Updating maximilian.js for Modern JavaScript Live Coding

Louis McCallum
CCI, UAL
l.mccallum@arts.ac.uk

Mick Grierson
CCI, UAL
m.grierson@arts.ac.uk

ABSTRACT

The original and widely used Maximilian[1] signal processing library was written in C++ and provided bindings to JavaScript. When `WebAudio's ScriptProcessorNode` was discontinued, previous incarnations of maximilian.js[2] were rendered unusable in contemporary browsers. In this paper, we provide details of the new implementation including updates to allow for live coding without interrupting the audio output and a focus on web-based collaborative editing.

INTRODUCTION

Whilst much development has been made with browser based live-coding for symbolic pattern manipulation (e.g. Strudel (Roos and Mclean 2022), WebChucK (Mulshine et al. 2023), Estuary (Oborn et al. 2021)), the evolution of web-based audio processing has been marked by significant developments and challenges, particularly when it comes to signal processing and audio generation. The original Maximilian library, a cornerstone in this domain, was initially developed in C++ and adapted for web applications through JavaScript bindings (Byszynski et al. 2017). However, the deprecation of the `WebAudio's ScriptProcessorNode` posed a substantial challenge, rendering previous versions of maximilian.js de facto deprecated in modern browsers. This setback coincided with the development of the Sema library, under the ambit of the AHRC MIMIC project, which was successfully utilising the robust core of Maximilian for an efficient audio engine compatible with `AudioWorklets` (Bernardo et al. 2019). Rather than begin from nothing, we decided to use the engine updated for Sema as a foundation for a new iteration of maximilian.js in efforts to overcome the limitations imposed by ever-tightening web standards that make the browser a hostile environment for audio development.

The initial design of the Maximilian library, while not explicitly tailored for live coding, presents a robust and comprehensive audio framework that is well-suited for adaptation to a dynamically updated environment. Indeed, the adoption of Maximilian by the explicitly live coding library Sema exemplifies this potential. It is important to note that Sema's approach necessitates users to contend with the challenging task of language design, which, despite its powerful capabilities, is a niche requirement that diverges from the straightforward API of Maximilian and the established familiarity its regular users have with it. Our objective has been to leverage the audio engine of Maximilian and user-friendly API to facilitate an engaging and expressive live coding experience. Moreover, born out of remote, pandemic collaborations we aimed to allow for playful and collaborative online connections.

SIGNAL PROCESSING ON THE WEB

Keeping the signal processing within the web-based live coding environment, despite its inherent complexities (Grierson, M et al. 2019), offers notable advantages over traditional methods involving external Digital Audio Workstations (DAWs) or synthesis engines. One of the primary benefits is the elimination of third-party software installations, significantly enhancing accessibility and ease of use for educational purposes and general user engagement. The sole requirement to access the full capability of the library is a web browser. Additionally, when considering networked interactions, browser-based

[1] https://github.com/micknoise/Maximilian
[2] https://mimicproject.com/course/making-music/overview

processing naturally lends itself to remote and collaborative performance. To maintain consistency across local executions, it can be favourable to have all audio generation happening within one shared library.

Primarily, to access sample level audio processing in the browser, the `WebAudio` API provided a `ScriptProcessorNode` to be added into the audio graph. This was initially leveraged by Maximilian for its JavaScript adaption. For full details see ICMC 2019 paper (Bernardo, F et al. 2019).

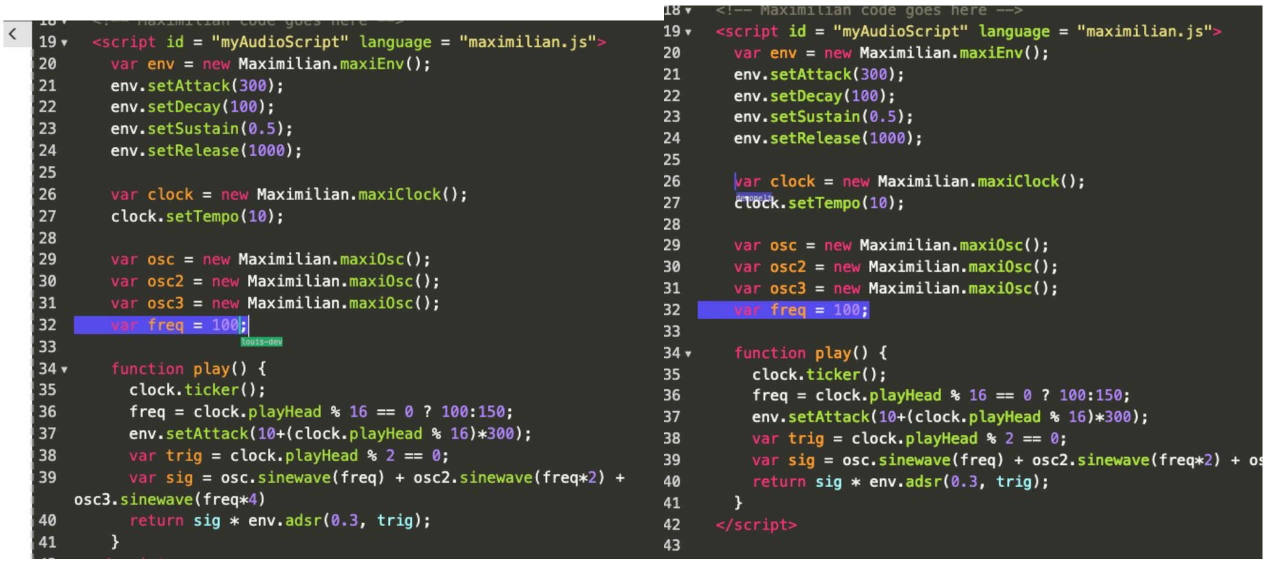
The adoption of `AudioWorklets` over the `ScriptProcessorNode` in web-based audio processing represented a significant advancement in the field. The primary reason for this shift is the enhanced performance and flexibility offered by `AudioWorklets`. Unlike `ScriptProcessorNode`, which operates on the main browser thread and is prone to audio glitches and performance bottlenecks, `AudioWorklets` run in a separate audio processing thread (Choi, 2018). This allows for more reliable and efficient processing of audio data, significantly reducing latency and ensuring smoother audio playback. Furthermore, `AudioWorklets` provide a more robust framework for complex audio processing tasks, enabling developers to create more sophisticated and nuanced audio applications. This shift aligns with the evolving needs of web-based audio applications, demanding higher performance, precision, and stability, especially in scenarios requiring real-time audio processing. Given its clear advantages, although they ran it parallel for a while, in 2021 `ScriptProcessorNode` was deprecated in most major browsers[3]. This rendered current iterations of maximilian.js unusable without major refactoring.

MANY LIVE CODERS IN THE SAME DOCUMENT ON THE MIMIC PLATFORM

Since 2018, the authors, in collaboration with other researchers, have been instrumental in the development of the MIMIC creative coding platform[4]. This platform, following the paradigm established by its predecessor, CodeCircle, enables users to write JavaScript code and immediately visualise its outcomes, a feature later seen in other projects including the popular p5.js web editor. Interactive refreshing where the whole document is re-executed in relation to user input has proven highly beneficial for artists engaged in creating interactive visual works (Grierson, 2018). However, this approach presents challenges when applied to audio processing.

In REPL (Read-Eval-Print Loop) editors, like the Sema Playground (Bernardo et al. 2019) and Strudel Live Code (Roos and Mclean 2022), the entire script is updated upon user request. This global update methodology, while efficient in some contexts, poses a challenge for collaborative live coding. A fundamental requirement in such a setting is the ability to re-execute only specific portions of a document, thus avoiding interference with other contributors' code. The MIMIC interface addresses this by allowing selective re-execution of single or multiple lines of code, utilising JavaScript's `eval()` function. This feature is akin to the functionality offered by the SuperCollider editor (McCartney, 2002) enabling users to alter specific code elements (such as a variable controlling an oscillator's frequency) without necessitating a full restart of the audio engine or causing disruptions in audio output. This selective execution also ensures that multiple users working on the same document (the "Shared Code" paradigm (Nilson, 2007)) can independently update their code segments without the risk of crashing the performance due to incomplete or erroneous code inputs from others.

[3] https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode) [4] https://mimicproject.com/



```
<script id = "myAudioScript" language = "maximilian.js">
  var env = new Maximilian.maxiEnv();
  env.setAttack(300);
  env.setDecay(100);
  env.setSustain(0.5);
  env.setRelease(1000);

  var clock = new Maximilian.maxiClock();
  clock.setTempo(10);

  var osc = new Maximilian.maxiOsc();
  var osc2 = new Maximilian.maxiOsc();
  var osc3 = new Maximilian.maxiOsc();
  var freq = 100;

  function play() {
    clock.ticker();
    freq = clock.playHead % 16 == 0 ? 100:150;
    env.setAttack(10+(clock.playHead % 16)*300);
    var trig = clock.playHead % 2 == 0;
    var sig = osc.sinewave(freq) + osc2.sinewave(freq*2) +
osc3.sinewave(freq*4)
    return sig * env.adsr(0.3, trig);
  }
```

```
<!-- Maximilian code goes here -->
<script id = "myAudioScript" language = "maximilian.js">
  var env = new Maximilian.maxiEnv();
  env.setAttack(300);
  env.setDecay(100);
  env.setSustain(0.5);
  env.setRelease(1000);

  var clock = new Maximilian.maxiClock();
  clock.setTempo(10);

  var osc = new Maximilian.maxiOsc();
  var osc2 = new Maximilian.maxiOsc();
  var osc3 = new Maximilian.maxiOsc();
  var freq = 100;

  function play() {
    clock.ticker();
    freq = clock.playHead % 16 == 0 ? 100:150;
    env.setAttack(10+(clock.playHead % 16)*300);
    var trig = clock.playHead % 2 == 0;
    var sig = osc.sinewave(freq) + osc2.sinewave(freq*2) + os
    return sig * env.adsr(0.3, trig);
  }
</script>
```

Figure 1. Two browsers updating synchronously, with coder name and re-execution highlighted

MIMIC allows for collaborative coding by employing an operational transforms approach for document editing, facilitated by the JavaScript library ShareDB[4]. Changes made to the document are propagated to all collaborators via WebSockets, ensuring that each participant's local version of the document is synchronously updated. This includes live code partial re-executions (which are sent as transforms to the document). As a result, while code execution occurs on each user's local machine, updates to both the state through re-execution and the source code itself are shared among all performers. Local code execution with collaborative, real-time document synchronisation is an expressive and valid approach to collaborative live coding in a web-based environment. Here, we prioritise synchronising of source code over synchronising of clocks (an ongoing and possibly insurmountable challenge) in a way that can work in remote-online or collocated-live contexts (for example if only one performer is broadcasting an audio out) (Roberts 2022).

LIVE CODING IN A SAMPLE LOOP WITH MAXIMILIAN

Maximilian was originally created to facilitate the teaching and learning of complex audio digital signal processing methods in C++ for novice coders. Maximilian borrowed key features from existing creative coding environments including Processing[5] and openFrameworks[6], in order to provide a bestfit API combining buffer-level audio digital signal processing capabilities alongside rapid-prototyping features for synthesis, sampling, analysis and audio/music information retrieval tasks. Furthermore the API was designed to make use of existing and recognised music technology nomenclature rather than conventional textual programming or engineering conventions for method names, such as the names of specific generators (`sinwave`, `sinebuf`, `ADSR`, `line`), filters (`lores`, `hipass`, `SVF`). and techniques (granular, stretch, mosaic). This was done in order to lower the barrier to entry for those with a good understanding of creative music technology engineering principles but for whom C/C++ was still an emerging skill.

Another key factor in its design was that all objects and functions would update and maintain their own phase as much as possible, reducing the need for setup variables. Objects behave by generating single value updates when called, even in cases where vectors are processed under the hood (such as FFT processing). This was done to encourage a single sample thought process amongst teachers, learners and coders, to encourage people to think about what happens when complex audio calculations are being done one step at a time and provide a method for rapidly designing them with minimal code. This allowed

[4] https://share.github.io/sharedb/
[5] https://processing.org/
[6] https://openframeworks.cc/

instructors to introduce notions of timekeeping with constant reference to sample rate, foregrounding buffer-level precision in all mathematical operations, allowing for higher quality output as much as possible, with optimisations left to the compiler.

These approaches allow users to create complete synthesiser and sequencer systems in a single line of code by nesting operations within each other, each parameterising the other. This makes it extremely simple to live code complex systems rapidly, providing a platform that can be used to create music immediately that would otherwise take much longer, and without the need for pattern libraries. It also allows users to introduce entirely bespoke buffer-level signal processing routines and interoperate these with existing Maximilian objects. This makes it relatively simple even for beginners to extend the Maximilian library once the core concepts have been understood – for example, a sample-and-hold clock can be created with a phasor cast to an Int, and this can index a look up table at a modulating rate to create a polyrhythmic structure that parameterises a waveform, filter and envelope in three lines of code.

There are challenges with this approach, focussing as it does on continuous maths applied to music technology concepts, and users with less experience or interest in the domain can struggle. Overall, however, this continues to be a relatively minor issue when compared to the potential benefits of rendering buffer-level signal processing as immediate, intuitive and creative an approach as possible. Further, the familiarity that many have with Maximilian, and its ease of use when compared to other buffer-level complex DSP tools makes it a good choice for live coding. Possibly as a result, the library has been used as the basis for several live coding experiments in the past[7], but the issue of controlling execution, in particular for the relatively new field of collaborative live coding, continues to be an area of development that offers significant challenges. Others working along a similar line include the Rust based Glicol (Lan and Jensenius, 2021).

MAXIMILIAN TO SEMA TO MAXIMILIAN.JS

The comprehensive development of the Sema engine by Bernardo et al. is well documented in the literature (Bernardo F 2019). However, our decision to integrate certain advancements back into a distinct maximilian.js library, while concurrently developing additional functionalities, is motivated by several key factors. Firstly, it is imperative to retain the familiar API of Maximilian for its substantial existing user base, as Sema's primary contribution is the affordance of novel end user live code language design. This level of abstraction, while highly interesting to a specialised audience, does not align with our current objectives. We have previously discussed our preference for the MIMIC Code editor's approach to live coding over Sema's functionality of full document update, which, despite its ability to maintain symbolic state, results in undesirable interruptions in audio output due to the restarting the audio signal upon each re-execution (Grierson, M and Kiefer, C 2011).

Moving to the new engine leads to a significant departure from previous iterations of maximilian.js as there is necessity for user code to be transmitted to a separate audio thread for execution, as opposed to running as standard JavaScript on the main thread. In the context of most interactive web applications, it is critical to maintain elements of the original HTML5 ecosystem, such as DOM access, third-party library integration, and user interactions, and to facilitate smooth and synchronous communication between these elements and the audio processing tasks. Our challenge was to author an API that accommodates this requirement, while staying as true as possible to the original Maximilian API and avoiding the introduction of additional complexities for novice users, who often comprise the library's primary audience (Grierson, M 2009).

As much as possible, we want the library to be platform-neutral allowing the greatest flexibility to users. This also applies to the MIMIC site, as although optimised towards interactive creative coding, it is by no means a maximilian.js specific sole use platform. This is to say that we did not wish maximilian.js to require a tailor-made environment to run well and designed its live coding API to fit in with a

[7] e.g. https://vimeo.com/35012356

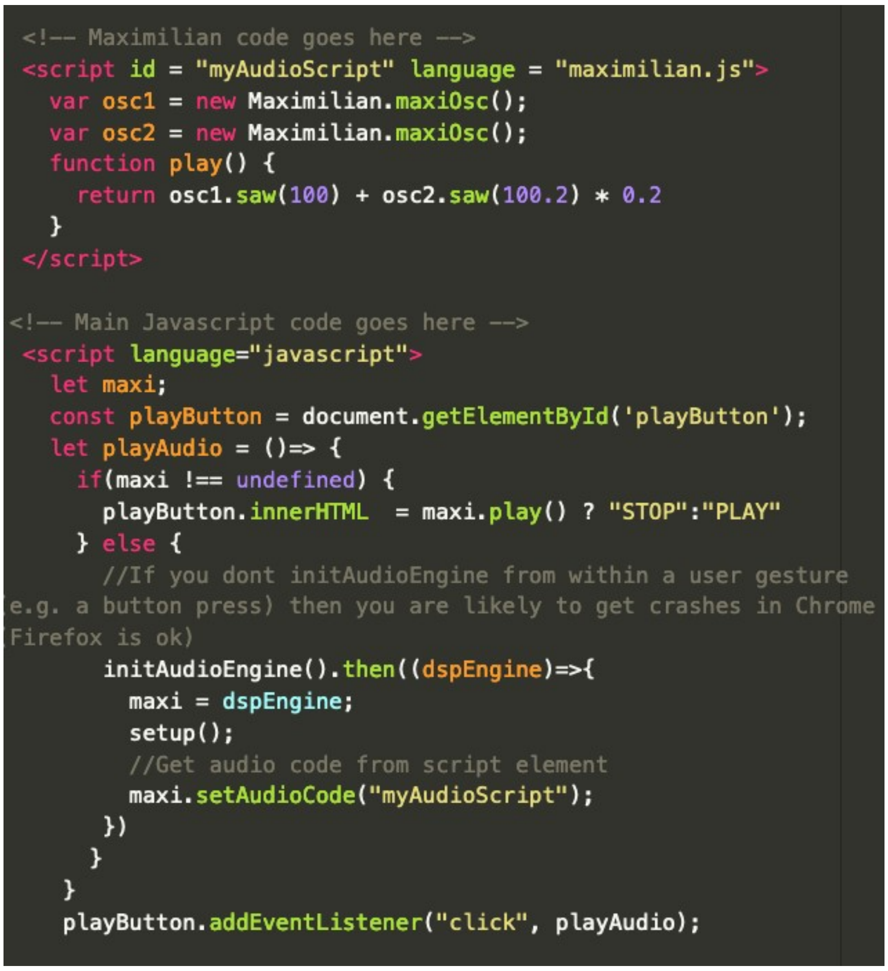
generalpurpose platform that allowed for re-executing code in library agnostic web projects. We see this as a

massive bonus as it will not require participants to learn a new editor, can be deployed elsewhere, and easily integrate into other web projects without the developers needing to make specific integrations on a per library basis. Crucially for the audio-visual practice common in live coding performance, this involves graphics packages such as Three.js[8] or p5.js[9].

All audio code is written separately and then transmitted to the AudioWorkletNode as a string. This is achieved using the built-in post messaging functionality. Given that post messaging does not guarantee synchronicity and consistent delivery times, which are critical for audio rate signal processing, the engine has used `SharedArrayBuffers` to make a shared ring buffer between the threads. This approach ensures reliable data transfer between the main thread and the audio thread at high refresh rates, thereby maintaining the efficiency and integrity of the audio processing workflow.

For maximum flexibility execution we allow for 3 ways to write audio code and pass to the library

1. As code within a script element. This allows the code to be in the same document, for text highlighting of the JavaScript to still work and for a clean logical separation of audio and main code. See fig above.
2. As a string literal
3. As a URL to the text hosted remotely. This can be used to access audio code from separate tabs in MIMIC documents



```
<!-- Maximilian code goes here -->
<script id = "myAudioScript" language = "maximilian.js">
  var osc1 = new Maximilian.maxiOsc();
  var osc2 = new Maximilian.maxiOsc();
  function play() {
    return osc1.saw(100) + osc2.saw(100.2) * 0.2
  }
</script>

<!-- Main Javascript code goes here -->
<script language="javascript">
  let maxi;
  const playButton = document.getElementById('playButton');
  let playAudio = ()=> {
    if(maxi !== undefined) {
      playButton.innerHTML  = maxi.play() ? "STOP":"PLAY"
    } else {
      //If you dont initAudioEngine from within a user gesture
e.g. a button press) then you are likely to get crashes in Chrome
Firefox is ok)
      initAudioEngine().then((dspEngine)=>{
        maxi = dspEngine;
        setup();
        //Get audio code from script element
        maxi.setAudioCode("myAudioScript");
      })
    }
  }
  playButton.addEventListener("click", playAudio);
```

[8] https://threejs.org/
[9] https://p5js.org/

Figure 2. Example of Sending maximilian.js code to the audio thread

UNINTERRUPTED AUDIO WITH SEPARATE STREAMS

Our experience with collaborative live coding, specifically with multiple performers editing a single document, has identified distinct requirements essential for an effective live coding environment in network music. These requirements, some resonating with established norms in live coding and network music work, include:

1. The environment must support multiple audio streams, allowing updates without affecting the existing streams. This feature is vital for dynamic and improvisational performance scenarios.
2. Updating algorithms and symbolic states should occur without any interruption to the ongoing audio output.
3. The system should maintain audio output without interruption even when invalid code is executed, thus supporting experimental and exploratory coding practices.
4. The system should allow for updates to individual parts of the document. For instance, while the entire document may not be valid in its current state, specific code segments may be updated and re-executed creating valid updates to the musical output.
5. It is crucial to keep the code in sync between performers, ensuring a cohesive and coordinated performance.
6. Code execution should be shared synchronously between performers.

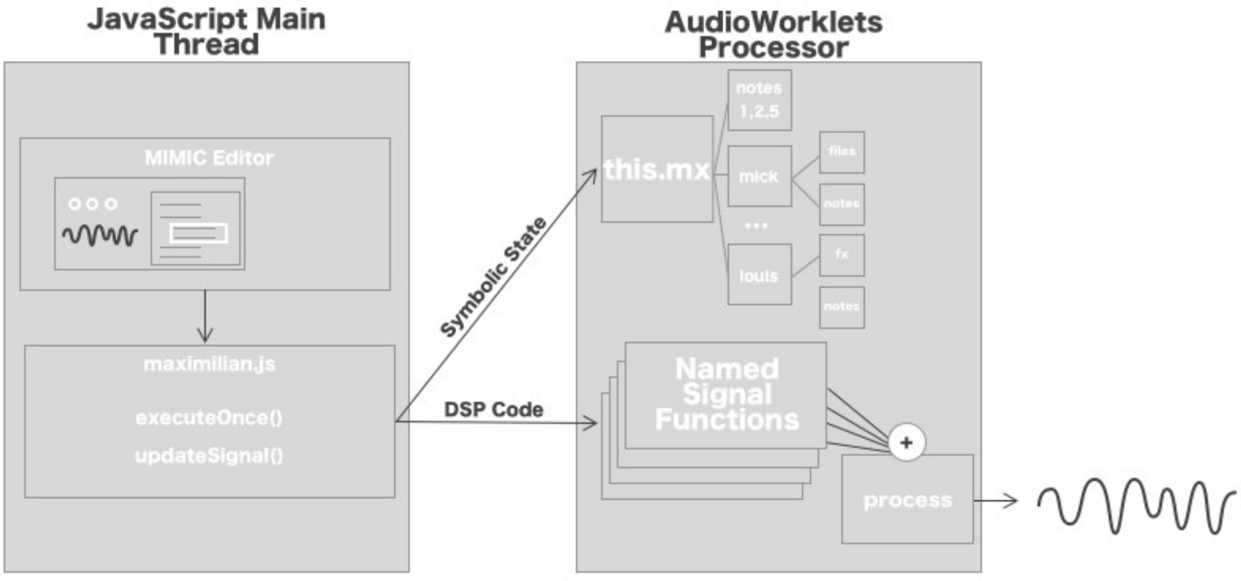


Figure 3. System diagram for execution of code between threads

Addressing the challenges highlighted in prior studies (notably by Roberts (2022) and Bernardo (2019)) regarding dynamic code updates in the `AudioWorklet` thread, our approach and API design focus on setting up a system that meets these requirements. We have introduced a `this.mx` namespace within the `AudioWorkletProcessor` to facilitate simple and extensible access to symbolic and named variables on the audio thread. This JavaScript object, capable of being extended to prevent collisions (for instance, adding namespaces like `this.mx.louis` or `this.mx.mick`), can maintain and dynamically update any JSON-stringifiable objects for use in music production on the audio thread.

As established, a fundamental requirement for a maximilian.js program is a `play()` function, executing at audio sample rate and returning audio samples. This precision is a key feature of its utility as a music library. The existing Sema engine is designed with a singular `play()` function which can cause discontinuities when redefined during new code execution. Our solution is the `updateSignal()` function, enabling the creation of named streams, each with its own `play()` function. These streams are evaluated and mixed on the audio processor for output.

For multiple collaborators, this design means each performer can manage one or more personal streams, updating or experimenting without impacting others' outputs. This aligns with our first requirement. Channels can be muted by replacing them with a `play()` function that returns zero. Should performers choose not to update a named `play()` function and instead wish to update an element within the this.mx object, they can utilise the `executeOnce()` function, satisfying our second requirement. To ensure requirement three, we have implemented a system where code is evaluated before integration into the audio chain, rejecting any code that returns errors.

The MIMIC coding platform, as previously described, addresses requirements four, five, and six, providing a robust foundation for our collaborative live coding environment.

REFLECTIONS AND EVALUATION

Over the past three years, the current iteration of our library and platform has been utilised in a variety of performances, notably at the Network Music Festival (Grierson, Yee-King and McCallum 2020), NIME 2021 (Grierson, Yee-King and McCallum 2021), and the TOPLAP Winter Solstice 2023[10]. In this section we aim to provide critical insights into the advantages and limitations encountered during these implementations.

With regards to the namespaces for variables allowed for simultaneous yet non-interfering operations among users, all synchronised to the same clock. This feature was crucial in avoiding crashes due to inprogress code (e.g., non-executable scripts) while allowing concurrent updates in other document sections.

Choosing the general-purpose MIMC platform offers substantial advantages, such as reduced learning curves, broad accessibility to creative coders, and a single codebase. Particularly, the ease of integrating visual elements with general-purpose code without requiring separate integrations was a notable benefit. However, this flexibility can lead to the absence of specialised features that might streamline a platform dedicated to live coding or music production. Despite this, the current balance between flexibility and specificity appears to be effective.

As highlighted by Roberts (2022), effective communication among remote performers is vital. Our current setup lacks an integrated chat feature, leading us to use external tools like Zoom for spoken communication whilst performing. While this approach is effective in synchronising performance, it occasionally could be seen to interfere with the primary listening experience of the audience, the music. This being said, in terms of the transparent philosophy of live coding, introducing contextual elements of performer communication can provide added context for the audience.

Our reliance on ShareDB for document synchronisation has not been without challenges, necessitating additional scaffolding to avoid corruption. Future stress tests are planned with larger groups to identify and address potential technical and creative bottlenecks in collaborative scenarios.

This paper primarily focuses on the technical aspects, and future work will explore the creative affordances of the library more comprehensively.

[10] https://solstice.toplap.org/

CONCLUSION

In this paper we have presented the requirements for bringing the popular and powerful Maximilian C++ library into use for a modern JavaScript live coding experience. We have identified several key requirements for such an update and detailed how the newly developed maximilian.js can allow creative and powerful sample level code collaboration in real time, especially in tandem with the MIMIC platform.

Acknowledgments

This research was supported by AHRC/ UKRI MIMIC project grant

REFERENCES

Bernardo, Francisco, et al. "An AudioWorklet-Based Signal Engine for a Live Coding Language Ecosystem." 2019.

Byszynski, Michael, Mick Grierson, Matthew Yee-King, and Leon Fedden. "Write Once Run Anywhere Revisited: Machine Learning and Audio Tools in the Browser with C++ and Emscripten." In Web Audio Conference 2017, 21-23 August 2017, Queen Mary University of London, United Kingdom, 2017.

Choi, H. "Audio Worklet: The Future of Web Audio." In International Conference on Music and Computing, 2018.

Grierson, Mick, Matthew Yee-King, Louis McCallum, Chris Kiefer, and Michael Zbyszynski. "Contemporary Machine Learning for Audio and Music Generation on the Web: Current Challenges and Potential Solutions." In International Computer Music Conference, 16-23 June 2019, New York, 2019.

Grierson, M., and C. Kiefer. "Maximillian: An Easy to Use, Cross Platform C++ Toolkit for Interactive Audio and Synthesis Applications." Proceedings of the International Computer Music Conference 2011, University of Huddersfield, UK, 31 July - 5 August 2011.

Grierson, Mick. "Creative Coding for Audiovisual Art: The CodeCircle Platform.". In The Routledge Research Companion to Electronic Music: Reaching out with Technology, edited by Simon Emerson, London:Taylor Francis 2018

Grierson, Mick, Matthew Yee-King, and Louis McCallum. "Executive Order." In NIME 2021. PubPub, 2021.

Lan, Q., and A. R. Jensenius. "Glicol: A Graph-Oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet." In Web Audio Conference, 2021.

McCallum, Louis, Mick Grierson, and Matthew Yee-King. "Local Code for Local People." Live performance. Network Music Festival, July 5-18, 2020. https://networkmusicfestival.org/programme/performances/local-code-for-local-people/.

McCartney, J. "Rethinking the Computer Music Language: SuperCollider." Computer Music Journal 26, no. 4 (2002): 61–8.

Mulshine, Michael R., Ge Wang, Jack Atherton, Chris Chafe, Terry Feng, and Celeste Betancur. "Webchuck: Computer Music Programming on the Web." In New Interfaces for Musical Expression, 2023.

Nilson, Click. "Live Coding Practice." In Proceedings of the 7th International Conference on New Interfaces for Musical Expression (NIME '07). Association for Computing Machinery, New York, NY, USA, 112–117, 2007.

Ogborn, David, et al. "Estuary 0.3: Collaborative Audio-Visual Live Coding with a Multilingual Browser-Based Platform." Web Audio Conference 2022 (WAC2022), Cannes, France, 28 June 2022.

Roberts, Charlie, Ian Hattwick, Eric Sheffield, and Gillian Smith. "Rethinking Networked Collaboration in the Live Coding Environment Gibber." In NIME 2022. The University of Auckland, New Zealand: PubPub, 2022.

Roberts, C., and G. Wakefield. “Tensions and Techniques in Live Coding Performance.” In The Oxford Handbook of Algorithmic Music, edited by A. McLean and R. Dean, 293–317. Oxford: Oxford University Press, 2018.

Roos, F., and A. McLean. “Strudel: Live Coding Patterns on the Web.” Proceedings of the International Conference on Live Coding, Utrecht: ICLC, 2023.