities, and its simplicity concerning commands for drawing visuals. The entire system consists of four distinct parts. The first part is the editor where the user types code in the *LiveLily* language as shown in the left side of the screen capture in Figure 1. Even though the *LiveLily* system is not solely a code editor, and only part of the focus of this system's development has been dedicated to developing the editor, it includes all the necessary functionalities a user would expect from such an editor, including line numbering, syntax highlighting, auto-insertion of closing brackets, copying and pasting to/from the clipboard, and others. The second part of the system is the interpreter that translates the code to executable commands. This is invoked when the user asks the system to execute a line, or a larger chunk of code. Executable code being called explicitly by the user gives the flexibility of being able to type anything into the editor but execute only part of it. This means that the users are able to code their scores at their desired pace and execute their code bit by bit, leading to a fine control of how the score is played. The third part is the score that displays what is written in the *LiveLily* language in traditional Western-music notation, as shown on the right side of Figure 1. The last part is the sequencer that plays the score and communicates it with OSC or MIDI.

[5] `https://openframeworks.cc/`

2.1 The Score

The score of the *LiveLily* system allows the visual representation of the coded musical indications. It is written using the Sonata font and primitive shapes provided by OF. The Sonata font is used for symbols like note heads, rests, clefs, meters, articulation symbols, accidentals, dynamics, and octave symbols. Shapes like stems, beams, glissandi, slurs, and the actual staves are drawn using primitive shapes like lines and curves.

A previous Live Scoring approach with the *Data Mining / Live Scoring* project [10] included the *Lilypond* engraver and OF, where the former rendered scores in the PNG format, and the latter displayed them and highlighted the currently playing bar with a blinking cursor at the beginning of it. This approach though was not responsive enough in a Live Scoring context. This is because *Lilypond* takes a few seconds to render a single bar of music. Additionally, extra work was needed to spot the beginning of each bar, as the score was rendered as a PNG file and not displayed with OF's drawing classes. This required computer vision techniques provided by the ofxOpenCV addon for OF, which are time consuming in a Live Scoring context.

The *LiveLily* approach is much more responsive as it takes very little time to render a bar. While evaluating the system's responsiveness, rendering bars with four staves took less than one millisecond. The bar in Figure 3 —composed by Yiannis Sfyris— was rendered twenty-seven times to compare the various rendered durations between each iteration, with the shortest duration being 377 microseconds and the longest 799 microseconds. Bars that have already been rendered are stored in *LiveLily*'s memory and can be called anytime during a session without any delay at all, since no additional rendering takes place. Also, spotting the position of any element of the score does not require any perceived computational time, as all elements are OF objects. This means that the program is instantly aware of where each item is drawn and no computer vision techniques are required.

This approach provides much more flexibility as to which item of the score can be highlighted and in what way. In Figure 1 we can see a green cursor at the third beat of the bar displayed over the score. This is a blinking cursor that appears at the currently playing beat, to display the sequencer. This is useful for the audience to follow the score, but more importantly for a performer that plays along to be synchronized. In this Figure, we can also see the currently active notes and rests displayed in red color. All this highlighting is possible because all the score elements are drawn as OF objects, which gives full control over their appearance.

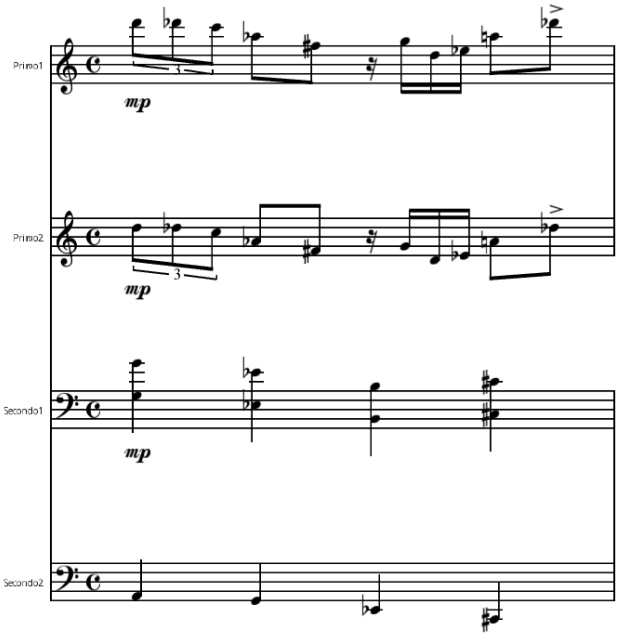

Figure 3. A four staff bar composed by Yiannis Sfyris, rendered to time the rendering duration.

2.2 The Sequencer

Three out of four parts of this system run in the same thread, with the sequencer having its own thread. This due to the fact that OF, being primarily a graphics-oriented toolkit, operates at a fixed framerate that works well for visuals, but not necessarily for processes that need a faster rate. Even at high framerates like 60, or even 120 frames per second, running a musical sequencer at high tempi would cause jitter, due to the time grains necessary to synchronize fast notes (e.g. sixteen or higher) being shorter than what a high framerate provides.

OF provides threading that runs at different clock speeds than the framerate of a program written in it. Utilizing this feature, *LiveLily*'s sequencer runs at a clock speed of 100 microseconds. This clock speed provides the necessary resolution to play scores at high tempi with fast notes without jitter.

3. BASIC USAGE

In this section we include some examples of the *LiveLily* language, to give an impression to the reader of how the *LiveLily* system works. The main commands to create a simple score are *insts* and *bar*. The first one initializes the instruments of the score. Once the instruments are created, they cannot be changed, so the user must created all instruments at the beginning of each session. This command takes the names of the instruments as arguments. The line below will create a string quartet with the names *violin1*, *violin2*, *viola*, and *cello*.

```
\insts.init violin1 violin2 viola cello
```

Once created, if the score is already visible, four staves will be created, one for each instrument. The score can become visible with the *score.show* command.

Once the instruments have been created, the user can start creating bars of music. The command to create a single bar is *bar*. This command takes a name for the bar to be created, and the contents of the bar inside curly brackets. The lines for each instrument inside a bar definition must be separated, as shown in the code example below.

```
\bar 1 {
  \violin1 c'''2 b''4( c''')
  \violin2 g''4 fis'' f'' e''
  \viola e'2 d'8( des') c'4
  \cello \clef bass c2 g2
}
```

The name of the bar in this example is *1*, but it could be anything, be it a number or a string. Once a bar has been created, it can be called by its name with a backslash prepended to it. In the example above, if the user wants to display this bar, they should type *1*. Note though that when a bar is created, if the sequencer is not running, the bar will immediately be displayed. Calling a bar by its name to display it is necessary if the sequencer is running and another bar is being currently displayed.

If the user creates more than one bars, these can be combined in a loop with the *loop* command. This command is very simple. The line of code below demonstrates its use.

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

The example above supposes that the user has defined at least two bars whose names are *1* and *2*. This line will create a loop with these two bars which is named *1-2*. As with bars, loop names take a backslash at their beginning, once created. An addition to the *loop* command is the asterisk which is used to repeat a bar in the loop defined for a certain number of times. The line of code above can be changed to the line below, in which case, the loop will repeat bar *1* twice before playing bar *2*.

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

Being inspired by the *Lilypond* language, the *LiveLily* language supports elements such as articulations, slurs, ties, glissandi, and dynamics. All these are handled by the sequencer of the system, without the user needing to handle them in the DSP environment they work in. Glissandi and dynamics are communicated via OSC or MIDI. When used with OSC, the frequency and amplitude changes are sent to their respective address, */note* and */dynamics*, respectively. In the case of MIDI, glissandi are handled with Pitch Bend messages, while dynamics are handled with the overall volume control. Slurs and ties are handled either by not sending a zero amplitude when used with OSC, or by sending the Note Off message after the next Note On, when used with MIDI.

Articulations include staccato, staccatissimo, accent, marcato, trill, tenuto and portando. The staccato, staccatissimo, and tenuto articulations are handled by the sequencer, by shrinking or stretching the Note On durations. The percentage of these shrinks and stretches to the overall Note On duration can be set by the user. Their default values are 50%, 25%, and 90% of the Note On duration for staccato, staccatissimo, and tenuto, respectively. The rest of

the articulations are aimed at instrument performers that play following the accompanying software mentioned in the *Introduction* of this paper, which displays part of the full score.

4. NEW FEATURES

Since its initial announcement in 2023 [6], the *LiveLily* system has seen a lot of development. New features have been added to the editor, the interpreter, the handling of errors, the language design of the system, the capability to read MusicXML files, the capability to connect to external serial devices, its OSC communication features, and the integration of the Python programming language. In this section we will analyze these features in detail.

4.1 Additions to the Editor

The additions in the editor of the *LiveLily* system include syntax highlighting, editor modes, and the ability to split the editor window in panes. In this section we will analyze these separately.

4.1.1 Syntax Highlighting

One of the feature that were included in the *Future Work* of the initial paper for this system was syntax highlighting [6]. As seen in Figure 1, commands like *\bar*, *\time*, and *\tempo* are displayed with a different color than instruments like *\Primo1*, *\Primo2*, etc. The colors are used to separate the following features of the *LiveLily* language: first level commands, second level commands, instrument names, bar names, loop names, functions, comments and digits.

4.1.2 Editor Modes

Another addition to the editor is four different modes, following the paradigm of the Vim editor. Vim has four modes of operation, Normal, Insert, Visual, and Command mode. *LiveLily* imitates parts of each of these modes. In Insert mode, like in Vim, the user can type text into the editor as they would do with any simple text editor. Contrary to Vim, whose default mode is the Normal mode, in *LiveLily* the Insert mode is the default one. In Normal mode, the user cannot type text, but can navigate through it with the H, J, K, and L keys, like in Vim. This feature is useful when touch typing, so the user does not need to change the position of their hands, thus can maintain eye contact with the screen, uninterrupted.

In Visual mode, the user can copy text to the program's memory —not the clipboard—, and can paste that text in the editor. These text copies are called *yanks* in Vim's terminology. Copying to and pasting from the clipboard is possible in Insert mode with the combination of CTRL+C for copy and CTRL+V for paste. In Command mode, the user can type certain commands, either to load or save a text file, navigate through the computer system —aiming at locating files to load—, and quit the program. Inspired by NeoVim, when in Insert mode, the cursor is shown as a line. When in Normal or Visual mode, the cursor is shown as a rectangle. When in Command mode, the cursor moves

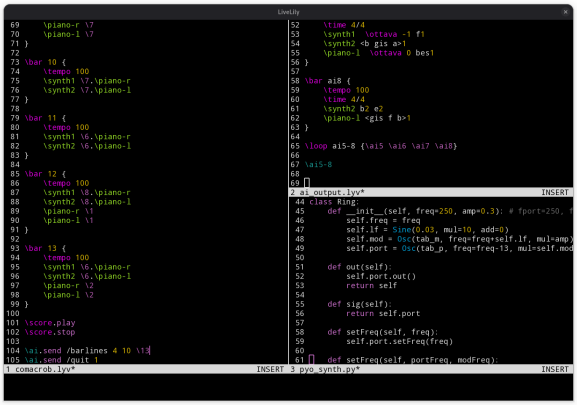

Figure 4. Splitting the editor into panes.

to the message area of the *LiveLily* window (see *4.2.1 Error, Warning, and Note Messages* subsection below).

4.1.3 Panes

The last addition to the editor is splitting its window into panes. This can be done either horizontally, or vertically, or in a combination of the two. Figure 4 illustrates the editor split into three different panes, with two columns, one row for the first column and two rows for the second. In this Figure we can see that the panes are being numbered from left to right and top to bottom. The numbering is shown in the thick bar below the pane and its order is regardless of the order in which the panes are created. The active pane can be spotted by the cursor shown as a line —because it is in Insert mode—, where the inactive panes have their cursor as an outlined rectangle, regardless of the mode these panes are in. Switching between panes is done with the Alt key and the number key of the pane the user wants to activate. It should be noted that the commands executed in any pane are directed to the same interpreter. Regardless of which pane is active, any command executed will affect the same score, unless a pane is set to execute Python code, in which case the Python interpreter is invoked. All Python panes share the same interpreter too, like all *LiveLily* panes do.

4.2 Additions to the Interpreter

Additions to the interpreter include new commands that extend the *LiveLily* language's command set, improvements to error, warning, and informational messages, as well as changes to the way the interpreter executes commands passed as arguments to other commands. For the sake of clarity in writing this paper, the new commands that have been added to the language are analyzed in their own subsection and not in this subsection.

4.2.1 Error, Warning, and Note Messages

The first version of the *LiveLily* system included error messages that were triggered when a command did not exist, or where there was a syntactical error in it. The current

Figure 5. An error message and its highlighted line.

version of the system includes warning and note messages, like C or C++ compilers.

When an error occurs, the command is not executed, and the error message is printed in the message area of the *LiveLily*. This is the bottom area shown in Figures 1, 4 and 5. The error message is displayed in red, containing the line number in which it occurred, and the actual line in the editor turns red too, as shown in Figure 5.

Warning and note message let the command be executed, but are also printed in the message area. The warning messages are displayed in orange color, while the note messages are displayed in white. The warning messages highlight the respective line, like the error messages do, but in orange color.

4.2.2 Nested Commands

In the current version of the *LiveLily* system, the interpreter can parse commands that are passed as arguments to other commands. The syntax for this feature is the following: if a command takes one argument, this is written immediately after the name of the command, while if the command takes more arguments, these are placed within curly brackets. To maintain consistency with *Lilypond*, which has inspired the development of *LiveLily*, when a command takes multiple arguments, the first one can be written immediately after the command name, not placed within curly brackets, and the rest of the arguments can be placed inside curly brackets right after. This has been enabled due to the way tuplets are written in *Lilypond*, which take the following form:

```
\tuplet 3/2 {c'8 cis' d'}
```

4.3 New Commands

The following new commands have been added to the *LiveLily* language: *bars*, *function*, *list* and the wildcard used to create loops with multiple bars.

4.3.1 The \bars command

In the *Future Work* of the initial paper on this system, the *bars* was mentioned [6]. This feature has now been implemented and enables users to create multiple bars with a single command execution. The example below demonstrates how to create two bars with the *bar* command, followed by a loop that iterates over these two bars.

```
\bar 1 {
   \inst1 e''4 ees''~ 2
   \inst2 c'2 r
}

\bar 2 {
  \inst1 ees''4 g''2 gis''4
  \inst2 g'8 fis' e' ees' c'2
}

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

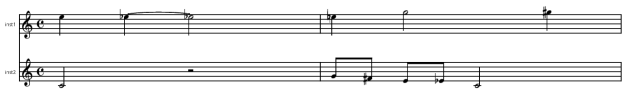

Figure 6. A two-bar loop.

This example will create the two-bar loop shown in Figure 6. With the addition of the *bars* command, the code above can now be written as follows:

```
\bars 1-2 {
  \inst1 e''4 ees''~ 2 | ees''4 g''2 gis''4
  \inst2 c'2 r | g'8 fis' e' ees' c'2
}
```

The code above will create both bars and the loop of the first code example. The different bars are separated by the vertical bar character (|). Also, when creating multiple bars this way, the user still has access to the each of these bars separately by appending a hyphen followed by the bar index to the name of the bars. In the example above, the first bar can be accessed with *1-2-1*, and the second bar can be accessed with *1-2-2*.

4.3.2 Using the wildcard

The wildcard character can now be used in a *loop* definition to include either all rendered bars or all bars whose names share the same character(s). It works similar to the way this character is used in the terminal of a Unix system. If, for example, the user creates some bars whose names start with the word "intro", while other bars do not contain this word in their name, then they can create a loop with all the bars that do contain this word in their name with *loop introloop* {*intro**}, —here "introloop" is used as an example name for the loop of all bars whose names start with "intro". To create a loop with all the defined bars, regardless of their names, the user can type *loop loopname* {*}.

4.3.3 The \function command

The next addition to the command-set of the *LiveLily* language is the *function*. This command enables the user to create a function with or without arguments that they can call at a later stage. The code below is an example of the use of the *function* command.

```
\function solo1-3 {
  \solo \inst1
  \solo \inst3
}
```

The function defined above will mute all instruments but *inst1* and *inst3* with a single call. Once defined, the user can call the function with the name they defined it with. In the example above, the function can be called with *solo1-3*.

Functions can also be bound to a certain event, like the beginning of a loop, the beginning of a bar, the beat, the frame rate, or a specific note. To bind the function of the example above to the beginning of a loop, the user must type the following:

```
\solo1-3.bind loopstart
```

Function binding to events can happen for a specified number of times and certain number of binding events can be skipped. If the user wants to bind the function of this example at the beginning of every other loop for four times, they should type the following:

```
\solo1-3.bind loopstart every 2 times 4
```

4.3.4 The \list command

The last addition to the command-set of the *LiveLily* language is *\list*. This command enables the user to create lists with data that can then be traversed to enable less typing. The code example below creates four staves for percussive instruments and uses a list to turn them all to one-line staves that are used for rhythm. The dollar symbol in the example below is replaced by the items of the *\perc* list, one at each iteration.

```
\insts perc1 perc2 perc3 perc4
\list perc {\perc1 \perc2 \perc3 \perc4}
\perc.traverse {$ rhythm}
```

4.4 Language Design

The language design of the *LiveLily* system has changed to allow for a more coherent structure. The commands of the language are now separated to first and second level. An example of a first level command is *\score*, and an example of a second level command is *show*. These two are concatenated with a dot, *\score.show*. Some commands take arguments, which follow the command with a white space, or as already mentioned above, in case these are more than one, inside curly brackets. Such an example is the *\score.show barcount*, where *barcount* is an argument to the second level *show* command.

Apart from the predefined commands, instrument names, bars, loops, functions and lists that are defined by the user behave like commands. To easily distinguish one from another, each of these types take a different color in the syntax highlighting, but they behave like commands in the context of taking either arguments or second level commands. For example, a bar name followed by an instrument name as a second level command, will return the line of that instrument of this bar. The example below uses this feature to copy the melodic line of the saxophone from bar *\1* to the clarinet in bar *\2*.

```
\bar 2 {
  \clarinet \1.\saxophone
}
```

4.5 Python Integration

In the *Future Work* of the initial paper on *LiveLily*, integrating Python was mentioned, among other features [6].

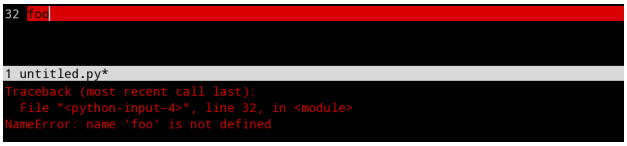

Figure 7. A Python error message integrated into *LiveLily*.

Python has now been integrated into the system through the *Pyo* integration in OF. *Pyo* is a DSP module for Python written in C [11]. The code that integrates *Pyo* in OF has been modified to route Python's both STDOUT and STDERR into the C++ code of *LiveLily*, so that can be fully integrated into the error, warning, and note message handling of *LiveLily*, as show in Figure 7.

Since Python has been integrated through *Pyo*'s OF embedding, the *Pyo* module is also included. This enables users to write their DSP code in Python with *Pyo* in the *LiveLily* editor and control it with the scores they write in the same system. In this case, no OSC or MIDI is needed to control the various *Pyo* synthesizers. All the user needs to do is define an instrument to send its notes to Python with *\instname.sendto python*. This command directs the sequencer to route the Note On, Note Off, and dynamics messages straight as Python messages. For this to work, the user must create a Python class with the name of the instrument (without the backslash), and in this class, to define a *setFreq()* and a *setAmp()* method, to control the frequency and amplitude of the class. For an instrument defined in *LiveLily* with the name *synth*, the Python code example below would be valid. Then, in the pane with the *LiveLily* code, the user should write *\synth.sendto python*.

```
class Synth():
  def __init__(self):
    self.adsr = Adsr()
    self.supersaw = SuperSaw(mul=self.adsr).out()
  def setFreq(self, freq):
    self.supersaw.setFreq(freq)
  def setAmp(self, amp):
    self.adsr.mul = amp
    self.adsr.play()

synth = Synth()
```

4.6 Reading MusicXML Files

MusicXML is a format for music scores that enables the exchange of scores between programs [12]. This feature was added during the postdoc research of the author, where a collaboration with composers that provided original music made this feature necessary, as these composers used various score engraving software. The MusicXML format provides an extensive flexibility in communicating score elements. *LiveLily* being rather limited as to what it can display —see the *Basic Usage* section—, ignores elements that are not used within it.

When a MusicXML file is loaded in *LiveLily*, the score is written bar by bar in the native language of the system. Then, it is left to the discretion of the user whether they will create a loop with all the bars and play the entire piece, or

whether they will treat the score in some other way. Reading MusicXML files provides flexibility in the context of composers sharing their pieces with *LiveLily* users, or even composers loading their own compositions to *LiveLily* so they can experiment with their structure.

4.7 Connecting to Serial Devices

The *LiveLily* system can now connect to external serial devices like the Arduino. It can read and write bytes to/from such a device. This can be helpful if the user wants to control external devices through the score. By utilizing functions that can be bound to certain events, it is easy to bind physical output through a serial device to a musical event such as a specific note or the beginning of a bar. The code below is an example of how one can connect to a serial device and bind the sending of a byte with a random value to the beginning of a loop.

```
\listserialports
\openserialport 1
\function sendbyte {
  \serialwrite \random 255
}
\sendbyte.bind loopstart
```

4.8 Enhanced OSC features

In its initial release, *LiveLily* used OSC only to communicate the score to other software. In its current version, the user can define their own OSC clients to send arbitrary information to other software. An example of how this can be useful is the use of a clicktrack. Even though *LiveLily* includes a blinking cursor so that performers can visualize the tempo and synchronize themselves to other performers or electronics, having used this software with performers has highlighted the need for adding a clicktrack on top of the blinking cursor. Since *LiveLily* does not produce sound, a clicktrack can be created in another software and it can be controlled by *LiveLily*. The code below creates an OSC client sending messages to the home IP and port 9010, named *\click*. It then creates a function that sends the beat count —using the *\beatcount* command— to the OSC address */beat*, and binds this function to the beat of the sequencer.

```
\osc click
\click setup 9010
\function sendbeat {
  \click.send /beat \beatcount
}
\sendbeat.bind beat
```

The code chuck above results in *LiveLily* sending the beat count —with 1-based counting— via OSC. This can be used in another software to control a clicktrack. For example, a clicktrack can be created with a simple Pd patch with an oscillator with a short envelope, where when receiving *1* through OSC, the oscillator's frequency is set to 880, and for all other values it is set to 440. This way, the first beat of every bar will sound an octave higher than the rest of the beats, like the traditional clicktrack works.

With the integration of Python and *Pyo*, the same can be achieved inside *LiveLily*. To accomplish that though, an additional staff must be created for every bar definition, that will contain an A note above the staff with a G clef for the first beat, and an A note inside the staff for all other beats. It is therefore probably simpler to create the clicktrack in another software and control it with OSC. This way, two different audio interfaces can be used, one for the clicktrack that is played by another software and one for *LiveLily*, in case of using *Pyo* to combine instrumentalists with live electronics.

5. FUTURE WORK

Even though many new features have been added to the *LiveLily* system, there is still room for improvement. A feature that has been in the list of features to add but has not yet been implemented is to add support for the Lua scripting language. This language has been targeted mainly because of the ofxLua addon for OF which allows the user to write OF code in Lua for changing an OF program dynamically. The aim is to treat the score as a visual element and apply OF code to it, for example to blur, degrade, delay the image or apply other visual effects to it.

An addition that is currently being implemented is reading MIDI files to render them as scores. This feature is currently close to being realized, and will most likely be included in the next version of the software. A difficulty this feature presents is the interpretation of durations, as MIDI files include Note On messages only and their duration, which is not the entire duration of the note to be played. For example, for a quarter note at 60 BPM, the duration of the note will not be exactly one second but a bit less, to account for triggering the release part of the envelope of the MIDI instrument. If a note is written as a quarter note staccato, then the duration is even shorter, and it is difficult to decide how to treat it. Once these issues are dealt with, reading MIDI files will be added to *LiveLily*.

6. CONCLUSIONS

I have presented the evolution of the *LiveLily* system. This system provides a solution for Live Scoring a traditional Western-music score through Live Coding, a combination that, to the best of my knowledge, does not exist in other systems. Since its initial announcement, two years ago at the time of writing, there has been a significant number of additions to this system. These additions provide a smoother workflow when using this system for Live Scoring.

Having already been used in both professional artistic and academic contexts, it has been tested in realistic conditions where Live Scoring has been the main characteristic of these performances. These performances concerned both experimental new music and more conventional genres like techno. This demonstrates a wide range of applications that are possible with this system, concerning both the music style and the nature of such a performance, whether it is purely artistic or academic.

7. REFERENCES

[1] D. Fober, Y. Orlarey, and S. Letz, "INscore an environment for the design of live music scores," in *Proceedings of the 2012 Linux Audio Conference, LAC*, California, USA, 05 2012.

[2] G. Hajdu and N. Didkovsky, "Maxscore - current state of the art," in *Proceedings of the International Computer Music Conference, ICMC*, Ljubljana, Slovenia, 2012, pp. 156–162.

[3] A. Agostini and D. Ghisi, "Real-Time Computer-Aided Composition with bach," *Contemporary Music Review*, vol. 32, 02 2013.

[4] J. Bresson, "Reactive visual programs for computer-aided music composition," 07 2014, pp. 141–144.

[5] A. Mclean, "Reflections on Live Coding Collaboration," in *Proceedings of 3rd conference on Computation, Communication, Aesthetics and X (xCoAx)*, Glasgow, Scotland, 01 2015, pp. 213–220.

[6] A. Drymonitis, "Livelily: An expressive live sequencing and live scoring system through live coding with the lilypond language," pp. 256–261, May 2023. [Online]. Available: http://nime.org/proceedings/2023/nime2023_37.pdf

[7] T. Magnusson, "Algorithms as Scores: Coding Live Music," *Leonardo Music Journal*, vol. 21, pp. 19–23, 12 2011. [Online]. Available: https://doi.org/10.1162/LMJ_a_00056

[8] H.-W. Nienhuys and J. Nieuwenhuizen, "Lilypond, a System for Automated Music Engraving," in *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, Firenze, Italy, 05 2003.

[9] A. Drymonitis and M. Koutsomichalis, "Composing for acoustic robots - instant synthesis for computer-controlled acoustic instruments through live coding and ai: Research in progress," *Chroma: Journal of the Australasian Computer Music Association*, vol. 40, no. 1, Jul. 2025. [Online]. Available: https://journal.computermusic.org.au/chroma/article/view/21

[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*. AIMC, Jul. 2021, p. 10. [Online]. Available: https://doi.org/10.5281/zenodo.5137948

[11] O. Bélanger, "Pyo, the Python DSP Toolbox," in *Proceedings of the 24th ACM International Conference on Multimedia*, ser. MM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1214–1217. [Online]. Available: https://doi.org/10.1145/2964284.2973804

[12] M. Good and G. Actor, "Using MusicXML for file interchange," in *Proceedings Third International Conference on WEB Delivering of Music*, 2003, pp. 153–.