Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.21

Patches in a timeline with ossia score

Jean-Michaël Celerier

ossia.io | France

Abstract: Handling of time and scores in patchers such as PureData, Max/MSP has been an ongoing concern for composers and users of such software. We introduce an integration of PureData inside the *ossia score* interactive and intermedia sequencer, based on libpd. This integration allows to score precisely event that are being sent to a PureData patch, and process the result of the patch's computations afterwards in *score*. This paper describes the way this integration has been achieved, and how it enables composers to easily add a temporal dimension to a set of patches, by leveraging both the computational power of PureData and the temporal semantics of the *ossia* system, in order to create complex compositions.

Keywords: pure data, ossia score, timeline, interactive scores, patching

Received on July 1st, 2021. Approved on: November 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres. Creative Commons CC-BY-NC.

A core concern when writing music in patch-centric software is the creation of scores. That is, how does one sequence events, create transitions, and more generally define variations over time for the patch's parameters and inputs in a scripted, non-algorithmic way. The question of defining what constitutes a score for interactive and intermedia software and hardware has been studied extensively in the ossia project. In this paper, we will show how the result of this work can be applied to PureData, in order to improve on the current ways to author and execute scores in real-time for Pd patches, as this is a long-standing issue in the community, described in detail by Miller Puckette, PureData's author:

> In its most succinct form, the problem is that, while we have good paradigms for describing processes (such as in the Max or Pd programs as they stand today), and while much work has been done on representations of musical data (ranging from searchable databases of sound to Patchwork and OpenMusic, and including Pd's unfinished "data" editor), we lack a fluid mechanism for the two worlds to interoperate. (PUCKETTE, 2004).

This paper will first present the possible ways to write and execute such scores in music-centric patching environments, mainly PureData, Max/MSP, OpenMusic. Then, it will introduce the free and open-source *ossia score* software for interactive and intermedia score authoring. Finally, the integration of PureData in *ossia score*, a novel development based on the well-known *libpd* library (BRINKMANN, 2011), will be described in detail.

1. Scoring patches

1.1 Typology of scoring approaches

The conducted literature review brings us to a two-fold separation of scoring approaches, which we will call *external* and *internal*. We define external scoring approaches as any scoring system that runs outside of the patching environment itself. In contrast, internal approaches embed the score system inside the patch, either as primitives of the patcher environment or as external objects.

These two methodologies imply vastly different mindsets for the authors: in the first case, the primary interaction while writing the score will be with a software distinct from the patcher; the patch – or patches, as this approach generally allows to score multiple separate patches in a single score – are individual instruments or effects. In contrast, in the second case, the primary authoring environment is the patcher; multiple score parts can be embedded in the patch and triggered with actions and events originating from the patch software, such as, in the PureData case, bangs, [metro] etc.

We posit that these two approaches generally invite the composers to different styles of authoring and may lead to substantially different artworks.

1.2 External approaches

The most common way to score a patch is by using an external communication channel. For instance, sending MIDI or OSC messages from an external scoring system or even a hardware sequencer can already provide a first level of scoring to e.g. instruments authored as patches.

An important development which occurred during the last decade is the integration of Max/MSP into the Ableton Live sequencer. Whereas the previous approach provided a weak connection between the patcher and the scoring software, this approach allows for a much tighter integration of the patcher (Max) into a software usable for scoring (Ableton Live) due to its timeline. The main drawback is that both software are proprietary.

Another way to embed a patcher into an existing host is through the common plug-in specifications, such as Steinberg VST, AudioUnits, etc. This is the approach for instance followed by patching software such as Usine, or with Camomile (GUILLOT, 2018), which embeds Pd inside a VST or AU plug-in so that any compliant digital audio workstation (DAW) can load it. The main benefit of this approach is its broad compatibility: most music software support loading such plug-ins. This means that the composer can very continue using his usual DAW and import bits and pieces of Pd wherever needed. However, the communication interface between audio plug-ins and DAWs is fairly limited. For instance, the VST 2.4 specification only allows exchanging floating point values between 0.0 and 1.0, which makes it tedious or impossible to exchange string messages, bangs, and other non-float messages beyond what the VST specification allows.

Finally, some DAWs provide hybrid integration of a custom patching mechanism. This is the case for the commercial DAW Bitwig Studio with the Grid, and for *ossia score* as we will show afterwards.

1.3 Internal approaches

Conversely, due to the extensibility of most patcher software with externals, it is possible and even common to have externals dedicated to implementing a scoring mechanism as objects of a patch. Bach (AGOSTINI, 2015 and GHISI, 2017) is such an environment, for Max/MSP. It contains an extensive set of objects allowing to embed, author and playback western or even custom graphical scores directly from within Max.

Miller Puckette proposed as far back as 2002 using Pd as a score language (PUCKETTE, 2002), using its primitives for defining and playing back graphical structures. More recently, Gemnotes (KELLY, 2011) introduced an actual staff-like musical notation system based on a custom textual language and an external into PureData.

Finally, Patchwork and its successor OpenMusic (ASSAYAG, 1999) are patch-based systems for computer-aided composition: the objects provided by the system are all geared towards generating pieces of a score through algorithmic means. More recently, OpenMusic also acquired reactive semantics (BRESSON, 2014) in a way that brings it closer to real-time systems such as Pd.

1.4 Implications of each approach

The core benefit of the external approach is that it generally does not need any additional modification or addition of plug-ins to the patcher software to work. For instance PureData has built-in support for MIDI and OSC message input. This makes this approach very suitable for teaching, as it removes a friction point with people novice to PureData composition.

In contrast, the internal approach, may require additional set-up but tends to offer more freedom and of course a deeper integration with the patching environment; an expert user of a patching software will feel more at home with scoring primitives implemented as plug-ins to the patcher, than with an entirely different software.

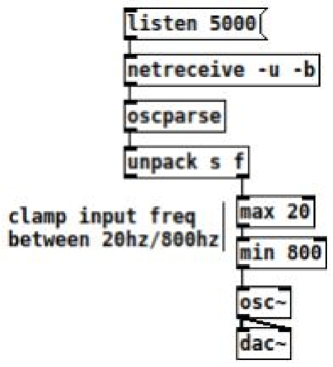
There are a few drawbacks to the external approach: the first is the lack of reflection (in the computer science meaning) on the inputs and outputs of the patch. This means that the person sending OSC or MIDI messages must check the documentation of the patch manually in order to discover which OSC parameters, MIDI control changes etc. are available, and what are their range of acceptable values. For instance, Fig. 1 showcases a trivial example of patch receiving OSC messages to generate a sound at a given frequency: the patch has to take care to manually limit the input as the protocol allows someone to send arbitrarily high or low values besides the acceptable range for the virtual instrument. In contrast, for the internal approach, one can generally easily connect the temporal scoring of a part of a song, to events in the patch, simply by using the same kind of connections between objects they are used to.

The second issue is the management of the project files and system – running multiple distinct software for a piece means that, for instance, their save file must be kept in sync for the final score, that one must take care of the order in which the software must be started for the case of automated performances where the score software sends a "play" message to the patch. This increases friction and difficulty for non-computer-science enthusiasts who must now handle a set of software issues entirely unrelated the artistic work. In the internal approach, this issue does not exist as there is only one software and document, which contains everything.

Finally, a third issue is that network or MIDI messages are by their nature asynchronous, and in the case of OSC messages, often sent over unreliable communication channels – even if band-aids such as time-stamping can help improve accuracy, it is in the general case not possible to reach the level of accuracy, consistency and reproducibility that one gets with a synchronous system.

The libossia library provides a set of objects allowing to give a specification to the network inputs / outputs of a patch (in PureData, Max/MSP and a set of other common creative coding software such as Processing, OpenFrameworks...) which can help alleviating some of the issues encountered here, thanks to the OSCQuery protocol which allows the parameters of a patch to be reflected over an internet protocol such as HTTP or WebSockets.

FIGURE 1 – Example of a Pd patch that receives a frequency information via OSC and then uses it to synthesize a sine wave which is output to the computer's speaker. The frequency is clamped manually between 20 Hz and 800 Hz, as there is no way in the OSC protocol to ensure that a parameter is fixed in a given range.



Source: Jean-Michaël Celerier

1.5 Motivation

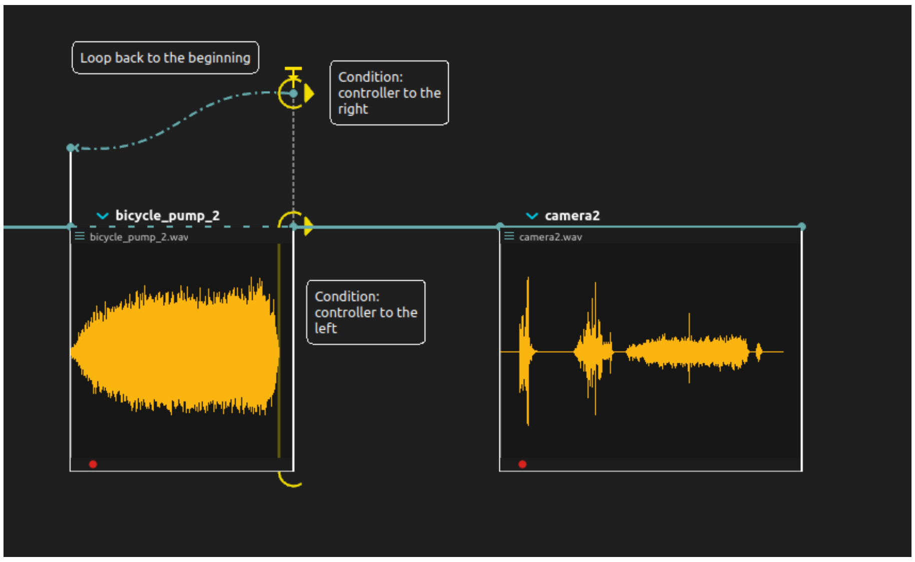
The motivation for this research is to provide an external, score-first environment tailored to PureData, which would be able to score its data types over time with as much temporal accuracy as possible, and able to manage complete scores involving multiple separate patches. We propose to do such a development in the *ossia score* software system, described thereafter, as it has been specifically tailored for this kind of embedding.

2. The ossia software system

The *ossia* project, introduced in (Celerier, 2015) and detailed in (Celerier, 2018) traces its roots back to Boxes (Beurivé, 2000), Iscore (Allombert, 2008). It is composed of the *ossia score* sequencer and *libossia* software library. The sequencer has three specificities:

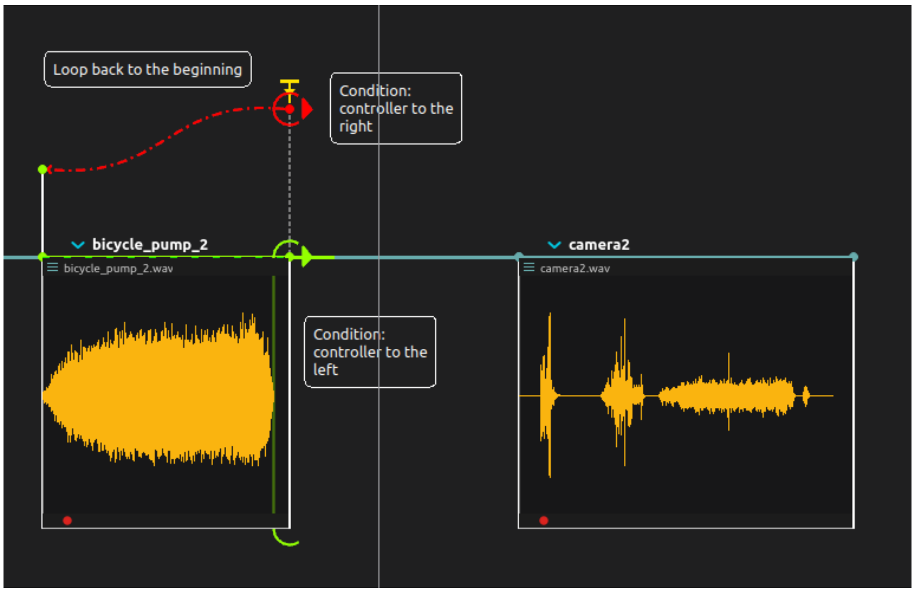
- Its timeline allows to score interaction and non-linear behaviors: that is, with the use of *triggers* and *conditions*, two elements of its domain-specific visual language, one can write a score with the following semantics: "Play a sound until a musician presses a button. Then, if a game-pad controller is pushed to the left, wait a second, play another sound and end the score. Otherwise, immediately loop back to the beginning of the score". This example score is given in Fig. 2.

FIGURE 2.1 – Example of an *ossia score* score which showcases three logic elements: the Trigger (yellow T with a downwards arrow) indicates that the score is waiting for an external interaction to proceed past that point. This interaction can be, for instance, receiving an OSC or MIDI message. The Condition (yellow C) indicates that what follows it will only execute if a given statement is true at the time the score's execution reaches that point. The dashed blue line indicates that this content's duration is not fixed (as it depends on when the following Trigger is going to be triggered). The full blue line indicates a fixed duration. Finally, the dashed-dotted line indicates an instantaneous "jump" in time, which can go backwards and thus easily implement looping behaviors.


Loop back to the beginning
Condition: controller to the right
bicycle_pump_2
bicycle_pump_2.wav
Condition: controller to the left
camera2
camera2.wav

Source: Jean-Michaël Celerier

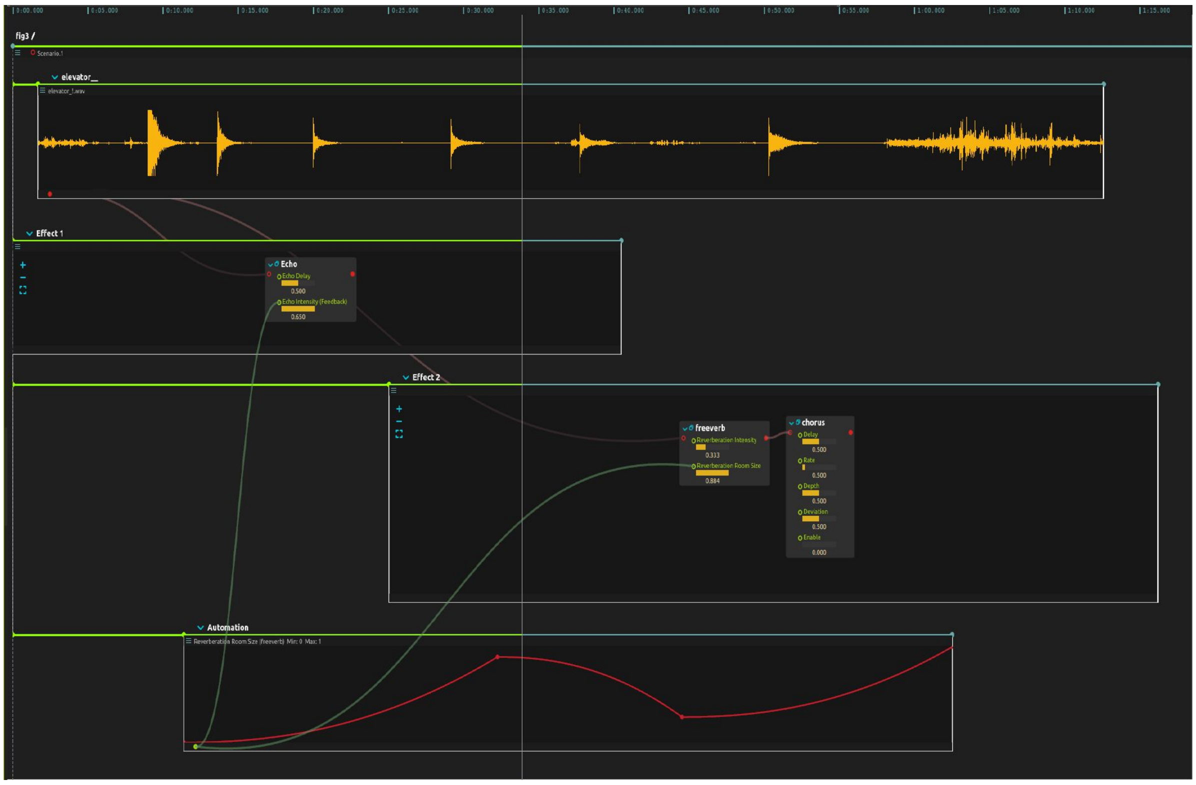
FIGURE 2.2 – Execution of such the score of Fig. 2.1. The branches not taken are in red, the branches taken are in red. The gray vertical bar indicates the absolute play time since the beginning: note that it is de-synchronized with the actual execution position for the elements of the score being played (in green): this is because each trigger can be triggered at a time unbeknownst to the composer, leading to an infinite set of executions instead of a single fixed one.


Loop back to the beginning
Condition: controller to the right
bicycle_pump_2
bicycle_pump_2.wav
Condition: controller to the left
camera2
camera2.wav

Source: Jean-Michaël Celerier

- Its data model allows to score various kind of values – audio, floating point, integers, character strings, lists, video, MIDI in a similar fashion to patchers and unlike common DAWs who generally only support audio, MIDI and floating-point controls as data types. The support for these data types stems from the origins of *ossia score* as a sequencer for OSC data.
- It merges the timeline and patcher model in a single generalized system allowing to patch inputs and outputs in a timely manner: an *ossia score* program's execution can be thought of as a sequence of processes automatically connected and disconnected from the underlying data-flow graph depending on whether they have been scored at a given point in time. That is, one can write a score where for the first half the sound input is routed to a first effect graph, and where for the second half the sound input is routed to another effect graph. Such a score is given in example in Fig. 3.

FIGURE 3 – A score with two successive effects being applied to a sound file. The first effect, an echo, will be applied from the beginning of the score to the 40 second mark ; the second effect, an effect chain consisting of a reverb and a chorus, will be applied from the 25 second mark to a bit after the song ended. A single automation is used to automate parameters pertaining to both the reverb and the echo effects.



Source: Jean-Michaël Celerier

Being built following a C++-plug-in-based design methodology, *ossia score* is easily extensible. It embeds multiple programming languages: Javascript, Faust, GLSL, C++, Exprtk, Bytebeat, which can all be readily used in scores: one can write a score which runs a first Javascript program regularly for the first minute of a score, then a sound, then a Faust synthesizer whose MIDI input is generated by another Javascript program.

From these, it follows that PureData is a natural candidate for being embedded in score, in a similar fashion to other programming languages.

2. 1. Processes in score

The main way computations, sound or MIDI playback, automation, ... are being performed within score is through the notion of process, which is analogous to Pd or Max objects ; the main difference being that every *ossia score* process has the built-in ability of doing timekeeping, having a duration, and more generally maintaining time-related information, which allows transformations to happen in an uniform way: scaling, speeding up or slowing down, increasing the duration without changing the relative time of the process's contents is possible. Otherwise, processes are simply nodes of a data-flow graph.

Every process has inputs and outputs, specified by ports (the small colored dots on the above figures). Ports can either be connected to an OSC address, MIDI control etc. or to another process by a cable. Communication between processes in score is synchronous and sample-accurate: every message is timestamped with the sample at which it was generated.

Inputs can optionally be matched with a specific kind of GUI control. For instance, sliders, check-boxes etc., which can of course be changed in real-time during execution of the score.

3. Embedding PureData in ossia score

We used libpd to embed the execution of PureData patches in *ossia score*. A *score* plug-in is used to create interoperability between *score*'s data structures and libpd's API. We try to match the data types as closely as possible: transport of floats, strings, bangs, boolean values, lists, MIDI, audio data are supported without data loss. A given PureData patch will be loaded into a *score* process, which will reflect the patch's declared inputs and outputs into *score*'s user interface.

The multi-instance feature of libpd is used to enable multiple concurrent patches. This has the consequence that externals need to be rebuilt (by setting the pre-processor define PDINSTANCE=1 when building the external), as the "normal" and the "multi-instance" builds of PureData are not binary-compatible. The libpd integration is configured in order to look for externals in the folder where the patch is; at some point in the future the software could give access to additional paths to look for externals in as part of its settings panel.

3. 1. User interface

The PureData integration behaves like the other *ossia score* processes integrating an external language or environment: it relies on drag-and-drop for easy manipulation. One can drop a **.pd** file into a score and the patch will be loaded at the point of the drop. If the user has Pure Data installed, a button will allow the user to open the patch for edition in the software; otherwise, only execution of the patch and modification of the specified parameters will be supported. The current implementation allows modification of the Pd patch to be taken into account by *ossia score* in real-time thanks to libpd's ability to open the Pd user interface.

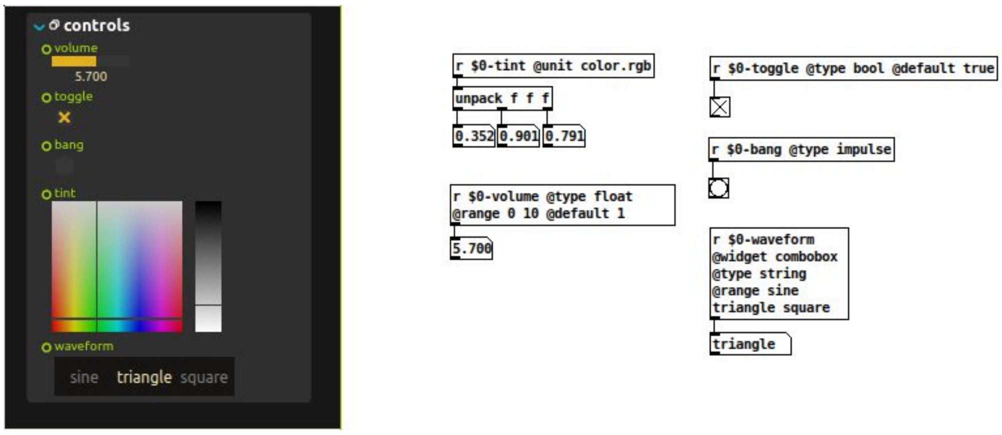
3. 2. Audio inputs and outputs

There is one major semantic difference between *ossia score* and Pd: in *ossia score*, audio cables and ports are multichannel by default. Conversely, in Pd, audio cables carry one channel. Thus, multiple audio inputs and outputs in the Pd patch are mapped to one audio port in *score*.

A small discrepancy occurs because of time handling in Pd being based on blocks of 64-frames: *ossia score*'s engine can (and sometimes will) ask to its plug-ins to process blocks of arbitrary length. This is because a given audio process can start, for instance, at sample 17 and run until sample 38, for a total of 21 samples. In that case, we zero-pad the data fed to PureData.

3. 3. Controls

FIGURE 4 – Controls generated in ossia score (left) from a Pd patch (right).



```
[r $0-tint @unit color.rgb]
[unpack f f f]
[0.352] [0.901] [0.791]

[r $0-volume @type float
@range 0 10 @default 1]
[5.700]

[r $0-toggle @type bool @default true]

[r $0-bang @type impulse]

[r $0-waveform
@widget combobox
@type string
@range sine
triangle square]
[triangle]
```

Source: Jean-Michaël Celerier

Score provides a user interface control library: 1D / 2D sliders, log sliders, buttons, color pickers... Those controls can have multiple properties. For instance, a min, max, init value, description. In order to be able to pick up this information from Pd, to create adequate controls in score, we leverage the [receive] and [send] Pd objects and the undocumented fact that they ignore anything past their first argument: we scan the patch for objects such as [receive $0-something <... arguments...>] and add them as controls to the score, by using additional metadata that can be appended to the object. In general, the same syntax than what is available for the [ossia] externals for Pd is used; note that this implementation does not use the [ossia] objects which as mentioned before are concerned with the specification of a network communication protocol over OSC. Here are a few examples; Fig. 4 provides a screenshot of the patch and matching process in *ossia score.*

- [r $0-volume @type float @range 0 10 @default 1]: creates a floating point slider ranging from 0 to 10 with a default value of 1 in score.
- [r $0-waveform @widget enum @range sine triangle square]: creates a multi-choice widget which allows to select one of the three values, sent as a symbol to Pd.
- [r $0-tint @unit color.rgb]: creates a color picker.
- [r $0-trigger @type impulse]: creates a button equivalent to a bang.
- [r $0-enabled @widget checkbox @default true]: creates a checkbox.

An extensive list of possible types, units, and other is available in the documentation of the software. The main benefit to this approach is that *ossia score*'s execution engine is aware of the ranges and units of its inputs and outputs ports, and will perform automatic conversions wherever applicable. For instance, if an OSC address specified as an RGBA color is used as input of a port specified as HSV, the color will automatically be converted from RGBA to HSV whenever a message is received. Further work will also look for the built-in visual objects of PureData which also have the ability of being annotated as receiving a symbol directly, and extract their metadata through parsing of the Pd data file.

Likewise, [send] objects will be mirrored in *score* as output ports.

3.4. Missing features

A few ossia features haven't yet found their way into the PureData integration. The two main features are the support for video processing & visuals, and the support for musical metrics. In the first case, *ossia score* leverages the Qt RHI library for handling real-time GPU visuals based on Vulkan, OpenGL, Metal and Direct3D; in particular it supports the Interactive Shader Format specification. It is as of yet unclear to the author how one may be able to insert Pd in the middle of a video processing chain handled by *score*, as it tries to offload as much computation as possible onto the GPU and does not do any CPU-based visual processing; everything goes through shaders for rendering.

In the second case, the main issue is that *score* maintains knowledge of musical metrics: tempo, measure. This allows the composers to quantize events on musical marks: for instance, if a trigger is triggered by an OSC message, it can optionally wait until the next bar, quaver etc. in order to keep synchronization with a rhythmic setup. This works for score processes even if they do not start on quantized times: an arpeggiator process which starts at frame 1357 (which would not fall on any reasonable musical mark, assuming for instance a sample rate of 44100 and 120bpm in 4/4) will still be in sync. The issue in the Pd synchronization is that we haven't found a way to indicate to Pd that its execution time is offset by some amount, for instance if one wants to use [metro] as a metronome within the patch: the first tick of the metronome will necessarily be at the first sample rendered by Pd which will not fall on the musical synchronization point. Thus, if one wants a Pd process to be synchronized, either the start time of the process in score must be for instance at the start of a bar, or the author of the score must add a special input to the patch that will provide bangs synchronized with the musical metrics of the score by connecting a suitable generator from score to this patch's input port.

4. Conclusion

We introduced a way to score PureData patches in a timeline, by integrating Pd as an *ossia score* process. This allows patch authors to easily use *score*'s time-centric features as a way to give a temporal specification to patches. For instance, by using automation, triggers, and more generally the entire temporal syntax of *score*. This is done in the hope of enabling authors of interactive and generative music to introduce time-based constructions more easily into their work, as well as in a way to access the vast amount of music software written in Pd from within *ossia score*. The integration covers most of the routine features of PureData, but lacks more advanced integration, most importantly in terms of video support.

It remains to be seen how various artists will adapt to this way of authoring scores, and in particular how being integrated in a scoring environment can change the way PureData patches are written. Anecdotal evidence from the author's usage so far tends towards small "do-one-thing" patches which are instanced at multiple points in the score: noise generators, filters, etc. Input of the Camomile users could also be worthwhile, as both systems lend themselves to similar use cases.

ACKNOWLEDGMENT

Part of this work has been funded by the SCRIME: Studio de Création et de Recherche en Informatique et Musiques Expérimentales (Creation and Research Studio in Computer Science and Experimental Music), Talence, France.

REFERENCES

AGOSTINI, Andrea and GHISI, Daniele. A max library for musical notation and computer-aided composition. *Computer Music Journal*, 2015, vol. 39, no 2, p. 11-27.

ALLOMBERT, Antoine, DESAINTE-CATHERINE, Myriam, and ASSAYAG, Gérard. De Boxes à Iscore: vers une écriture de l'interaction. *Actes des 13èmes Journées de l'Informatique Musicale*. 2008. p. 79-83.

ASSAYAG, Gérard, RUEDA, Camilo, LAURSON, Mikael, *et al.* Computer-assisted composition at IRCAM: From PatchWork to OpenMusic. *Computer music journal*, 1999, vol. 23, no 3, p. 59-72.

BEURIVÉ, Antoine. Un logiciel de composition musicale combinant un modèle spectral, des structures hiérarchiques et des contraintes. *Actes des Journées d'Informatique Musicale*, Bordeaux, France, 2000.

BRESSON, Jean and GIAVITTO, Jean-Louis. A reactive extension of the OpenMusic visual programming language. *Journal of Visual Languages & Computing*, 2014, vol. 25, no 4, p. 363-375.

BRINKMANN, Peter, KIRN, Peter, LAWLER, Richard, *et al.* Embedding Pure Data with libpd. In : *Proceedings of the Pure Data Convention*. Citeseer, 2011.

CELERIER, Jean-Michaël, BALTAZAR, Pascal, BOSSUT, Clément, *et al.* OSSIA: Towards a unified interface for scoring time and interaction. *Proceedings of the First International Conference on Technologies for Music Notation and Representation*. 2015.

CELERIER, Jean-Michael. *Authoring interactive media: a logical & temporal approach*. Doctoral thesis. 2018. Bordeaux, France.

GHISI, Daniele and AGOSTINI, Andrea. Extending bach: A family of libraries for real-time computer-assisted composition in max. *Journal of New Music Research*, 2017, vol. 46, no 1, p. 34-53.

GUILLOT, Pierre. Camomile: Creating audio plugins with Pure Data. *Proceedings of the 2018 Linux Audio Conference*. 2018, June.

KELLY, Edward. Gemnotes: a realtime music notation system for Pure Data. *Proceedings of the 4th international conference of Pure Data*. 2011.

PUCKETTE, Miller. A divide between "compositional" and "performative" aspects of Pd. *Proceedings of the First International Pd Convention*. 2004.

PUCKETTE, Miller. Using Pd as a score language. *Proceedings of the 2002 International Computer Music Conference.* 2002.

ABOUT THE AUTHOR

Jean-Michaël Celerier, born in France in 1992, is a freelance researcher, interested in art, code, computer music and interactive show control. He studied software engineering, computer science & multimedia technologies at Bordeaux, and obtained his doctorate on the topic of authoring temporal media in 2018. He develops and maintains a range of free & open-source software used for creative coding, digital and intermedia art, which he leverages in various installations and works. He enjoys organizing events centered on programming and media art and teaches all sorts of creative coding systems to both computer science and graphics design students. ORCID: https://orcid.org/0000-0003-1253-298X. E-mail: jeanmichael.celerier@gmail.com