Glicol: A Graph-oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet

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

This paper introduces the new music live coding language Glicol (graph-oriented live coding language) and its web-based run-time environment. As the name suggests, this language is designed to represent directed acyclic graphs (DAG), using a syntax optimised for live music performances. The audio engine and the language interpreter are both developed with the Rust programming language. With the help of WebAssembly and AudioWorklet, this language can run in web browsers. It also enables co-performance with the support for collaborative editing. Taking advantages of the Rust programming language design, the run-time environment is both safe and efficient. Documentation and error handling messages can be accessed in the web browser. All in all, we see Glicol as an efficient and future-oriented language for collaborative text-based musicking.

1. INTRODUCTION

When used in music contexts, the term *live coding* refers to musical performances during which performers write computer programs in real-time to make music [6]. So far, dozens of live coding languages have been developed.[1] Among these languages, the examples of language *abstractions* are ubiquitous. The abstraction level is related to how much the lower-level implementation details are hidden.

The C and C++ programming languages are commonly viewed as low-level languages as they can access the computer memory pointer directly. These languages are also popular in low-level audio programming [4, 17]. Most existing audio programming languages, e.g. SuperCollider [14], have their parser and signal processing back-end written in C or C++. This makes it possible to focus on the sound synthesis, and not on handling memory pointers. Many live coding languages such as TidalCycles [15], ixi lang [12], and Sonic Pi [1] are again built on top of SuperCollider. This additional abstraction layer allows for focusing on creating musical ‘patterns’, while leaving the sound synthesis to SuperCollider.

The abstraction level can influence which platform the languages can be used on. Early live coding languages were mostly based on OS-specific applications. In recent years, browser-based environments have become increasingly popular, of which Gibber.js [18] may be the most well-known example. Thanks to new technologies such as WebAssembly [8] and AudioWorklet [5], browsers such as Chrome or Firefox can now support high-performance audio at a near-native speed. Several audio programming languages and libraries, e.g. Csound [20] and Faust [11], together with the recently developed Sema live coding environment [3], have all adopted these technologies.

We can also see another abstraction type in web audio. Our previous live coding environment, QuaverSeries [9], is a functional programming language that runs in browsers. Its audio engine relies on Tone.js [13], a JavaScript library built on top of the Web Audio API. In practice, on the one hand, the language abstraction provides us with the convenience to use the concise customised syntax smoothly. On the other hand, we find it less flexible for sample-level sound synthesis and hard to handle the run-time errors from the memory aspect. These issues make it hard to guarantee the run-time environment robustness in musical performances.

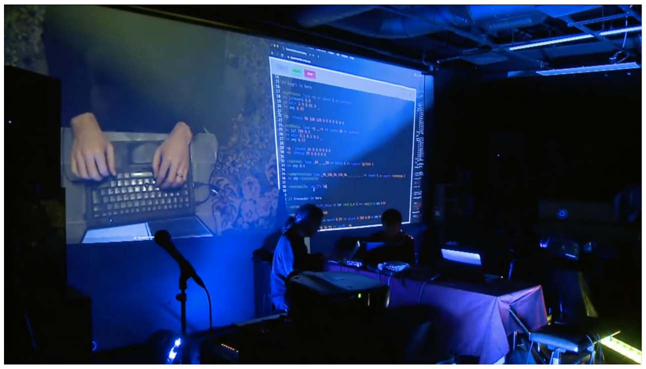


Figure 1: The authors engaged in a live coding performance during the Web Audio Conference 2019 in Trondheim. Here Glicol's precursor QuaverSeries was used. Glicol inherits the syntax style and the collaborative environment of QuaverSeries, while most of the back-end has been redeveloped.

Thus, we are motivated to rethink live coding languages

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

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

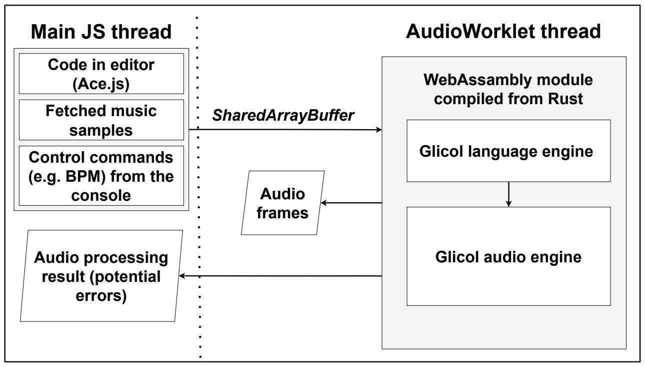


Figure 2: An overview of the Glicol architecture. Both the language and the audio engine are written in Rust and compiled to a WebAssembly module that runs in an AudioWorklet thread.

from the lower level and develop Glicol[2], the successor of QuaverSeries. In this process, the Rust programming language offers an effective alternative to C and C++. This system-level programming language is becoming increasingly popular for its robustness in memory safety and computing performance. These advantages are realised by the *ownership* mechanism in Rust, and this mechanism also further facilitates a 'zero-cost abstraction'.[3] This means that in theory, we may be able to write a parser in Rust and convert it into audio streams without losing performance redundantly.

Developing a live coding language from the low level can have various benefits: (1) better audio performance can be obtained; (2) errors can be handled in the memory level; (3) the language is easier to port to different platforms such as Bela [16]; (4) the language can serve as an intermediate structure to support higher-level customised languages. In this paper, we will focus on the design (Section 2) and implementation (Section 3, 4, 5) of the Glicol engine and its web-based IDE. In the end, we will have a general discussion (Section 6) and conclusion.

2. DESIGN CONSIDERATIONS

We begin the design considerations with the programming paradigm, which largely determines the interface we interact with in live coding. The two most popular programming paradigms in the live coding community are *functional* programming (FP) and object-oriented programming (OOP). There are benefits of both these approaches, but we have chosen a *graph-oriented* paradigm. One reason is to experiment with this new idea, as a research in sound and music computing. Another is to fully leverage some unique features in Rust, such as the ownership mechanism.

The term *graph* means the abstract collection of a series of *nodes* connected by *edges* [19]. In audio programming, the concepts of *graph* and *nodes* are ubiquitous, such as in the Web Audio API and SuperCollider. In the Rust Audio community, the concept of graph has also been widely adopted, e.g. the `FunDSP` project[4] and the `dasp_graph`[5] library. They both take advantage of Rust's *trait* feature by offering a template for implementing the `Node` *trait* for different structures.

[2]https://github.com/chaosprint/glicol
[3]https://boats.gitlab.io/blog/post/zero-cost-abstractions/
[4]https://github.com/SamiPerttu/fundsp
[5]https://docs.rs/dasp_graph

```
~lead: play >> sin_osc 440.0 >> amp ~mod

~mod: sin_lfo 1.5 0.1 0.5
```

Figure 3: QuaverSeries amplitude modulation syntax. We wrapped Tone.js *instances* in each function. There are some hacks here: all the functions before the amp are used to organise required information, and finally, in the amp function, we call the .play() method of a Tone.js object to make sound.

```
lead: sin 440 >> mul ~mod

~mod: sin 1.5 >> mul 0.2 >> add 0.3
```

Figure 4: Glicol syntax for amplitude modulation.

Though the concept of graphs and nodes is widely used in audio programming and development, few languages adopt a graph-oriented programming paradigm in the syntax design. In our experiments with QuaverSeries, we find its syntax based on the functional programming paradigm can also be used for a graph-oriented paradigm with some minor modifications (see Figures 3 and 4). We believe this paradigm can be easier for a beginner to master as its logic is fairly straightforward. It can also be used by advanced programmers as an intermediate structure for developing higher-level languages.

In Glicol, a *node* is represented by using specific keywords such as the `sin`, `mul` and `add` in the example shown in Figure 4, followed by its required parameters. A *chain* can be created by connecting nodes in series with the double greater-than sign (>>), and a *reference* can be used to denote this chain of signal flow. In the example, both `lead` and `~mod` are *references*. Using the *reference* as the node's parameter means that this parameter is controlled by another chain of signal, which is also called a *side-chain* in signal processing. Note that only the *reference* that comes without a tilde (~) will be sent to the audio interface. This is the syntax for separating control signals and audio signals, although they both run at audio rate. The reference of a signal chain can also be used as a node in another chain (see Figure 5).

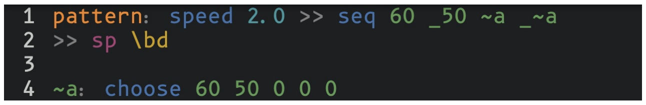
```
~fx: mul 0.3

~synth: saw 80

out: ~synth >> ~fx
```

Figure 5: References used as nodes.

Despite their similarities in appearance, there is some intrinsic discrepancy between these two kinds of syntaxes. The nodes in Glicol syntax are not functions, but the graph data structure in Rust. This data structure can be converted from the input code *String* smoothly. In contrast, QuaverSeries is based on 'impure' functions that can access global variables such as the Tone.js *Object* directly. As a result, the implementation in QuaverSeries is not as robust as the one in Glicol in terms of memory safety, as there may be potential data conflicts in these impure functions. In Rust, implementing Glicol in a functional programming manner is limited by the ownership mechanism, which is used to guarantee memory safety. For example, it is complicated for a global variable, such as the clock, to be 'moved' inside and outside the *scope* of different functions. Therefore, we believe that the graph-oriented paradigm fits the design of Rust better when it comes to implementing an audio programming language.



```
pattern: speed 2.0 >> seq 60 _50 ~a _~a
>> sp \bd

~a: choose 60 50 0 0 0
```

Figure 6: Sample playback in Glicol. The notes in the *seq* node can be controlled by a *choose* node, whose parameter number determines the probability of random selection.

3. IMPLEMENTATION IN RUST

To implement the syntax in Rust, we need to build different node structures and a language engine that can convert the code text to an audio graph. Also, we need to consider how we can dynamically manage these nodes in a graph during a live coding session.

3.1 Implementing the node trait

In the audio engine implementation, we have used a customised version of `dasp_graph` library.[6] The default buffer size in the library is hard-coded to a constant value 64, while in our customised version, the new Rust feature *min_const_generics* is used so that we can set the buffer size to any valid number, e.g. 128 in our Web Audio application. The library provides a template for us to implement a *trait* called `Node` for all these node structures such as `SinOsc`, etc. With this *trait*, these node structures are all embedded with a method called `process`. This method takes an input *buffer* array and output a *buffer* array. Within its definition, we write the DSP code to determine how the output *buffers* should be calculated from the input *buffers*. Thus, the update rate is based on the buffer size, while the control of some nodes can be in sample level such as in the `delayn` node.

The nodes that support sample playback require some special consideration. In Glicol, we define the behaviour of the `sp` (sample playback) node like this: once it receives a non-zero value from its input—which should be placed at the first position of the incoming block signal array—it will schedule a sample playback inside the node. The playback rate is determined by the trigger's value, which will consequently alter the playing pitch of the audio sample. For instance, the value 1.0 triggers the default playback rate of the audio sample. A trigger value of 2.0 will play the sample one octave higher (see Figure 6).

Thus, many nodes can be used to trigger audio sample playback. For instance, the `imp` node, i.e. 'impulsive node', and can send out an impulse signal that triggers a sample playback periodically. The node `seq` takes a sequence of MIDI note values or underscores as parameters. Notes are represented by integers while underscores denote rests (silence). The parser will divide one bar into equal length based on the spaces. The default bar duration is 2 seconds, equivalent to 120 beats per minute with a time signature of 4/4. Then, each segment can be further divided into smaller, equidistant sub-segments based on the number of MIDI notes and rests.

3.2 Converting code text to an audio graph

To convert a code string to an audio graph, we build a parser to process the code. In Glicol, we choose Pest.rs as the parsing tool[7]. It allows us to define the language rules in PEGs (parsing grammar expressions) paradigm [10]. Next, we can call its API to parse the code to node information such as the *reference* of a *chain* of nodes, the name of a single node, and its parameters.

To maintain the lazy evaluation manner, that is, writing first and defining later, we parse the code first, and the node information is kept in a *Vector* structure (re-sizable arrays in Rust) chain by chain. Then, these *Vectors* are saved in a *HashMap* structure (a dictionary-style data structure in Rust). When parsing each node, the side-chain information (the node index and *reference* name tuple) is stored in another vector called `sidechain_list`. Finally, the edges are handled only after all the nodes are parsed and the relevant information is stored.

As for the clocking, a `clock` node is connected to all the user-created nodes to ensure synchronisation. The `clock` node is invisible to the users but plays a vital role in avoiding over-processing for some nodes used as *references* in more than one places. When the `process` method is called within each node, the internal clock of that node will be compared with the input buffer of the clock node that contains the current clocking information. If it is already processed once, the node should yield a stored output buffer rather than calculate a new one.

3.3 Dynamic node management

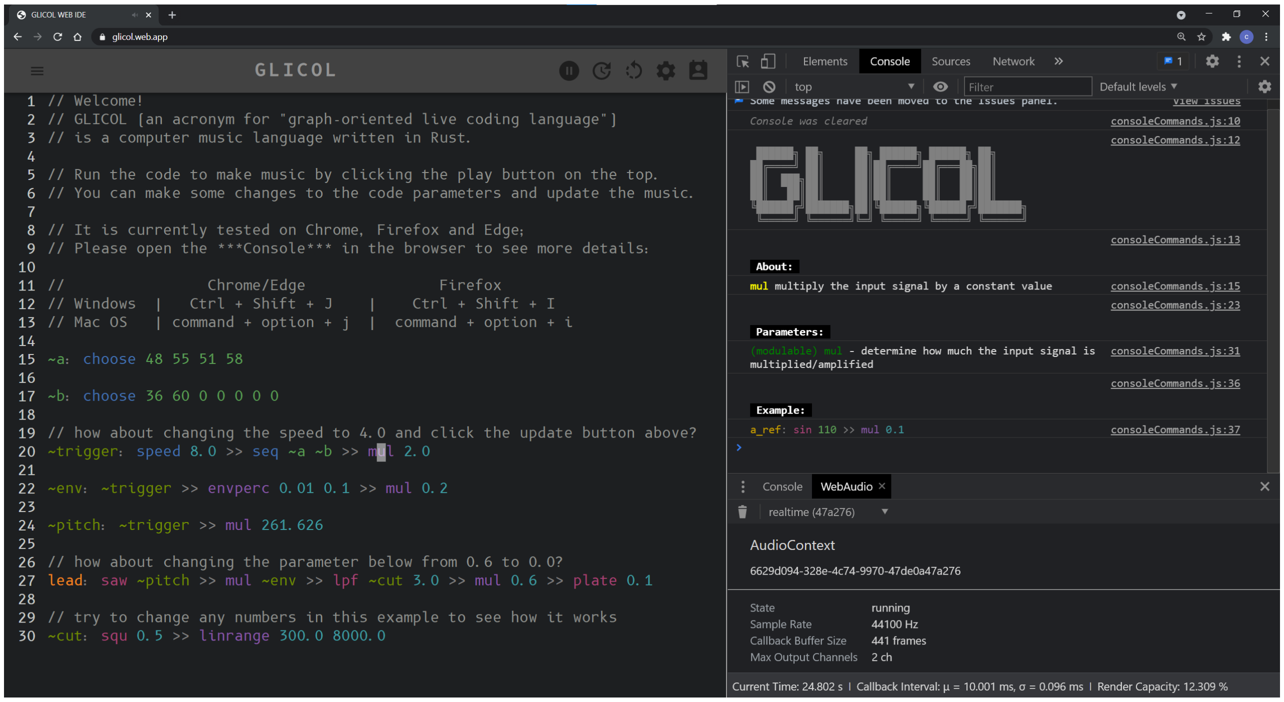
In live coding, the audio graph needs to be updated in real-time. In Glicol, we have chosen a WYSIWYG (what you see is what you get) approach as we believe that this can help the audiences better understand the code–sound relationship. This means that every time the user runs the code, the audio graph is always dependant on all current code. In the implementation, however, it would be a huge waste to the performance efficiency and would be prone to audio clicks if we reset the entire audio graph every time the update is scheduled.

The solution is to manage the nodes dynamically using the longest common subsequence (LCS) algorithm [2]. First, we parse the new code and process it chain by chain. When dealing with a chain, we compare it with the node by chain *HashMap* stored previously. The comparison has three possible outcomes. In the first case, the chain shows in the previous code but not in the new one. Then we will remove all the nodes in this chain from the graph and the side-chain information list. The second case is that this chain is a new one, so we can simply add all the node information to the graph. The last case is that this chain is a modified version in the new input code. Then we use the LCS algorithm to find out which nodes to add and remove, while keeping most of the nodes untouched in the graph.

Taking advantage of this dynamic node management al-

[6]https://bit.ly/3n2ehfI

[7]https://pest.rs/



```
// Welcome!
// GLICOL (an acronym for "graph-oriented live coding language")
// is a computer music language written in Rust.

// Run the code to make music by clicking the play button on the top.
// You can make some changes to the code parameters and update the music.

// It is currently tested on Chrome, Firefox and Edge;
// Please open the ***Console*** in the browser to see more details:

//                    Chrome/Edge               Firefox
// Windows  |    Ctrl + Shift + J    |    Ctrl + Shift + I
// Mac OS   | command + option + j  |  command + option + i

~a: choose 48 55 51 58

~b: choose 36 60 0 0 0 0 0

// how about changing the speed to 4.0 and click the update button above?
~trigger: speed 8.0 >> seq ~a ~b >> mul 2.0

~env: ~trigger >> envperc 0.01 0.1 >> mul 0.2

~pitch: ~trigger >> mul 261.626

// how about changing the parameter below from 0.6 to 0.0?
lead: saw ~pitch >> mul ~env >> lpf ~cut 3.0 >> mul 0.6 >> plate 0.1

// try to change any numbers in this example to see how it works
~cut: squ 0.5 >> linrange 300.0 8000.0
```

```
Some messages have been moved to the Issues panel.                 View issues
Console was cleared                                                consoleCommands.js:10
                                                                   consoleCommands.js:12
GLICOL
                                                                   consoleCommands.js:13
About:
mul multiply the input signal by a constant value                  consoleCommands.js:15
                                                                   consoleCommands.js:23
Parameters:
(modulable) mul - determine how much the input signal is           consoleCommands.js:31
multiplied/amplified
                                                                   consoleCommands.js:36
Example:
a_ref: sin 110 >> mul 0.1                                          consoleCommands.js:37
>
```

```
AudioContext
6629d094-328e-4c74-9970-47de0a47a276

State                  running
Sample Rate            44100 Hz
Callback Buffer Size   441 frames
Max Output Channels    2 ch

Current Time: 24.802 s | Callback Interval: μ = 10.001 ms, σ = 0.096 ms | Render Capacity: 12.309 %
```

Figure 7: The interface of Glicol Web-IDE. The editor has syntax highlights implemented with regular expressions. The browser console is creatively used to provide help documentation and command-based interaction. The WebAudio tab in Chrome DevTools can be used to monitor real-time audio performance.

gorithm, we further add a strategy to optimise the audio performance. In the pre-processing stage, the parameter of all the `mul` nodes, and oscillator nodes such as `sin` (sine wave oscillator), will be replaced by a control signal that contains a single `const` node. In this way, when the user changes the parameter of these nodes, it is the `const` node that will be removed from the graph and a new `const` node will be added, while the `mul` or `sin` node will remain in the graph. Then, within the `mul` or `sin` node, the previous audio state, e.g. the phase of an oscillator, can be retained. Thus, a smooth transit can be created.

3.4 Using Glicol to develop new nodes

For complicated nodes such as the Dattorro's reverb effect [7], we come up with a solution to write Glicol code insides Rust. As shown in Figure 5, the references in Glicol can be used as nodes too. Based on this feature, we create a special reference called ∼`input`, whose buffer can be manually updated in every audio block in the Glicol engine. With this design, we call use Glicol syntax to determine the behaviour of an engine and call the engine within a node definition to do the processing job. Therefore, we can use existing nodes in Glicol such as the `delayn` or the `delayn` nodes to build the Dattorro's reverb and wrap it within the `plate` node in Glicol.[8] We have also used *macros* in Rust to optimise the workflow. As an important feature in Rust, macros refer to the way of writing code that generates other code, often known as *metaprogramming*.

4. A BROWSER-BASED IDE

The language and audio engine we build in Rust is compiled to a WebAssembly module to run in the browser-based IDE (integrated development environment). To improve the user experience of the environment, we have implemented syntax highlighting in the code editor. We have also built in the documentation in the browser and added support for collaborative coding similar to QuaverSeries.

As Figure 7 shows, the browser-based code editor is developed with `Ace.js`.[9] When users click the **run** icon in Glicol, the code string will first be encoded into UTF-8 format. Then it will be sent to the AudioWorklet thread as an *Uint8Array* with a label `run`, using the `SharedArrayBuffer` feature in browsers such as Chrome or Firefox. Once the AudioWorklet thread gets the array buffers, it will call a function '`alloc_u8`' exported from the WebAssembly module. This function will create memory space for the code string. On the JavaScript side, we use the `array.set()` method to write the array data to the allocated memory location. Then we pass the pointer and the array size to the `run` function exported from Rust/WebAssembly. This function will call the Glicol engine inside to process the code string.

Similarly, we can pass the audio samples to the WebAssembly/Rust module as array buffers. First, users need to switch on the `use sample` option to fetch the audio samples from the Internet. These downloaded audio samples will be stored temporarily as JavaScript *Float32Array*. Then, these sample arrays can also be passed to the Rust/WebAssembly engine using the `SharedArrayBuffer`. Here, sending the audio samples requires the WebAssembly/Rust side to allocate memory locations for both the sample names and the sample data. The audio samples will be stored in a

[8]https://bit.ly/2WQht1C

[9]https://ace.c9.io/

HashMap and can be later used in the audio engine.

Users can also use the browser console for communication to the WebAssembly module. Adding samples can be done with the `addSamples(name, URL)` function export to the browser window. Beats per minutes can be set with the `bpm()` function. And the amplitude of each audio node chain can be set with the `trackAmp()` function.

Our preliminary user testing quickly uncovered the need to add easy access to documentation of the code. Our solution is to use the browser console for documentation. When a user opens the Glicol interface, all available nodes will be printed in the console. Similarly, when users load samples, all available samples will be shown. Users can also get the help of a node in the console (see Figure 7).

Based on our positive experiences with collaborative coding in QuaverSeries, we have also added this feature to Glicol. This feature is currently powered by Firepad[10], but we may switch to a decentralised solution in the future. Users can choose to enter the same Glicol 'room' where they can start collaborating immediately. Each user can decide when to run or update the code on their machines.

5. PERFORMANCE AND RELIABILITY

Although we have not yet done an extensive evaluation of the system, some measurements of the audio performance and some preliminary user studies can be presented.

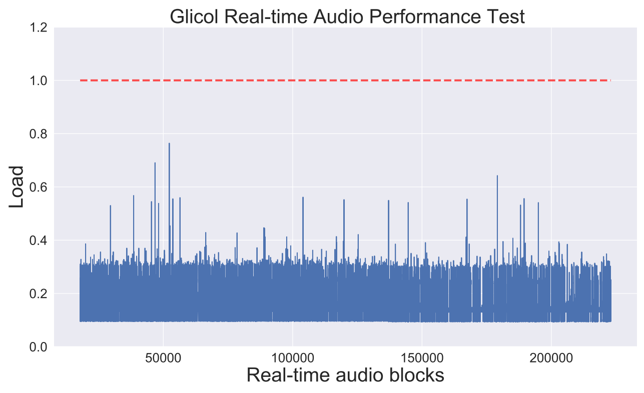


Figure 8: Glicol audio engine performance during a performance test with Chrome tracing tool. Only the period when the audio begins is shown.

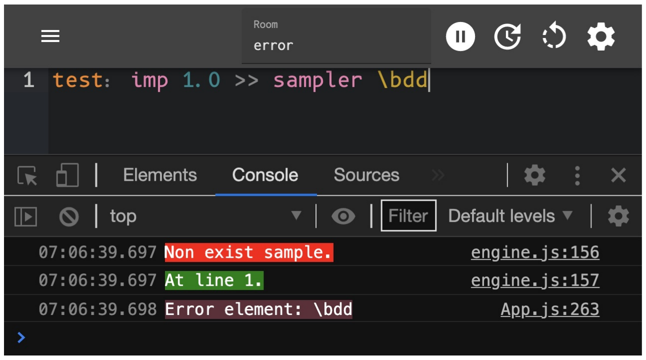
5.1 Real-time audio performance

In real-time audio processing, the audio is processed in blocks. In Web Audio API, the block size is 128, and our WebAssembly module needs to yield 256 audio samples (128 stereo frames) in each block time, i.e. the real-time budget. The audio load refers to the ratio between the render time of that block and the real-time budget. Thus, an audio load exceeding one can cause glitches in the audio.

Figure 8 shows the result of a performance test in Glicol, based on a live coding improvisation on top of the code displayed at the Glicol welcoming page (see Figure 7). The tracing tool[11] in Chrome is used to record the audio load and the result shows that the real-time audio performance is stable and efficient. Another way to test the audio performance of Glicol is to monitor the load value reported by the WebAudio tab in the Chrome DevTools in real-time. From the videos of Glicol recordings in this YouTube playlist[12], we can see that that the audio load of Glicol is typically stable and well below the limit.

5.2 Error handling

Handling code errors is imperative for a music system to be considered reliable. Fortunately, Rust offers a robust error handling solution with `Option` and `Result`. Each time the user makes an update, the engine will try to parse the code and make the graph. If there is an error, the result will be an Error type. Should this happen, the engine would use code from the last successful update and continue to play music using the previous code. Information about the error's type and position will then be passed back to the AudioWorklet thread along with the ongoing audio array. A message will also be printed in the console as is shown in Figure 9.



```
test: imp 1.0 >> sampler \bdd
```

```
07:06:39.697 Non exist sample.          engine.js:156
07:06:39.697 At line 1.                 engine.js:157
07:06:39.698 Error element: \bdd        App.js:263
```

Figure 9: Error messages shown in the console, with hints on where the error comes from and the error element.

6. DISCUSSION

Our experience with the development of Glicol is that Rust is a powerful language for audio-based live coding applications. It also provides new ways of handling errors, making the environment safe to run and efficient to program. Glicol is currently stable and deployed as a Web App[13] with documentation and tutorials available. In the following, we will reflect on how Glicol compares to some other live coding languages.

6.1 Glicol as an audio server

One of Glicol's main contributions is its graph-oriented syntax, together with the audio engine implemented seamlessly. With the help of Rust's famous zero-cost abstraction, we believe that this design pattern can bring a minimal performance loss for live coding language design. In terms of language abstraction, Glicol should be at the same level as audio programming languages developed in C++, such as SuperCollider. Furthermore, its syntax is optimised for live performances, and given its graph-oriented nature, Glicol's syntax can also be seen as an intermediate structure for creating higher-level customised languages. One difference

[10]https://firepad.io
[11]https://web.dev/profiling-web-audio-apps-in-chrome/
[12]https://bit.ly/3nUMYDH
[13]https://glicol.web.app

from SuperCollider is that we choose the WYSIWYG design mode in Glicol, followed by a dynamic node management solution to satisfy its use in live performances. As a comparison, in SuperCollider, the live coders need to determine which parts of the code should be executed, which is highly related to SuperCollder's *client-server architecture* under the hood [14]. Despite its popularity in the live coding community, in our experience, such an interaction mode is often hard for beginners to understand, and this is why we decide to experiment with the WYSIWYG mode as an alternative. In our tests with Glicol so far, the WYSIWYG mode works well both in live coding and quick prototyping for audio effects, and we believe that WYSIWYG is better for lowering the cognitive load and help the audience to build a code–sound relationship. Admittedly, this will be an interesting topic to be further investigated in the future user studies of Glicol.

6.2 Limitations and future work

Even though Glicol is fully functional in its current state, it has several limitations. For example, it has limited input capabilities. We are planning inputs for MIDI, and it would also be helpful to receive OSC messages. The graph-oriented syntax can also be extended. One work in progress is to support parameters in the references used as nodes (see Figure 5), so that users can define nodes directly in Glicol Web-IDE. Also, we plan to organise a formal user study. Here the idea is to test Glicol on people with various levels of musical and live coding experience. The user study will also guide the development of tutorials, targeted at different user groups. In parallel to the user study, we will develop more APIs for users to create their own customised language based on Glicol. When a stable version is finished, we also aim to run the language on microcontrollers, such as Bela. Also, we plan to port it to Python, using the wasmer[14] package, to leverage various types of machine learning libraries. The idea is to explore how it is possible to train virtual agents for different types of collaborative performance. We may start offline first and work towards a real-time implementation.

7. ACKNOWLEDGEMENTS

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.

8. REFERENCES

[1] 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.

[2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000.

[3] F. Bernardo, C. Kiefer, and T. Magnusson. An audioworklet-based signal engine for a live coding language ecosystem. In *Proceedings of Web Audio Conference (WAC-2019)*.

[4] R. Boulanger and V. Lazzarini. *The audio programming book*. the MIT Press, 2010.

[5] H. Choi. Audioworklet: the future of web audio. In *ICMC*, 2018.

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

[7] J. Dattorro. Effect design, part 1: Reverberator and other filters. *Journal of the Audio Engineering Society*, 45(9):660–684, 1997.

[8] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[9] Q. Lan and A. R. Jensenius. Quaverseries: A live coding environment for music performance using web technologies. In *Proceedings of the International Web Audio Conference (WAC)*, pages 41–46. NTNU, 2019.

[10] N. Laurent and K. Mens. Parsing expression grammars made practical. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 167–172, 2015.

[11] S. Letz, Y. Orlarey, and D. Fober. Compiling faust audio dsp code to webassembly. 2017.

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

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

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

[15] 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.

[16] A. McPherson. Bela: An embedded platform for low-latency feedback control of sound. *The Journal of the Acoustical Society of America*, 141(5):3618–3618, 2017.

[17] W. C. Pirkle. *Designing Audio Effect Plugins in C++: For AAX, AU, and VST3 with DSP Theory*. Routledge, 2019.

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

[19] S. Shirinivas, S. Vetrivel, and N. Elango. Applications of graph theory in computer science an overview. *International journal of engineering science and technology*, 2(9):4610–4621, 2010.

[20] S. Yi, V. Lazzarini, and E. Costello. Webassembly audioworklet csound. In *4th Web Audio Conference, TU Berlin*, 2018.

[14]https://github.com/wasmerio/wasmer-python