OSSIA SCORE 3

Jean-Michaël Celerier
ossia.io
Talence, France
jcelerier@ossia.io

Myriam Desainte-Catherine
Univ. Bordeaux, LaBRI- UMR 5800, CNRS, INRIA
Talence, France.
myriam@labri.fr

Pia Baltazar
ossia.io
Talence, France
pia@ossia.io

ABSTRACT

The ossia system has been introduced in 2015 as a notation for interactive scores. We present the result of seven years of usage and improvements to the *ossia score* software which acts as both an editor and player for such scores, and how it morphed from a simple OSC-only control sequencer to a fully-fledged multimedia system supporting live audio and video processing, live-coding with multiple embedded programming languages, communication with a variety of software and hardware and adaptations for more traditional music creation such as support for varying tempo and time signatures. In particular, we mention a few "original sins" and implementation mistakes done at the beginning of the software engineering process and how they had to be fixed.

1. INTRODUCTION

In 2015, the foundations for a method of authoring interactive scores were presented to the TENOR conference, in a paper titled: "OSSIA: Towards a Unified Interface for Scoring Time and Interaction"[6].

This paper presents the result of seven years of active use in a variety of artistic settings, what has held and what hasn't, and how the ossia paradigm for interactive scores was extended. It gives an overview of the various research aspects that were studies for interactive scores so far, reports on their status and talks about upcoming research.

From the very beginning, the implementation software, originally named **i-score** and renamed **ossia score** – to highlight the from-scratch rewriting of the system and its inclusion in a larger software ecosystem – , has been conceived as a platform for fostering art-science research.

1.1 A primer on interactive scores

The visual syntax of interactive scores as defined in ossia is composed of a few basic elements, visible in Figure 1 which are combined together by the composer to create scores. Due to user feedback and evolutions of the system, some names have been changed since [6]. These changes are mentioned here:

- A *process* is an object which performs a computation over time; it is similar to e.g. a Max/PureData object or SuperCollider ugen. A specificity of processes is that they can be used to define both non-temporal content (e.g. an audio filter such as a distortion, an oscillator...) or content with an implicit temporality (e.g. an automation curve, a MIDI piano roll) directly.
- An *interval* (renamed from Constraint in [6]) allows to define the span of time over which a set of processes will run.
- A *time sync* (renamed from Time Node in [6]) allows to synchronize the start and end of multiple intervals, either at a fixed time or by waiting for an external event: OSC message, mouse click, etc. In the latter case, the external event is represented by a visual object called *trigger*, shown at the top of the time sync.
- A *condition* allows to enable or disable a set of intervals at run-time depending on the truth value of an expression at the time at which it is evaluated in the score (after the end of the previous intervals and before the start of the next intervals).
- A *state* (renamed from Event in [6]) indicates the start and end of an interval. States can carry messages such as OSC packets that will be sent immediately when the state is reached during the execution of the score.
- Time syncs and intervals form a graph which we call a *scenario*. A *scenario* is a process; this means per the definitions of intervals and processes that intervals and scenarios allow to define a hierarchical system.

2. SEMANTIC EVOLUTION

We detail here how the visual language and system have evolved since its introduction in 2015 and the areas of focus for the research and development.

2.1 Loop semantics

The original loop semantic, described in [5] was through a specific process which implemented a looping behaviour for a single temporal interval, isolated from its parent. This added a hierarchy level, and separated entirely the looped content, from the overall loop duration.

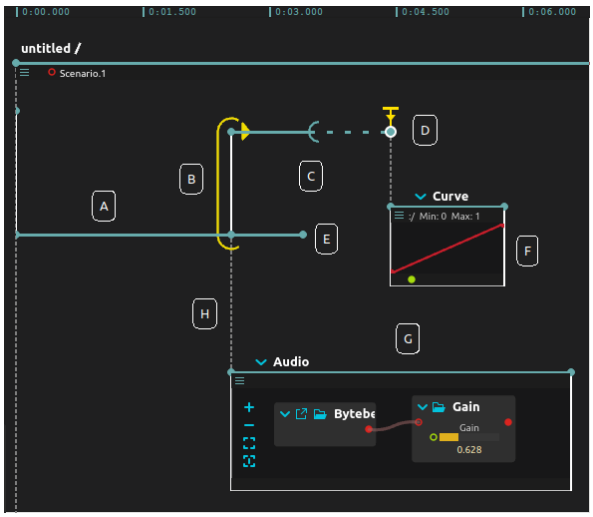

Figure 1: Visual syntax of ossia score. **A**: an *interval* of fixed duration. **B**: a *condition*. **C**: an interval of variable duration; the minimum can in this example be adjusted with a visual handle. **D**: the blue dot is a *state*. The white circling around it indicates that the state contains messages to be sent. The yellow T on top indicates a *trigger*. **E**: an empty state. **F**: an interval which contains an automation *process*. **G**: an interval which contains two connected audio processors, represented as dataflow nodes. **H**: a *time sync*. Note that this score is itself contained in a *scenario* process, itself a child of an *interval* which is the hierarchical root of the score.

After years of experimentation, while we observed that this method was able to cover a fair amount of use cases, we also noticed how unwieldy it was for artists as it made scores harder to read. The visual similarities between the looped content, and the parent interval containing the loop object, made understanding the behaviour of a score at a glance more complicated. In addition, looping could not easily be changed "on the fly": artists would often want to simply loop a sound for some time, but our original method required conceiving the temporal structure of the score beforehand.

Study of the use cases for loops led us to a separation of the original loop process into two distinct features:

- Looping an individual process through a simple UI control: each process with a temporal information can be looped: sound files; automations, etc. The process' actual duration is decoupled from its parent interval's duration. This kind of loop is outside of the temporal structure of the scenario, it is merely a filter which can be applied to any process, and can be toggled at will at any point in the user interface.

- A generic go-to mechanism for the scenario. Visually a new primitive, this is actually just an interval of duration zero, for which the restriction of "always going forward" is relaxed. This mechanism enables more generic cases than the previous loop primitive, and makes writing state machines in *ossia score* an easy task, as the end of an interval can now start any other interval in a score.

Figure 2: Example of tempo and musical grid variations: multiple time-line have different tempo curves. The first timeline will follow the tempo of the parent. The second will follow its own tempo curve. The third one takes its tempo from a low-frequency oscillator: the speed of that timeline will constantly oscillate. The grid switches from $\frac{4}{4}$ at the first bar, to $\frac{7}{8}$ at the fifth bar; bar 4 is an implicit $\frac{1}{4}$ bar as the start of bar 5 can be arbitrarily moved to simplify user interaction.

2.2 Musicality, polyrhythms, quantization

For the longest time, even though the interactive scores project was born out of a musical context, it did not provide any tooling to help the composer used to traditional and western musical cues. The software provided no beat grid, no notion of tempo, and no quantization to allow synchronizing multiple competing elements on a single beat which would be determined by the grid in which the musicians are playing.

In *ossia score 3*, we introduce the ability to specify metric modulations in the score (time signature changes through a musical grid, and tempo changes through automation curves). Both features can be seen in Figure 2.

Both musical grid and tempo curve are optional properties of an interval. If an interval does not have either, it and its children processes will stay synchronized with respect to the parent interval's grid, recursively. At any point, it is possible to dissociate a child interval from the parent's musical grid or tempo, in the first case by enabling a custom grid (through a GUI widget visible when the user selects an interval), in the second case, by adding a tempo process to it.

The tempo process is by default an automation curve which allows to define a fixed tempo variation. This process also provides inputs: these inputs allow to control the tempo, or the playback position, through the score. This enables writing scores where the execution speed of a part comes from an external input, a mathematical function, or any combination of either.

Quantization is another feature which is propagated hierarchically: by default, interactive triggers will wait for the next quantized time as defined by their parent, for instance the start of the next bar or next quarter note. Any interval can redefine it instead, to allow some parts of a score to

quantize to the next bar, and other parts to quantize to the next eighth note relative to this interval's internal musical grid and current execution time. For instance, given a fixed BPM of 60, if a trigger's boolean expression becomes true at 32.3 seconds since the interval started, it will actually trigger at 33 seconds if this interval's quantization is set to quarter notes, and 32.5 seconds if it is set to eighth notes.

2.3 Data processing, combining data flow and time flow

The original implementation focused on control signals: sending and receiving OSC and MIDI messages, for instance. It quickly became obvious that having to start a whole other software for the sake of playing a synchronized sound file and setting up OSC communication was cumbersome for the users of the software. This led to research about introducing an audio pipeline into the system. A second requirement quickly came up: applying audio effects such as VST plug-ins or Faust signal processors, as using systems such as Soundflower or JACK for routing audio between software was also too complicated for a part of the target user base. By then it became obvious that a built-in dataflow pipeline was required, with the usual affordances provided by the patching software paradigm: processes (called nodes, objects, ugens in other systems) with multiple input and output ports (controls, audio or MIDI inputs and outputs), connected together through virtual cables.

The main question was then: how to make a dataflow pipeline, which describes an invariant computation, fit as part of an interactive score where processes constantly change over the execution of the score. The solutions for this were twofold:

- In addition to the ability to connect cables showcased in Figure 3, the ports in *ossia score* can read and write data directly from a global environment, through OSC-like addresses. When a cable is connected, that address is overriden.

- Cables have two axes of configurability: direct and delayed, strict and glutton. Direct (the default) means that if process A and B are running at the same time in real-world clock, B processes the output of A just like in a traditional patcher. Delayed means that the cable acts as a delay line between A and B: if A starts a second before B, B will receive all the data that was produced by A since it started and thus be a second late. Glutton (the default) means that the connection of a cable has no impact on whether A and B runs, strict means that if A and B are connected, and either is not running, the other will also be disabled. This is mainly useable as a performance optimization, to make sure that complicated signal processing pipelines are not kept running if the data source has not started yet.

This has led to the introduction of a dual-visualization for intervals: at any point, the user can switch between the time-line view, where the horizontal axis is used to represent the evolution of time, and the so-called nodal view, which shows all the processes as nodes similar to traditional

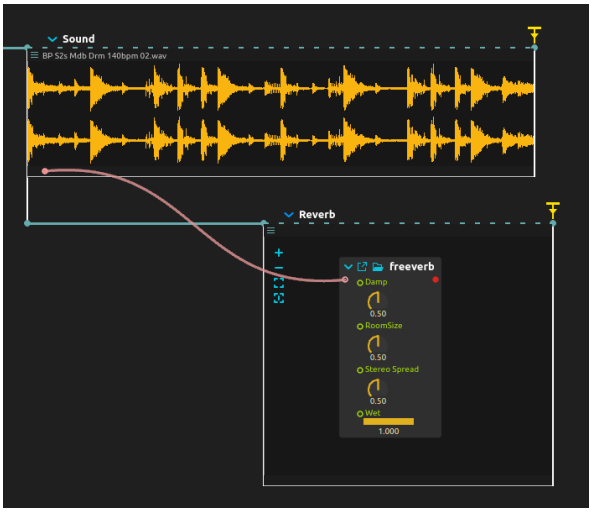

Figure 3: Applying an effect to a sound file with an explicit cable.

patching software. This way, the user can focus on editing temporal behaviour such as synchronizing automations with soundfiles, etc. in a timeline, and on editing complex signal processing flows in a more adapted user interface.

2.4 Live visuals

Shortly after artists querying about "simply playing back a sound file", the same question happened for video. Prototypes of efficient OSC-controlled video players were made [1] yet this also proved insufficient in terms of synchronization precision and capabilities.

Thus, score 3 introduces support for video playback and video processing through the same data-flow system as for audio, showcased in Figure 4. This has been done with the Qt RHI library. [2] which provides a small wrapping layer over modern GPU APIs: Direct3D 11, Vulkan, Metal (and more traditional OpenGL).

Some complexity for the implementation of this feature comes from the fact that processing happens on the GPU, not on the CPU: working with a modern graphics API is about preparing a set of commands that are being send to the GPU for rendering at a regular interval. The GPU generally works at a 60 Hz rate while the rest of *ossia score* follows the audio buffer rate. The implementation works by creating a mirror graph of GPU nodes which lives in its own thread. While cables look like they pass texture data, what they actually do is instruct the source node to render on a texture stored in the input of the sink node; this enables multiple nodes to render on the same target. Render order and blending options are not made available to the user yet and is a work-in-progress for an upcoming minor release.

Multiple outputs are possible; each output will only render the nodes that are directly connected to it. For instance, it is possible to render to multiple windows, or to both a window and a system such as Spout or shmdata.

[1] `https://github.com/OSSIA/ossia-videoplayer`
[2] `https://www.qt.io/blog/graphics-in-qt-6.0-qrhi-qt-quick-qt-quick-3d`

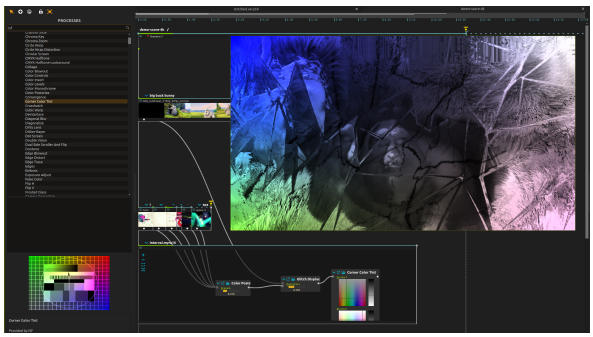

Figure 4: Mixing multiple video sources and applying GPU video effects through shaders following the Interactive Shader Format specification.

Leveraging the GPU has some advantages: the author has for instance prototyped a particle renderer process which was able to maintain 60 FPS for ten million particles with a GTX 1080 graphics card.

One feature which is not available is the delayed cables for GPU textures: this is because unlike audio or control data which can be expected to be bufferable for reasonable durations on the order of minutes or even hours, video data memory usage grows too fast to make this viable. [3]

2.5 Pragmatisms for live playback of interactive scores

Scores in *ossia score* until now allowed for what could be called "known unknowns" in a show. The classic use case is: a sound has to trigger ten seconds after an actor reaches the center of the stage. The actor being human implies that the time taken to reach the center of the stage is unknown: the durations of the sound start cannot be fixed as a duration from the beginning of the play, only from when the center of the stage is reached. Thus, the *ossia score* user can add a trigger point and use for instance position sensors to encode in the score the actual conditions for the sound starting to play, as well as add bounds for the time that the actor must take, to make sure that the score continues.

What remained was "unknown unknowns": consider a catastrophic case where all the sensors stopped working for unforeseen reasons. It does not make sense to account for such cases during the authoring of the score; yet in the midst of the action, it makes sense to still have a safeguard to allow a stage manager to take control of things manually, even if this does not respect the semantics originally written score: the show must go on.

Thus, multiple features were introduced over time to allow to override the written score as a last measure during playback:

- Triggers can be triggered by hand and through a remote interface.
- Interval speed can be adjusted live, independently for each interval.

[3] To give a reference point, merely storing 60 frames (one second) of raw 4K (3840 × 2160) textures in GPU memory with ARGB float32 texture format uses upwards of 7 gigabytes; most consumer GPUs do not even have that much memory. In addition, most GPUs are very limited in terms of maximum texture count: many high-end NVidia and AMD GPUs are limited to 4096 total allocations for a given GPU context

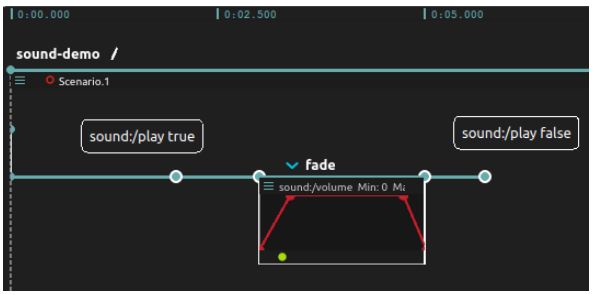

Figure 5: A score which controls an external sound file player through OSC.

- The "main playhead" can be moved: this is similar to transport in the usual audio-video software.
- Intervals can be started manually at any point in the score.
- States can be sent manually at any point in the score.

Starting an interval will follow quantization settings; implementation of quantization for starting states and global transport remains to be implemented. This enables very simple live-looping systems to be put in place.

Transport is an interesting feature, as it is in a sense equivalent to starting the execution of a software from any point in its code: it is not possible to ensure that it will always make sense. Two semantics are provided by *ossia score*, that the user can choose in the settings:

- Simply start executing from the given point visually.
- Try to compute the state in which every external system should be at this point, send them this state, and start executing.

Consider the score of Figure 5: it first sends a message to start an external sound playback device. If one wants to start the score from the 3 second mark, the `play` message has to be sent beforehand for the execution to make any sense.

2.6 Web

Having *ossia score* work on a web browser has been the result of a long process, which started in 2015 with Qt's NaCL port, then migrated to Asm.js and finally to WebAssembly, the web standard which enables compiling native C++ code and running it in a web browser.

The result of this work can be found at the address `https://ossia.io/score-web` which allows to write and execute simple scores in a web browser. Not all features are currently available, as more dependencies need to be recompiled for the WebAssembly target. The objective of this work is to allow *ossia score* to export a score as a web page so that it can be easily experienced over the internet.

2.7 Computational performance

One of the first supported system for audio effects in *ossia score* was Faust [8], the well-known DSP language. The software embeds the Faust compiler, which itself leverages

LLVM, the compiler framework. The C++ compiler `clang` is also based on LLVM; from there, it was relatively easy to introduce a C++-based add-on system [2].

The end goal of that add-on system is to ensure that the users of the software get the most performance out of their hardware: most native C++ software is compiled for a basic baseline in order to maximize compatibility. This means that someone with a recent Intel Skylake CPU most of the time does not benefit of the advanced vector instructions in that CPU, since it would break compatibility with people using older CPUs, unless the program author instructs the compiler to build multiple versions (which would multiply the size of the binaries). In contrast, by embedding the compiler inside the software, we can work towards rebuilding the hot paths directly on the user's computer, as we can probe the exact CPU instructions that are available there.

Right now, this feature is used only for external add-ons; when it will have undergone enough testing on enough hardware the plan is to migrate the various signal processing plug-ins which the software provides to that system.

2.8 Live-coding and scripting in scores

A certain focus was put on using score as a platform for live-coding. Multiple languages are embedded in the software and can be recoded on-the-fly [3], during the execution of the score:

- The first one was Javascript, through Qt's ES7 interpreter, and allowed to quickly devise which features made sense as actual processes implemented in native code.
- The Faust language integration supports live recompilation.
- The math expression language ExprTK is used [9].
- Likewise for the ISF shaders used for GPU visuals.
- PureData has been embedded thanks to libpd [1]; patches can be edited live. The `send` and `receive` objects are used to create GUI controls in *ossia score* which can be used to score parts of a PureData patch over time [4].
- C++ code can also be live-recompiled. This comes up in three different processes: Bytebeat (which executes Bytebeat expressions), Texgen (which generates 2D textures for visuals) and CPP JIT (a generic C++ object which can be used to implement general objects with any kind of input or output ports).

3. IMPLEMENTATION NOTES

Some fairly deep changes came up during the software development process, which make sense to discuss with the broader community mainly so that pitfalls in which the project fell can be avoided by others who would encounter similar situations.

3.1 Evolution of time-tracking

When the development was started, time was counted in milliseconds stored in double values. This was a mistake: due to the way double arithmetic works, we encountered after a few years cases where the execution would be stuck for scenarios running over the course of multiple days, for installations. This was due to the expectation $t + \epsilon > t$ not holding anymore for orders of magnitude of ϵ which start to be relevant to our use-cases. For instance, the following assertion does not hold for double-precision floating-point: $3600 * 24 * 1000 < 3600 * 24 * 1000 + 10^{-9}$. That is, after merely a day, adding a nanosecond does not have an effect anymore. Thus, we migrated in version 2 to using audio samples for storing time internally in the engine, while staying with the original time format for the user interface in order not to break the save format. This worked better, but now made it harder to work in preparation of the video format support, as many common video formats timestamps are not easily divisible by audio rates (for instance, 24 images per second versus 44100 samples per second). Thus, for *ossia score* 3, a migration to flicks was done: this unit, originally introduced by Oculus VR, is defined as a common multiple of most time formats used in media production. One second holds 705600000 flicks; timestamps for every common audio sample rate and video frame rate can be represented without loss of precision while keeping flicks an integral value.

3.2 Save format and JSON woes

ossia score leveraged from the very beginning JSON as its save format, as it is widely understood, and easily readable. However, the migration to flicks mentioned before had an unforeseen impact. The JSON specification does not mandate a minimal precision for storing numbers; in particular, the library we had been using, QJson which is part of Qt, makes the choice of converting every integral number to floating-point which had not been an issue until then, but looses precision for flicks representing long durations.

It was hence necessary to migrate to a library which supported 64-bit integers to allow us to safely store durations without data loss and without breaking compatibility with the existing format – the RapidJSON library was chosen.

3.3 Threaded networking

The first implementation of network protocols was threaded. Incoming OSC messages for instance would be processed in a specific thread; callbacks would be called from that thread. Users of the *libossia* library would have to take care of thread-safety for their own data structures. While this work was done in *ossia score*, in practice doing it this way was a mistake: many environments would only support single-thread callbacks (Python, Unity3D) and require a lot of busywork for repatriating data from the network threads to the main thread. A recent change was to use the Asio [7] library which provides an event-loop approach where the user of the API can choose whether processing must occur in a separate thread or not; doing so from the beginning would have saved dozens of hours of work with threads. Addition-

Figure 6: Nebula.

ally, using the Asio library in *libossia* allowed us to leverage its support for multiple networking protocols: besides the traditional UDP, *libossia* and score now support OSC over TCP, Unix sockets (both stream and datagram), WebSockets and serial port, with various encodings. Unix sockets in particular are interesting when staying on the same machine: the author measured an improvement of 10-15% when measuring how many messages can be transmitted per second when compared to UDP on loopback network interfaces.

3.4 Forked libraries

Various libraries forked during the development process, mainly to provide more efficient versions of the originals without having to retain API compatibility which was not useful in our case since the software was being developed from scratch:

- A fork of Ross Bencina's OSCPack library for OSC performance, adaptability to different back-ends and modernization for compatibility with more recent C++ standards. [4]
- A fork of RtMidi [10] into libremidi [5] with many improvements: greatly reduced memory allocations, support for more backends (UWP on Windows, WebMidi with Emscripten, raw ALSA for sending large SYSEX messages), notification of newly available devices, and modernization for compatibility with more recent C++ standards. This fork also merges and improves the ModernMIDI library which provided SMF reading and writing.

4. ARTWORK MILESTONES

We would like to discuss here artworks made with *ossia score* which provided useful as feedback mechanism: they evidenced the need for new features which were developed in tandem and can now benefit the greater community.

4.1 Nebula

> Nebula is a sensory installation. The visitor enters a dark room. In the centre of the room, a black cylinder is placed on the floor. A coloured and luminous mist sculpture emerges from the top of the inert object.

Nebula (Figure 6) is an art installation created by the duo Les Baltazars. It is one of the first artwork which used the software at a "non-toy" scale: the score lasts for 30 minutes and contains hundreds of automations controlling evolving patterns of lights and mist. It was useful as a benchmark of the software, in particular for the UI rendering which had to show thousands of lines on-screen while maintaining acceptable performance, and was used a lot as part of optimization work, and for the device tree, which contains more than a thousand parameters.

[4] `github.com/jcelerier/oscpack`
[5] `github.com/jcelerier/libremidi`

Figure 7: Quarrè.

4.2 Rain of Music

Rain of Music is a work-in-progress hybrid human-machine opera, which involves soundpainting for both human and robot performers. Technically, it has been used as a way to test various integrations of protocols inside *ossia score*, and most notably direct serial port control for the Metabots, the robots used for the performance.

4.3 Quarrè

Quarrè (Figure 7) by Pierre Cochard was one of the first works which explored multi-user interaction for a single score. Up to five participants are each given a phone; the score broadcasts instructions to each phone which will show distinct user interfaces over the duration of the score. The user interfaces have widgets which give back feedback to the score, which then uses it to control generative sound processes to create eerie ambiances and textures.

4.4 HERMÈS

HERMÈS by João Svidzinski [11] explores multi-user interaction, over the internet instead of in a same-room setting. It used *ossia score*, OpenStageControl and Max/MSP to provide a performance during the Journées d'Informatique Musicale 2021.

4.5 Carrousel Musical

The Carrousel was the largest incentive for introducing a native audio data-flow system in *ossia score*. It is a merry-go-round located in "L'Abbaye aux Dames" in Saintes, France, with multiple instruments and sensors. The songs contain

Figure 8: Jeu de la Marelle.

Figure 9: Phonemacore performance.

parts which record and replay the young player's musical performance with an improved rhythm. Dozens of VST instruments such as Kontakt and various effect busses are output in spatialized sound with low-latency, and an additional DMX light show. The project was first prototyped in Ableton Live and Max4Live but the performance was not adequate, which prompted the implementation of a custom solution.

4.6 Jeu de la Marelle

Laurence Bouckaert and Olivier Moyne devised an augmented game of hopscotch in their piece "Jeu de la Marelle" (Figure 8). Each square is a pressure sensitive area that is able to detect the impact of the stone thrown by the player. They can sense the weight distribution along their X and Y axis. Throwing the stone or stepping on either of these tiles then triggers a specific behavior of the system, through the RGB lighting on the ground, the octophonic sound system surrounding the player, and the visual projection on the wall. The score was able to run on a Raspberry Pi 4 computer connected to a multi-channel sound card, a video projector and an ArtNet node for controlling LED strips. Live audio and video processing is done with Faust and ISF shaders respectively.

This project was one of the impulsions for having a complete video pipeline, and making sure that the software was performant enough to be useable from low-power embedded computers such as Raspberry Pis.

4.7 Phonemacore

Phonemacore (Figure 9) was a musical project started in 2018 to explore the possibilities of *ossia score* for adding interactivity to popular-ish music (metal with simple $\frac{4}{4}$ and $\frac{12}{8}$ signatures). The performers were a guitarist, a keyboardist and a drummer: the band Phonema. Bass tracks were stored in the score. The song had 7 parts; a participant chosen from the public would play a video game made for the song. Whenever a level would be completed, the musicians would move on to the next part of the song, and keep playing until the player reaches a high enough score. This work was used as a basis to implement musical metrics, tempo changes, and quantization in the software, as those all had to be worked around during the creation of the piece.

4.8 Twitch live-scoring sessions

The author regularly presents live sessions on a Twitch channel [6], where a score is created and played "live": the execution of the score runs at all time; process creations, connections and Faust, JS, ISF scripting are all done while the score is running. It is mainly used as a debugging exercise, to make sure that live editing is reliable enough for allowing live-coding performances to take place.

5. CONCLUSION

We gave an overview of the state of the research on the applicability of the ossia system for interactive scores of various forms.

An upcoming work is the extension to less art-centric use-cases: a student is currently working on protein diffusion sonification with the system, which opens a set of issues related to parsing and processing efficiently large amount of data.

Another ongoing issue is applying generative and procedural techniques for creating scores programmatically: providing an efficient API for allowing users to generate scores from Javascript code both offline and during the execution is a priority.

Acknowledgments

This work has been financed by: SCRIME, ANR, ANRT, Blue Yeti, Epic Megagrants, SNI, Région Nouvelle-Aquitaine, and kind private donations.

References

[1] Peter Brinkmann et al. "Embedding pure data with libpd". In: *Proceedings of the Pure Data Convention*. Weimar, Germany, 2011.

[2] Jean-Michaël Celerier. "A Cross-Platform Development Toolchain for JIT-Compilation In Multimedia Software". In: *Proceedings of the Linux Audio Conference*. Stanford, USA, 2019, pp. 37–42.

[3] Jean-Michaël Celerier. "Leveraging Domain-specific Languages in an Interactive Score System". In: *Sonic Arts Today* 10.1 (2018), pp. 28–33.

[6] `https://twitch.tv/jcelerier`

[4] Jean-Michaël Celerier. “Patches in a timeline with ossia score”. In: *Vortex Music Journal* 9.2 (2021), pp. 1–14.

[5] Jean-Michaël Celerier, Myriam Desainte-Catherine, and Jean-Michel Couturier. “Graphical Temporal Structured Programming for Interactive Music”. In: *Proceedings of the International Computer Music Conference (ICMC)*. Utrecht, The Netherlands, 2016, pp. 377–380.

[6] Jean-Michaël Celerier et al. “OSSIA: Towards a Unified Interface for Scoring Time and Interaction”. In: *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. Paris, France, 2015, pp. 82–91.

[7] Christopher Kohlhoff. *Boost.Asio*. 2008. URL: `https://www.think-async.com`.

[8] Yann Orlarey, Dominique Fober, and Stéphane Letz. “Faust: an efficient functional approach to DSP programming”. In: *New Computational Paradigms for Computer Music*. Paris, France, 2007, pp. 65–96.

[9] Arash Partow. *C++ Mathematical Expression Toolkit Library (ExprTk)*. 2000. URL: `http://www.partow.net/programming/exprtk/`.

[10] Gary P Scavone and Perry R Cook. “RtMidi, RtAudio, and a synthesis toolkit (STK) update”. In: *Proceedings of the International Computer Music Conference (ICMC)*. 2005, pp. 327–330.

[11] João Svidzinski. “HERMES v2 - WEB CONCERT COLLABORATIF EN TEMPS RÉEL”. In: *Journées d’Informatique Musicale 2021*. AFIM. Remote conference, France, July 2021.