Designing a Flexible Workflow for Complex Real-Time Interactive Performances

Esteban Gómez
Departamento de Música y Sonología
Universidad de Chile
Compañía 1264, Santiago, Chile
esteban.gomez@ug.uchile.cl

Javier Jaimovich
Departamento de Música y Sonología
Universidad de Chile
Compañía 1264, Santiago, Chile
javier.jaimovich@uchile.cl

ABSTRACT

This paper presents the design of a Max/MSP flexible workflow framework built for complex real-time interactive performances. This system was developed for *Emovere*, an interdisciplinary piece for dance, biosignals, sound and visuals, yet it was conceived to accommodate interactive performances of different nature and of heterogeneous technical requirements, which we believe to represent a common underlying structure among these.

The work presented in this document proposes a framework that takes care of the signal input/output stages, as well as storing and recalling presets and scenes, thus allowing the user to focus on the programming of interaction models and sound synthesis or sound processing. Results are presented with *Emovere* as an example case, discussing the advantages and further challenges that this framework offers for other performance scenarios.

Author Keywords

Interactive performances, Max/MSP, Emovere, OSC

ACM Classification

C.3 [Special-Purpose and Application-Based Systems] Real-time and embedded systems D.2.6 [Programming Environments] Interactive environments H.5.5 [Information Interfaces and Presentation] Sound and Music Computing.

1. INTRODUCTION

NIME performances utilize a great variety of resources in order to create interactive experiences that comprise sound, music, lights, projections and control of mechanical structures, among other components. Each interactive live performance manages these materials in diverse ways depending on its objectives and artistic goals, combined with the artist's experience with software and hardware tools. Examples cover a wide spectrum, from performances with robotic musical instruments [3] to sleeping performers [4]. A popular set-up includes Max/MSP [1] in combination with communication protocols such as MIDI and OSC [8]. In this paper, we present a software framework built on these tools, which we believe to represent an underlying structure in common among performances of different nature. This system can be used as a base to develop a performance set-up compatible with different live scenarios, regardless of the resources needed in a particular performance.

The framework presented in this paper was developed for *Emovere*, a performance for contemporary dance, sound, projections and lights. This performance was the result of an interdisciplinary creative research project lasting over 18 months. During this time, a team that involved dancers, composers, sound designers and visual artists, among others, developed a methodology based on a lab setting. The first phase of the project incorporated the development of different creative materials that included several sound objects (SOs), interaction design models, software tools and choreographic structures that formed the building blocks of the approximately one hour long interactive piece.[1]

Emovere portrays different emotions through a dance choreography divided in three acts. In this performance, each dancer has four sensors attached to his/her body. The physiological signals from the dancers drive an interactive performance constructed around the theme of emotion. *Emovere* measures electrocardiography (ECG) and electromyography (EMG) of four dancers, which are processed and then mapped to a series of sound objects (SOs) in order for the performers to be constantly modulating and shaping the sound and visual environment of the piece. This creates a dynamic and unpredictable soundscape that is mediated by the corporal state of the performers, which in turn is affected by their volitional movements and self-induced emotion.

This document will focus on the Max/MSP-based sound framework developed for addressing all the needs of the sound composition during rehearsals and live performances. This framework is oriented to provide a flexible workflow for several performances beyond *Emovere*, thus, saving programming time and making real-time parameter control easier and more convenient than starting always from scratch.

Even though this framework is currently designed to manage sound environments, it is still under development and more implementations will be added in the future to provide a similar workflow when managing other type of resources and outputs. For the next versions, and after appropriate stability tests, we intend to make the framework files available for download.

1.1 Motivation

Max/MSP has been widely known around the world for its short an attractive phrases like "Modular by design" [1] and "No matter what you have on your table, you can probably make it talk to Max" [1] because it proposes a more "organic and immediate" [1] programming system compared to common written languages such as C++. This appealing idea can lead to endless possibilities in a relatively reasonable amount of time, but as soon as a patcher (a Max/MSP program) becomes more complex, it gets harder to be understood by a third party because every programmer can use his/her own organization within the Max/MSP environment. Previous efforts such as Jamoma [5, 6] have been developed in order to standardize how a complex patch can be planned and put together, in order to create a common architecture to make third party understanding and modifications easier than they currently are.

Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). Copyright remains with the author(s).

NIME'16, July 11-15, 2016, Griffith University, Brisbane, Australia.

[1] http://www.emovere.cl

However, approaching a complex performance set-up often requires building your own tools in order to accommodate a particular set of inputs/outputs, different data formats, heterogeneous signals and variant bandwidths. In the case of *Emovere*, several complex tasks needed to be performed as quickly as possible to optimize rehearsal times as well as to consolidate certain acts of the performance and recall them rapidly and exactly as they were planned.

Under this paradigm, many questions arise regarding information management storing preset files and recalling efficiently as many copies as needed of a multipart patch (a patch with other nested patches) used to process the data and output of a sound under a specific method.

Those questions opened a discussion about how to program the system itself such as if a SQLite database was convenient enough for all the information storage needed, or if other structures were more convenient regarding the information volume and speed needed to operate successfully in a real time basis, such as the dict Max/MSP object based on the .json format, or the .xml files associated with the Max/MSP pattr object family, or a combination of them.

Many Digital Audio Workstations (DAWs) such as Pro Tools, Logic or Cubase use a design capable of accomplishing many of this tasks successfully, but their environment is not suitable to perform complex tasks with sensors and data streams in real time, since their focus is audio edition in depth.

From this starting point, the framework behind *Emovere* tries to gather the best of both worlds to create an environment suitable for complex real-time interactive performances based on a DAW architecture and at the same time, create an organized and intuitive way of working interaction design, avoiding as many external dependencies as possible to prevent the addition of new learning steps before creating patches.

Consequently, we discarded the idea of using another framework or third party design rules (such as SpatDIF[2]) as a starting point. Besides, our goal was not to develop a comprehensive interface for every possible patch inside Max/MSP as Jamoma, but to develop a suitable interface to deal with sensor data as the main input stage and process them efficiently in order to produce a sound output. For this purpose, we needed (among others) a tailored virtual mixer compliant with an output stage for our needs.

2. METHODOLOGY

2.1 Sound Objects: The Cornerstone for Modular Workflow

Observing several DAW systems, we thought that the overall model behind them has become very intuitive, because the principles emulated by these software are the same principles behind most analog mixing consoles and they are a common place for sound engineers, musicians and sound artists. This model can be understood as an input-processing-output (IPO) model, where the input can be a MIDI signal from a controller or an audio signal, the processing block can be the internal routing and plugin processing units and the output will be an audio signal or a MIDI signal as well.

We needed to expand the input concept to receive multiple inputs of different kinds, such as MIDI and OSC simultaneously as a stream with fixed or variable frequency. The output could be a single signal or several signals of different types. For example, four audio signals and one OSC message. In the case of *Emovere,* it was initially conceived as a quadraphonic performance that requires four audio outputs for each SO, but in some cases this includes OSC messages for the video projections computer and other control signals.

[2] http://www.spatdif.org/

The processing block needed to be designed to allow multiple possibilities, in order to process different interactive mappings and configurations. Besides adding typical processing features, such as reverbs or compressors, we needed to process OSC and MIDI data streams and/or signals to create certain interactions. For example, sometimes data can trigger the reproduction of recorded files under specific conditions, or it triggers the recording of a microphone signal based on a specific physiological pattern of the dancers.

At this point is where the Sound Object (SO) concept emerges as an encapsulation of all the complex processing we could imagine, becoming the cornerstone of the framework. The whole model can be comprehended now as an expanded IPO architecture where the SO is the complex processing unit and at the same time it is fully opened to the programmer (see Figure 1). Further information about how SOs interact with each other will be described in the next sections.

2.2 *Emovere* Signal Flow within the System

As it was stated previously, four dancers performed in *Emovere* with four sensors each. The data coming from the sixteen sensors was processed within a laptop exclusively dedicated to input processing, monitoring and routing of the incoming data, since the amount of information in every second was substantial and needed to be distributed to the visual resources unit as well. This laptop was called "node", but in smaller set-ups the data processing stage can be integrated to the framework as a SO or part of one, allowing performances to be controlled entirely using only one laptop. In the specific case of *Emovere*, if the data processing performed by the node was added to the framework, less CPU would have been available for the SO and output signal generation.

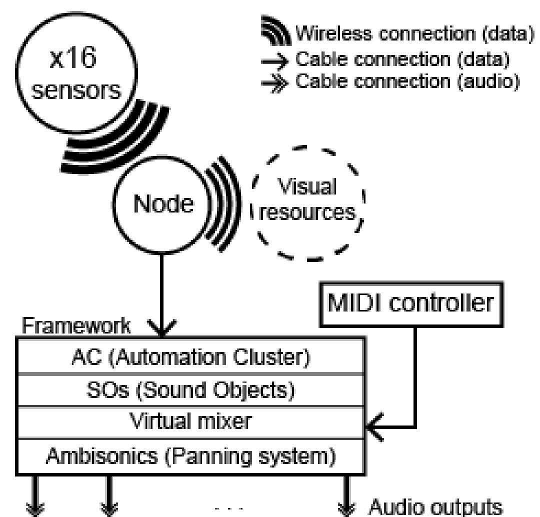

Figure 1. Signal flow diagram.

Once the information arrives from the processing stage (which is the framework itself) through OSC protocol, it is distributed to every SO after being routed through the Automation Cluster (AC), which is the unit of the framework included as part of every SO that allows them to discern whether OSC messages are for them or not.

After this, the SO receives the information and generates output in real time according to the patch algorithms. The output is transmitted to one or several channel strips inside a virtual dynamic mixer depending on the SO configuration. Every SO has its own configuration sheet inside a Max/MSP dict object, thus, the framework creates instantly the necessary amount of channel strips to control and visualize the SO outputs. Aside from channel strips for every individual output

of a SO, there is also a master channel strip that allows modifying all SO outputs at once, and this was especially useful to create crossfades manually between different acts of the performance where different SOs were being mixed.

After the virtual mixing console, there is a panning stage based on Ambisonics [7] that is able to pan all the signals between every number of desired speakers and switch speaker configuration rapidly, as well as having the option to store them in a preset to be recalled later when moving from one venue to another.

Besides the normal signal flow, there are data flows for specific tasks, such as controlling the mixer through a MIDI controller or creating/destroying SOs remotely through OSC. Every incoming signal has a unique path to separate high priority task from low priority tasks.

2.2.1 Input Tools and Data Processing

As already mentioned, in order to optimize resources and separate tasks, our proposal was to separate the sensor signal processing and feature extraction from the sound and video generation machines. For *Emovere*, four dancers were connected with physiological signals, each one being streamed at 250 [Hz] via Bluetooth to the node computer.

This computer (the node), running Max/MSP, handled the signal pre-processing stage, which includes artifact detection and feature extraction for electromyogram (EMG) and electrocardiogram (ECG) signals. Once the significant features have been extracted, these were connected to different mapping strategies, depending on the interaction modes that were constructed for each section of the piece. For example, an ECG signal would first be pre-processed to remove DC components and heartbeats within a specific range, to then extract features such as heart rate and heart rate variability [2]. These features can then be mapped to different SO via OSC existing in the framework computer.

2.2.2 Sound Objects Management Architecture

Every SO inside the system should have a unique name and can be created or destroyed. Beneath the system there is a dict object containing all the information of every SO, such as inputs and outputs and the route to the folder where the SO's patches are located. This way, SOs can be traced and the organization of the virtual mixer can be recalculated using this information every time a new SO is created or destroyed, as well as the necessary calculations to keep the Ambisonics panning system organized.

A SO can be created from the terminal or the GUI of the system. There are commands that can be written in the terminal and they will trigger the necessary functions included in the core scripts of the framework using a different path from the one used to receive OSC, and since the core scripts are made of JavaScript, tasks like creating an object have a low priority.

The advantage of the terminal commands is that they can be recalled remotely because the terminal has its own dedicated port (see Figure 2).

2.2.3 Bidirectional Framework-Object Communication

When a new object is incorporated to the framework, it has to follow certain rules to work properly and to be communicated with the framework. However, the rules are kept as simple as possible.

2.2.3.1 Folder structure

Every SO should have its folder inside the "obj" folder and it should have a subfolder called "presets" where all the preset files will be stored automatically. The folder's name can be whatever the user wants, but the main patcher used as abstraction inside the framework should be called with the same name (see Figure 3).

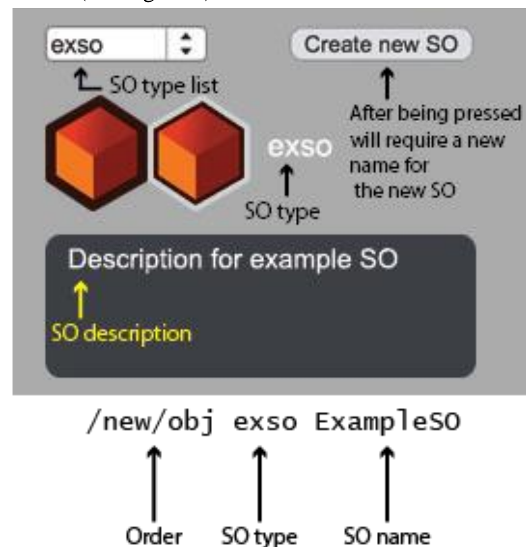

Figure 2. Creating a new SO of the "exso" type named ExampleSO. The "Create new SO" button will prompt a window requiring a name for the new "exso" type SO. The same result could be obtained writing the command below using the terminal.

Figure 3. SO folder distribution diagram where "exso" stands for "Example SO" which is a SO type used as example. The "exso.maxpat" patch could be a complex interaction such as a granulator.

2.2.3.2 Local variables

Some pv objects[3] are incorporated in order to establish communication between the framework and the SO, the LOC%id% will contain the unique ID of a SO inside the system and the LOC%obj_name% will contain the name given to the SO by the user. This will allow the core scripts to locate SOs using the this.patcher JavaScript method (see Figure 4).

2.2.3.3 Shortcuts and AC

Every SO has its shortcut subpatcher and AC subpatcher. The shortcut subpatcher contains the shortcuts for every object attached to an active object to avoid triggering shortcuts when the SO window is not on focus, this helps to prevent shortcut overlapping. The AC subpatcher controls the incoming OSC messages. Every AC can be detached individually if desired to optimize CPU and to keep the shortest path possible between

[3] pv objects can store and share a variable within a patch hierarchy, thus, being invisible for other top-level patchers.

the incoming OSC message and the object inside the SO that receives that OSC message (see Figure 5 and Figure 6).

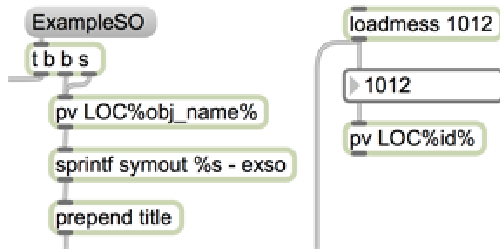

Figure 4. Every SO has a unique ID assigned when it is created that is stored in the pv LOC%id% object. The name of the SO is stored inside the pv LOC%obj_name% and will be used for many purposes such as printing the name of the SO as the window title. Every variable of a SO, such as name and ID can be accessed by the core scripts using its scripting name. Inside the SO, those variables can be accessed locally by the pv objects.

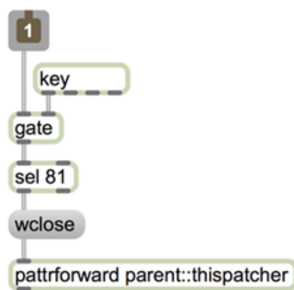

Figure 5. The shortcut structure inside the shortcuts subpatcher. Inlet 1 is connected with an active object to prevent shortcuts overlapping. If the window is on focus and a combination such as 81 (Shift+Q) is detected, the example shortcut will close the parent patcher's window calling a thispatcher object with the same scripting name.

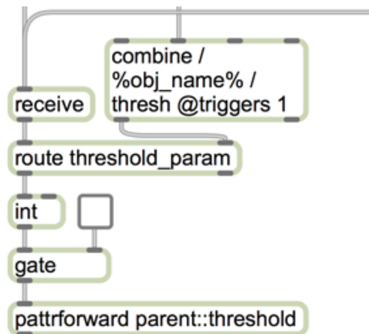

Figure 6. An example of an automation cluster (AC). With the information provided by local variables, the combine object will set up the adequate OSC message to be received and decoded. The receive object has no argument, this way it will be dynamic and can be closed using the "set" message without second argument. After the route object, the int object will help to guarantee the type of the decoded variables and the path can be closed using the toggle object. Finally, the pattrforward object will deliver the variable to the target object within the parent patcher.

2.2.3.4 Preset manager

The preset manager will perform all the tasks related with saving and loading presets. This subpatcher should be slightly modified before being pasted inside a new SO because of the folder routes of the new SO. Preset can be loaded remotely using terminal command lines (see Figure 7).

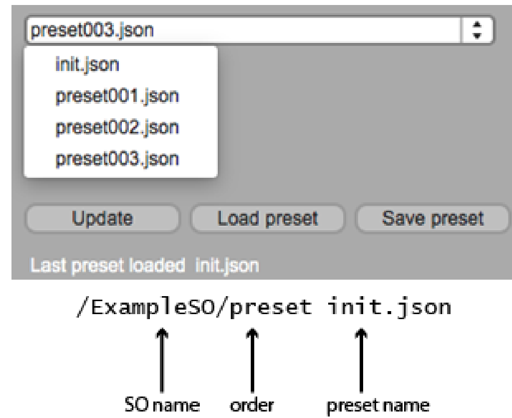

Figure 7. The preset manager can load and save presets and update the list rescanning the preset folder when needed. The last loaded preset will be shown at the bottom. A preset can be loaded using the "Load preset" button or remotely using the command line described. Any new SO will start with the init.json preset after being created.

3. RESULTS ANALYSIS

3.1 CPU and DSP

The framework ran over a MacBook Pro and Max 6. The same computer was used in every performance and lab session. Unnecessary services such as Bluetooth and Wi-Fi were always turned off to keep the minimum necessary to reinforce stability and operate comfortably as well. Table 1 describes hardware and software specifications used during the performance and relevant settings of Max/MSP.

Table 1. Computer specifications and settings

Computer specifications	
Model	MacBook Pro (13-inch, MID 2012)
Processor	2.9 GHz Intel Core i7
Memory	8 GB 1600 MHz DDR3
Graphics	Intel HD Graphics 4000
Operating system	OS X Yosemite 10.10.2
Audio interface	MOTU 828x
Framework specifications	
Size without audio files	13.3 Megabytes
Full size	1.23 Gigabytes
Externals	ICST_ambisonics_2_3_1
Max/MSP configuration	
Sampling rate	44100 Hz
I/O Vector Size	256 samples
Signal Vector Size	256 samples
Scheduler in Overdrive	Enabled
Scheduler in Audio Interrupt	Enabled
Parallel processing	Enabled

Data was transmitted from the sensors at 250 [Hz], this way, 4000 int numbers were received each second in the node unit. However, the same rate was not necessarily reflected in the input of the MacBook Pro running the framework, because the node unit processed data before reaching the framework and depending on the case, float numbers could be received instead, or specific messages at a different frequency, but always using the OSC protocol.

Each act of *Emovere* used different amount of resources as shown in Table 2. *Emovere* featured the design of nine different types of SO combined under artistic criteria in every act.

On the other hand, although the DSP CPU monitor was always a concern during performances, evaluation was assessed only anecdotally by visually monitoring the CPU usage. Further stress tests are pending in order to assess the framework's capabilities. The design premise was that the DSP CPU monitor should not exceed the 60% during a performance under normal conditions, including muting the DSP of a SO once it was considered idle. During the development of the framework we experienced that over 60% or 65% of CPU usage, and with the chosen configuration, screen might freeze for short periods of time, thus, making visual monitoring gradually harder. Finally, this criteria was stated as a fine limit for safe operation.

Table 2. Number of audio channels and SOs used during acts of *Emovere*.

Act	Audio channels (mono outputs)	SOs used
Act #1	19	8
Act #2	26	16
Act #3	13	13
Total	58	37

4. DISCUSSION

4.1 Real-Time Effectiveness

The framework was used during 20 live performances and seen by over 1200 people with no issues. Only minor instabilities were detected, such as the DSP engine restarting unexpectedly during rehearsals. We were not able to reproduce this error, and it never occurred after restarting Max/MSP.

The framework was developed initially over the latest versions of Max (7.03 at the time) but eventually we noticed that the improvements made from Max 6 to Max 7 were accompanied by an increased CPU usage, specifically for a poly~-based patches. Because we were running several SOs with poly~ during the performance, each with a substantial amount of voices, we decided to go back to Max 6 in order to improve stability. Correspondence with Cycling74 informed us later that the increased CPU usage seems to be related to the new dirac feature included in Max 7. Aside from decreasing CPU usage, the overall GUI was affected because Max 7 GUI includes new additions that are not entirely retro-compatible with Max 6. In future versions of the framework the GUI will be kept as independent as possible from the GUI provided by Max.

Future improvements are focused on extending the capabilities of the framework for managing different kinds of outputs other than sound, such as light systems or reducing the two laptops system to one in the case of smaller scale performances. Further implementations were tested but not fully implemented yet, such as a virtual mixer and SO control over wireless networks from a portable device such as tablets or mobiles.

4.2 Flexibility in Different Venues

The speaker manager included in the framework and the preset system was sufficient to change the whole configuration overnight when changing venues. The performance was shown in different venues with totally different speaker settings and every time the system proved to be adjusted successfully. Obviously, some minor modification and skimming through the whole performance is needed to fine adjust the overall sound and correct the panning to get a proper spatialization as you would normally do with a band or a play.

5. CONCLUSIONS

As the input-processing-output (IPO) structure is common among very complex systems, taking advantage of this architecture leads to an intuitive design with a simple workflow. Although the framework is at an early stage of development, the results obtained reinforce its suitability for live interactive performances.

The use of the sound object (SO) as a processing block, allowed the expected independency between the framework and the interactions inside every SO. Even though the dependency between SOs and the framework can be improved, the current degree of dependency was not an obstacle for programming several SOs to deliver a variety of possibilities within a single performance.

Additional testing should be done to establish more accurate conclusions regarding the minimum requirements of the framework over different operating systems, as well as its maximum capabilities. Nonetheless, the framework developed was able to successfully control and manage a complex interactive performance such as *Emovere*, proving to be a flexible and reliable system.

Further analysis on poly~-based patches should be done in order to allow users to include the improvements incorporated in recent versions of Max 7 for the development of future versions of the framework, as well as quantifying the extra CPU usage due to dirac features. This will also improve the recommendations we could elaborate regarding the maximum framework capabilities.

6. REFERENCES

[1] Cycling '74: *https://cycling74.com/*. Accessed: 2016-01-27.

[2] Jaimovich, J. and Knapp, R.B. 2015. Creating Biosignal Algorithms for Musical Applications from an Extensive Physiological Database. *Proceedings of the 2015 Conference on New Interfaces for Musical Expression (NIME 2015)* (Baton Rouge, LA, Jun. 2015).

[3] Mathews, P., Morris, N., Murphy, J., Kapur, A. and Carnegie, D.A. Tangle: a Flexible Framework for Performance With Advanced Robotic Musical Instruments.

[4] Ouzounian, G., Knapp, R.B., Lyon, E. and DuBois, R.L. 2012. To be inside someone else's dream: On Music for Sleeping & Waking Minds. *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME''12)* (2012).

[5] Place, T. and Lossius, T. 2006. Jamoma: A modular standard for structuring patches in Max. *Proceedings of the International Computer Music Conference* (2006), 143–146.

[6] Place, T., Lossius, T., Jensenius, A.R. and Peters, N. 2008. Flexible control of composite parameters in Max/MSP. (2008).

[7] Wakefield, G. 2006. Third-order Ambisonic extensions for Max/MSP with musical applications. *Proceedings of the 2006 ICMC*. (2006).

[8] Wright, M. 2005. Open Sound Control: An Enabling Technology for Musical Networking. *Org. Sound*. 10, 3 (Dec. 2005), 193–200.