
Skini on Node.JS

Instructions for use and implementation

B. Petit, 11/2022

Table of contents

1	Introduction	5
1.1	Composition process.....	5
1.2	Patterns.....	5
1.3	Orchestration.....	5
1.4	Musical Pieces with a DAW	6
1.5	Pieces with musicians (to be ported to node.js).....	6
2	The configuration	7
2.1	Installation	7
2.2	Installation of Processing for the osc/MIDI gateway	7
2.3	Configuration	7
2.3.1	IP configuration and directory	7
2.3.2	Access to pieces and pattern descriptors	8
2.4	MIDI configuration.....	8
2.5	Room configuration	10
2.5.1	How to receive MIDI commands	10
2.5.2	Reaction mode	10
2.5.3	Sound files for clients and customers	10
2.5.4	Synchronization.....	11
2.5.5	Groups of patterns	11
2.5.6	Musicians (Not in place).....	12
2.6	Pattern configuration.....	12
2.7	Configuration of the orchestration display on a large screen	14
3	Launch Skini.....	15
4	Using the controller	17
5	Using the MIDI Configurator	17
6	Using the SKINI client	18
7	Using the simulator.....	19
8	Orchestration programming.....	21
8.1	The Skini graphical interface	21
8.2	Orchestration blocks	22
8.3	Hiphop blocks	23
8.4	The programming of time in a room.....	23
▪	Use of the tick signal	23
8.4.1	Use of the pulse signal.....	24

8.5	Tempo	24
8.6	Reservoirs and groups of patterns.....	25
8.6.1	Groups of patterns	25
8.6.2	Actions related to groups.....	25
8.6.3	Creation of tanks.....	26
8.6.4	Actions on the reservoirs	27
8.7	Instrument queues	28
8.7.1	Emptying of queues	28
8.7.2	Pause and test queues	29
8.7.3	Queueing a specific pattern	29
8.8	Patterns.....	29
8.9	Display of the orchestration.....	30
8.10	Skini Game.....	30
8.11	MIDI Command and Control Change	31
8.12	Ableton Live specific	32
8.13	OSC commands for Raspberry.....	32
8.14	Advanced blocks	32
8.15	Control Interface Z	33
8.16	Programming of stinger transitions	35
8.16.1	Case of the reaction to the execution.....	35
8.16.2	Case of the pattern set return from the DAW	36
8.17	Priority in queues	37
8.17.1	Example of programming a duel of solos (review for Node).....	38
8.17.2	Example of transposition in loop.....	40
8.18	Some subtleties of synchronous reactive programming.....	41
8.18.1	Immediate reaction	41
8.18.2	Double emission of the same signal	41
8.19	Spatialization by tiling with the Pré device.....	42
9	With musicians (not in place with Node.js).....	45
10	OSC interface	46
10.1	Principle	46
10.2	Since the orchestration	46
10.3	Implementation with orchestration.....	46
11	OSC with Bitwig Studio.....	47
12	OSC with Max4Live.....	48
12.1	Conversion track	48

12.2	Midi routing track.....	48
12.3	Feedback track.....	49
12.4	M4L patches	50
12.5	Conclusion OSC Skini with Ableton.....	51
13	Pattern control on Raspberry	52
14	Annexes	52
14.1	Skini with Ableton Live	52
14.2	Skini with Bitwig Studio	53
14.3	Examples of orchestrations	53
14.4	Organization of the file system	53
14.5	Score recording in Finale	54
14.5.1	From an Ableton recording.....	54
12.a.i.	Directly from Skini.....	55
14.6	Switch from Finale to Microsoft Word or PowerPoint	55
o	Receive MIDI clip launch info from Ableton.....	55
14.7	Recording of live patterns in the distributed sequencer	57
15	An example of a piece: Opus5.....	58

1 INTRODUCTION

Skini is a collaborative and generative music composition platform. This document addresses the technical points related to the implementation of Skini on Node.JS and the creation of orchestration. It corresponds to the functioning of Skini in its June 2022 version. To understand the functioning of Skini, it is advisable to refer to the document "*Time and Duration: from synchronous programming to musical composition*" or to the various articles published on this platform (Programming journal 2020, ICMC 2021, NIME 2019).

This document also includes in appendix some procedures for using Finale and Ableton Live with Skini.

1.1 COMPOSITION PROCESS

Skini was designed to compose music that will be performed in interaction with an audience or produced automatically by random processes. The solution includes a web server that integrates orchestration modules written using a programming tool that is an abstraction layer on top of the HipHop.js language. The composition method is based on two basic concepts: patterns and orchestration. These are described in the thesis "Time and duration: from synchronous reactive programming to music composition". We only discuss here the practical dimension of the tool, i.e., its implementation.

1.2 PATTERNS

The composer can create the patterns without any constraints from Skini. They will be seen as elements activated by a MIDI command and having defined durations in number of pulses. The patterns are made available to the audience as groups¹. There is no constraint on the size of the pattern groups.

1.3 ORCHESTRATION

The composer will define how the groups are made available to the audience through orchestration. The orchestration allows to *activate* and *deactivate* groups of patterns. Among the information that allows it to evolve we have, listening to the group selections by the audience, the duration measurement, listening to MIDI or OSC information.

Orchestration can be seen as a "super sequencer" bringing *queue-based* interaction functions and complex automation to commercial DAWs. The orchestration can also control the queuing of patterns without an audience. It is possible to issue MIDI, note or "control changes" commands directly from the orchestration.

¹ If the composer wants to make a particular pattern available to the audience or the simulator, he can create a group with a single element for example.

The orchestration is written in HipHop.js language in a graphical form. A basic knowledge of synchronous programming is necessary to start designing a first orchestration. With a good mastery of HipHops.js it will be possible to produce rich and inconceivable pieces otherwise.

1.4 MUSICAL PIECES WITH A DAW

In this document, we will only discuss the case of Ableton Live and Bitwig Studio. The use of another DAW will be based on the same principles as those proposed for one or the other solution.

The composer must create patterns (clips). To each of these patterns the composer will associate a Skini note. This Skini note will be converted and sent to the DAW from commands issued by the audience or from a random process.

The composer can create as many patterns as he wants while keeping in mind that these patterns will be organized in groups and that these groups will be made available to the audience or the random process. For a collaborative piece, the dimensions of the groups must therefore be compatible with the possible display on an interface for the audience. The composer must also find a good balance between short patterns that will dynamize the interaction and long patterns that will stabilize the musical discourse.

In the case of an interaction with the audience, for each pattern the composer will have to create a *sound file*, mp3 or wav, whose name is associated to the pattern in the pattern configuration file. By default, it is an mp3 file that will be downloaded by the audience for listening before a selection.

Once the patterns have been created and the associated MIDI commands, all that remains to be done is to create the configuration files as described in the configurations chapter, and then move on to orchestration.

1.5 PIECES WITH MUSICIANS (TO BE PORTED TO NODE.JS)

Skini can be used to communicate with musicians, with Raspberry Pi having the adapted client, with or without synthesizers. The process is identical to that of a DAW. The activated patterns no longer consist of issuing a MIDI command, but of displaying on clients dedicated to musicians scores previously deposited in a subdirectory of the `"/image"` directory. This subdirectory is configured in the configuration file of the piece. These files are in jpg format. The names of the jpg files are the same as those of the sounds associated with the patterns.

2 THE CONFIGURATION

The directories are referenced to the main directory where Skini is installed.

2.1 INSTALLATION

All you must do is install Node.js, which is a widely used solution, and copy the Skini files into a directory. If packages are missing when Skini is launched, Node.js will report it with error messages. You just have to install the packages with npm.

2.2 INSTALLATION OF PROCESSING FOR THE OSC/MIDI GATEWAY

The OSC/MIDI gateway is used when the DAW only understands MIDI commands and the Skini server is not on the same machine as the DAW. If the Node.js server and the DAW are on the same computer there is no need to install the OSC/MIDI gateway and therefore this step is not necessary. You can use Skini by accessing the MIDI port of your machine directly.

So, if the communication with the DAW is done in MIDI and you use a computer which hosts the Node.js server and another computer which hosts the DAW, you will have to install Processing (www.processing.org) the two computers will be able to communicate in OSC and you will need the gateway. In Processing you will have to install the libraries oscP5, TheMidiBus, WebSockets (menu: add a tool -> libraries).

2.3 CONFIGURATION

There are two configuration files to update before launching Skini. One concerns the network configuration, the other the MIDI configuration.

2.3.1 IP configuration and directory

It is done with the file ". /serveur/ipConfig.json".

Example of ipConfig.json:

```
1. {
2.     "remoteIPAddressImage": "192.168.82.96",
3.     "remoteIPAddressSound": "localhost",
4.     "remoteIPAddressLumiere": "192.168.82.96",
5.     "remoteIPAddressGame": "192.168.82.96",
6.     "serverIPAddress": "localhost",
7.     "webserverPort": 8080,
8.     "websocketServeurPort": 8383,
9.     "InPortOSCMIDIfromDAW": 13000,
10.    "OutPortOSCMIDItoDAW": 12000,
11.    "distribSequencerPort": 8888,
12.    "outportProcessing": 10000,
13.    "outportLumiere": 7700,
14.    "inportLumiere": 9000,
15.    "sessionPath": "./pieces/",
16.    "piecePath" : "./pieces/"
17. }
```

serverIPAddress : server address.

remoteIPAddressLumiere : for use with QLC for an OSC dialogue

remoteIPAddressDAW : IP of the Processing server for MIDI commands to Ableton or another DAW.

remoteIPAddressSound : IP of the Processing server for MIDI commands to REAPER, this is for the GOLEM show.

remoteIPAddressImage : IP of the Processing server for a large screen visualization.

For standard use, you just need to update remoteIPAddressAbleton and serverIPAddress.

The change of port for Websockets "websocketServeurPort ", is done with the powershell script ***changePortSkini.ps1***. (Otherwise, it is necessary to pass again browserfity on the clients).

2.3.2 Access to pieces and pattern descriptors

The parameters *sessionPath* and *piecePath* are part of the system settings and are therefore on the same level as the network. *sessionPath* defines the directory of the piece settings and *piecePath* those of the Blockly orchestration. It is not possible to access the full path from a browser, that's why these parameters are necessary. These paths must be in a directory immediately below the *nodeskini* root (constraint linked to the access to midiConfig.json).

2.4 MIDI CONFIGURATION

The configuration of the MIDI ports is done using the `./server/midiConfig.json` file. This file defines the MIDI buses according to the computer configuration. This file is used by Skini and Processing.

Here is an example using LoopMIDI on Windows:

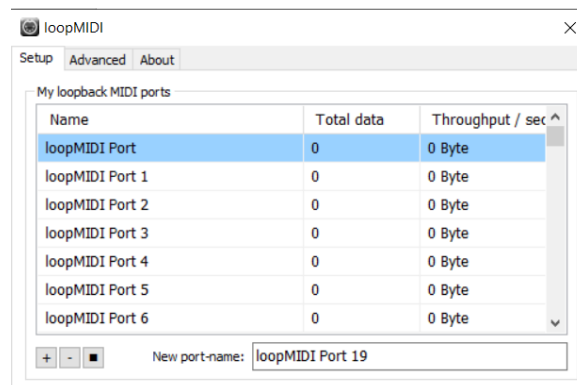


Figure 1: The LoopMIDI virtual cable

```
[
  {
    "type": "OUT",
    "spec": "clipToDAW",
```



```

    "name": "loopMIDI Port 6",
    "comment": "Bus for launching the clips in the DAW"
  },
  {
    "type": "IN",
    "spec": "syncFromDAW",
    "name": "loopMIDI Port 9",
    "comment": "for sync message from DAW"
  },
  {
    "type": "IN",
    "spec": "clipFromDAW",
    "name": "loopMIDI Port 12",
    "comment": "for clip activation message from DAW"
  },
  {
    "type": "IN",
    "spec": "controler",
    "name": "nanoKEY2",
    "comment": "to test a MIDI controller"
  }
]

```

The "type" field defines whether it is an IN or OUT port.

The "spec" field defines the usage with:

- The "clipToDaw" port allows the DAW to receive MIDI commands from Skini.
- "syncFromDAW" defines the port that will receive MIDI synchronization from the DAW.
- "clipFromDAW" defines the port through which the DAW sends the clip start messages.
- "Control" corresponds to a port to which a MIDI controller is associated (keyboard, PAD...). is not necessary for the basic operation of Skini.

The "name" field contains the name of the MIDI port on the computer. Here, these are ports on the LoopMIDI interface.

The "how" field allows you to comment on the use of the port.

Here is an example of the MIDI port configuration corresponding to the above example in Ableton.



The port 6, in IN for Ableton and OUT for Skini allows the control from Skini.



This is only used when using MIDI or MIDI/OSC sync mode with a gateway. With Link this is not necessary.



Port 12 is used for "feedback" information from Ableton, when a clip is played.



For more details, especially with Bitwig Studio, see the appendices of this document.

Note: The *busMidiDAW* parameter in the configuration file of a piece is only useful when using the *Processing* gateway. It is an index that allows *Processing* to find the MIDI port to send the *noteOn* to the DAW. If the *Processing* gateway is not used, *busMidiDAW* is not used. The use of the gateway in the case of Node.js only occurs when the DAW and the server are on two different machines because we can speak MIDI from node.js.

2.5 ROOM CONFIGURATION

The pieces are configured from a window opened by the "Parameters" button in the main Blockly window (Patterns are configured by clicking on the "Patterns" button).

For the curious: The piece is configured in a JavaScript file, this file is loaded at the same time as the orchestration. It is selected from the programming window of the orchestration. The name of this file must correspond to the orchestration. For example for the orchestration *opus1.xml* we must have the configuration file *opus1.js*.

We will review the room configuration parameters.

2.5.1 How to receive MIDI commands

For the parameter "Direct Midi" checked means that the communication between Skini and the DAW is done via MIDI. Otherwise, the communication between Skini and the DAW is via OSC. OSC is used with the *Processing* gateway mainly for Ableton Live, Bitwig Studio can communicate directly in OSC by installing the Skini controller.

2.5.2 Reaction mode

The "React on Play" parameter defines the way to make the orchestration automaton react. By default, it is on selection. With "React on Play" checked, it is at the moment the pattern is played. This has an important impact on the way the automaton is thought. *Stingers* (for transitions) are only possible with "React on Play" active.

2.5.3 Sound files for clients and customers

The Sound Files Path parameter defines the path of sound files, associated with patterns, which are downloaded by clients from the `.\sounds` directory.

The "Number of client groups" parameter sets the number of groups of people in the audience that the orchestration can handle.

With the parameter "Simulator in a separate Group" we introduce the possibility to dedicate a group to the simulator. This means that the patterns available for the simulator will not be seen by the audience. For this we need a number of groups greater than 2, because the last group will be the simulators. If the "Simulator in a separate group" parameter is not checked, it means that there is no group dedicated to the simulator.

The parameter "Algo Fifo management" allows to activate with an integer a processing algorithm on the queues (see below).

2.5.4 Synchronization

There are 4 possible modes of synchronization. By Midi, by Midi via OSC, by Ableton Link and locally with a node.js worker. You can only have one mode at a time.

The Midi mode is the simplest, it works with all DAWs that can send a Midi sync. It is possible to communicate using OSC with Bitwig studio because there is a Skini controller that allows this. For Ableton you will have to use the Processing gateway. Ableton Link is very simple and allows you to synchronize in a network.

The Node.js worker is useful for projects without DAW, so with Musicien or Raspberry that don't give sync, and without external Ableton Link sync from any software. It's mostly a test tool when you don't have a DAW.

To choose the synchronization mode we have the parameters:

- *Syncho On Midi Clock* is used for MIDI synchronization from the DAW
- *Syncho Link* is used for synchronization from the Ableton Link protocol.
- *Syncho Skini* is used for a synchronization since Skini manages its own synchronization according to a *timer* parameter that gives the tick in ms. In this case we can't change the tempo in the orchestration.

If all these synchronizations are inactive Bitwig can send the sync to OSC via the Bitwig Skini controller.

Warning:

- 1) You have to restart Skini when you change the synchronization mode.
- 2) You should not have the Bitwig OSC sync via the Skini0 controller at the same time as another sync. You would receive too many potentially offset and duplicate sync messages.

2.5.5 Groups of patterns

The naming of the pattern groups is done via a table accessible from the main window and the "Parameters" button. The names of the pattern groups are used to create the HipHop signals of the orchestration. Here is an example:

	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupe0	0	group	20	50	20	#CF1919		1
2	groupe1	1	group	20	200	20	#008CBA		1
3	groupe2	2	group	20	350	20	#4CAF50		1
4	groupe3	3	group	20	500	20	#5F6262		1
5	groupe4	4	group	20	600	20	#797bbf		1
6	groupe5	5	group	200	50	20	#008CBA		1
7	groupe6	6	group	200	200	20	#E0095F		1
8	groupe7	7	group	200	350	20	#A76611		1
9	groupe8	8	group	200	500	20	#b3712d		1
10	groupe9	9	group	200	600	20	#666633		1
11	groupe10	10	group	340	50	20	#039879		1
12	groupe11	11	group	340	200	20	#315A93		1
13	groupe12	12	group	340	350	20	#BCA104		1
14	groupe13	13	group	340	500	20	#E0095F		1
15	groupe14	14	group	480	50	20	#E0095F		1
16	groupe15	15	group	480	200	20	#E0095F		1
17	groupe16	16	group	480	350	20	#E0095F		1
18	groupe17	17	group	480	500	20	#E0095F		1
19	groupe18	18	group	480	600	20	#E0095F		1

2.5.6 Musicians (Not in place)

The presence of musicians must be specified with the lines:

1. `exports.withMusician = true;`
2. `exports.decalageFIFOwithMusician = 4;`
3. `exports.patternScorePath1 = "";`

`decalageFIFOwithMusician` gives the pulse count before the first pattern is played. Indeed, a musician needs to prepare himself before playing a pattern, unlike a DAW. This parameter introduces a systematic delay if there is no pattern in the queue for the instrument concerned. If there are patterns in the queue, the musician client displays the pattern following the current one. This allows the musician not to be surprised.

The `patternScorePath1` is the subdirectory of the `./images` directory where the orchestration scores are located.

In the musician client you must log in with the instrument number. This is what allows the `server.js` to manage the messages as images.

2.6 PATTERN CONFIGURATION

It is done from the window opened from the orchestration page by clicking on "Patterns". (This generates a csv file that is located in the directory specified in the room configuration.)

Skini note		Send Note		CC command		CC value		Send CC						
IP OSC		OSC Message		OSC Value		Send OSC		CLOSE						
	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration	IP address	Buffer num	Level
1	11	510	0	FM8-1	Piano1	1	0	4	0	0	4			
2	12	510	0	FM8-2	Piano2	1	0	4	0	0	4			
3	14	510	0	Biopoly1	Piano9	2	0	4	0	2	4	192.168.1.24		60
4	15	510	0	Biopoly2	Piano10	2	0	4	0	2	4	192.168.1.24		60
5	16	510	0	Capa1	Capa1	3	0	4	0	3	4			
6	17	511	0	Capa2	Capa2	3	0	4	0	3	4			
7	18	511	0	MassiveX1	MassiveX1	4	0	4	0	4	4	192.168.1.28		70
8	19	511	0	MassiveX2	MassiveX2	4	0	4	0	4	4	192.168.1.28		70
9	30	511	0	Gam1	Gam1	5	0	4	0	5	4			
10	31	511	0	Gam2	Gam2	5	0	4	0	5	4			

The first column gives the MIDI note corresponding to the pattern in the DAW. These notes do not correspond exactly to MIDI notes. Indeed, to simplify the coding we have not imposed any limits on these numbers, unlike the MIDI standard which only allows 128 notes on a channel. We can go beyond the limit of 127. Transcription into MIDI notes consists in applying the following calculation:

1. `var channel = Math.floor(note / 127) + 1;`
2. `score = score % 127;`

We can therefore see that the notion of MIDI channel is included in the Skini *note*. The equivalence between Skini notes and MIDI notes is therefore only immediate for notes < 127. This makes it possible to associate MIDI commands with patterns without any constraint on the channels. This method allows us to free ourselves from tedious channel management.

The **Note Stop** column, is the MIDI note that allows you to interrupt a running pattern. This is specific to Ableton Live, it is not useful in general.

Flag Usage is not a parameter, it is a tool for the Skini engine.

The **name** column gives the texts that will be associated to each pattern for the different clients. The column *sound file*, gives the names of the sound files, which are in the directories defined in the JavaScript configuration file, by default they are mp3 files. To use *wave* files, you must add a ".wav" extension to the file names.

The **instrument** column associates patterns with a specific instrument that corresponds to a MIDI instrument or musician. There is no correspondence between these numbers and a MIDI configuration.

The **slot** column is related to ongoing developments in live pattern recording. They can be ignored for Skini sessions without live recording. The same applies to the last two columns, "waiting" and "pseudo".

The **type** column allows you to qualify a pattern in order to reorganize the queues; the description of this functionality is in the chapter 37, "Priority in queues".

The **group** column corresponds to the group indexes described in the group configuration (parameters). It is this parameter that associates the pattern to a group and allows the provision of patterns via the orchestration.

The **duration** column defines the length of the pattern in number of **pulses** emitted by the synchronization, usually MIDI.

The **IP address** column allows to associate a pattern to an OSC command when Skini works with distributed devices like Raspberries Pi with an adapted application. Always with Raspberries Pi **Buffer Num** is a parameter of the OSC command which defines the buffer associated with the pattern. **Level** is used to define the sound level for this pattern. If the **buffer num** field is empty the pattern is activated on the DAW. It is possible to send OSC commands from this window to test distributed patterns.

2.7 CONFIGURATION OF THE ORCHESTRATION DISPLAY ON A LARGE SCREEN

It is possible to display a flow of the orchestration in a browser. Skini offers a display based on connected boxes. Groups are represented by rectangles and tanks by rectangles with rounded edges. The size of the tanks reduces as they are emptied. A visual of the orchestration can be accessed from the programming window or with the url :

`http:// 'server'/score`

The parameterization of this display is done from the "Parameters" window of the piece. Below we will see an extract of a graph setting for opus1. For each group of patterns, we have fields giving the position of the rectangles, a color, a list of predecessors i.e. groups pointing to the group of the line and a scene number. The format is a little different for groups and tanks. In brackets it is the index of the row's table.

For groups: group name (0), index (1), type (2), x(3), y(4), number of elements(5), color(6), predecessors(7), graphic scene number(8)

For tanks: tank name (0), index (1), type (2), x(3), y(4), tank number(5), color(6), Previous(7), graphic scene number(8) .

	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupe0	0	group	20	50	20	#CF1919		1
2	groupe1	1	group	20	200	20	#008CBA		1
3	groupe2	2	group	20	350	20	#4CAF50		1
4	groupe3	3	group	20	500	20	#5F6262		1
5	groupe4	4	group	20	600	20	#797bbf		1
6	groupe5	5	group	200	50	20	#008CBA		1
7	groupe6	6	group	200	200	20	#E0095F		1
8	groupe7	7	group	200	350	20	#A76611		1
9	groupe8	8	group	200	500	20	#b3712d		1
10	groupe9	9	group	200	600	20	#666633		1
11	groupe10	10	group	340	50	20	#039879		1
12	groupe11	11	group	340	200	20	#315A93		1
13	groupe12	12	group	340	350	20	#BCA104		1
14	groupe13	13	group	340	500	20	#E0095F		1
15	groupe14	14	group	480	50	20	#E0095F		1
16	groupe15	15	group	480	200	20	#E0095F		1
17	groupe16	16	group	480	350	20	#E0095F		1
18	groupe17	17	group	480	500	20	#E0095F		1
19	groupe18	18	group	480	600	20	#E0095F		1

Setting up tanks is more complex than setting up groups. A tank is a set of groups. A tank is defined by the field in the fifth position. All groups/patterns in a tank have the same tank number.

Attention must be paid to the numbering of the predecessors. A group or a tank can be used as a predecessor.

In the configuration below we have added at the end of the line the number of the group as predecessor. Up to line 18 everything is simple. At line 18 we start a tank that ends at line 21. These 4 lines are in the same "predecessor" of value 11. In line 22 begins the "predecessor" of value 12. (The color must be expressed in hexadecimal format).

```
1. [ "violinsScale",    0, "group",    323, 176, 12, ochre, [3,8], 1], //0
2. [ "violinsChrom",   1, "group",    800, 180, 16, ochre, [9], 2], //1
3. [ "violinsTonal",   2, "group",    450, 25, 24, ochre, [30], 3], //2
4. [ "altosScale",     3, "group",    200, 114, 12, purple, [5], 1], //3
5. [ "altosChrom",     4, "group",    800, 270, 16, purple, [9], 2], //4
6. [ "cellosScale",    5, "group",    52, 173, 10, green, [], 1], //5
7. [ "cellosChrom",    6, "group",    800, 360, 16, green, [9], 2], //6
8. [ "cellosTonal",    7, "group",    650, 100, 8, green, [2], 3], //7
9. [ "ctrebassesScale", 8, "group",    200, 244, 12, blue, [5], 1], //8
10. [ "ctrebassesChrom", 9, "group",    590, 430, 16, blue, [25, 29], 2], //9
11. [ "ctrebassesTonal", 10, "group",    650, 160, 6, blue, [2], 3], //10
12. [ "trumpetsScale1", 11, "tank",    273, 374, 1, orange, [5,0], 1], //11
13. [ "trumpetsScale2", 12, "tank",    200, 10, 1, orange, [], ],
14. [ "trumpetsScale3", 13, "tank",    200, 10, 1, orange, [], ],
15. [ "trumpetsScale4", 14, "tank",    200, 10, 1, orange, [], ],
16. [ "trumpetsTonal1", 15, "tank",    650, 280, 2, orange, [2], 3], //12
17. [ "trumpetsTonal2", 16, "tank",    200, 100, 2, orange, [], ],
18. [ "trumpetsTonal3", 17, "tank",    200, 100, 2, orange, [], ],
```

The orchestration display is triggered by the orchestration.

Note: The score client uses the JavaScript version of Processing, P5js. The source code can be found in `./Processing/P5js/score/score.js`

3 LAUNCH SKINI

For Skini to work with musicians without electronics, only Node.js is needed. With electronics you need :

- A Digital Audio Workstation (for example Ableton Live or Bitwig Studio).
- Eventually Processing of MIT which will bridge the gap between OSC and MIDI.

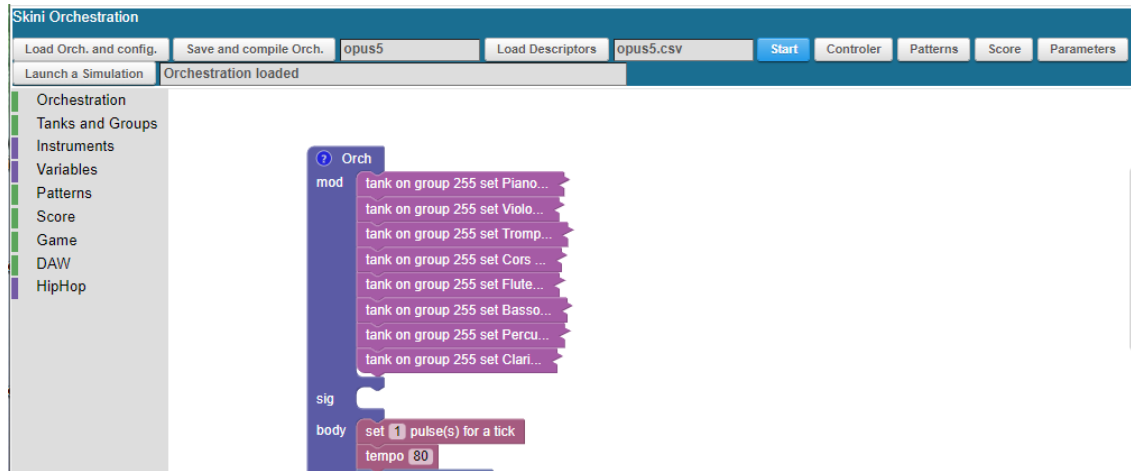
To launch Skini you need:

- Throwing: node skini

With electronics it is then necessary:

- 1 If you use the OSC/MIDI gateway, launch *Processing* with the program *sequencerSkini.pde*. The Processing console reports that the gateway has been connected to the server. Remember that with Bitwig studio and the Skini_0 controller, this gateway is not used.
- 2 Launch and load the patterns in the DAW. Start playback on the DAW to activate the MIDI synchronization if it is used. The Processing display should scroll, which means that Skini is on its way.

It is now possible to load the orchestration programming in a web browser with *http://IP of the server :8080/block*. Skini is ready for a performance.



The orchestration window gives access to the controller, to the configuration and to the display of a graphical follow-up of the piece (score).

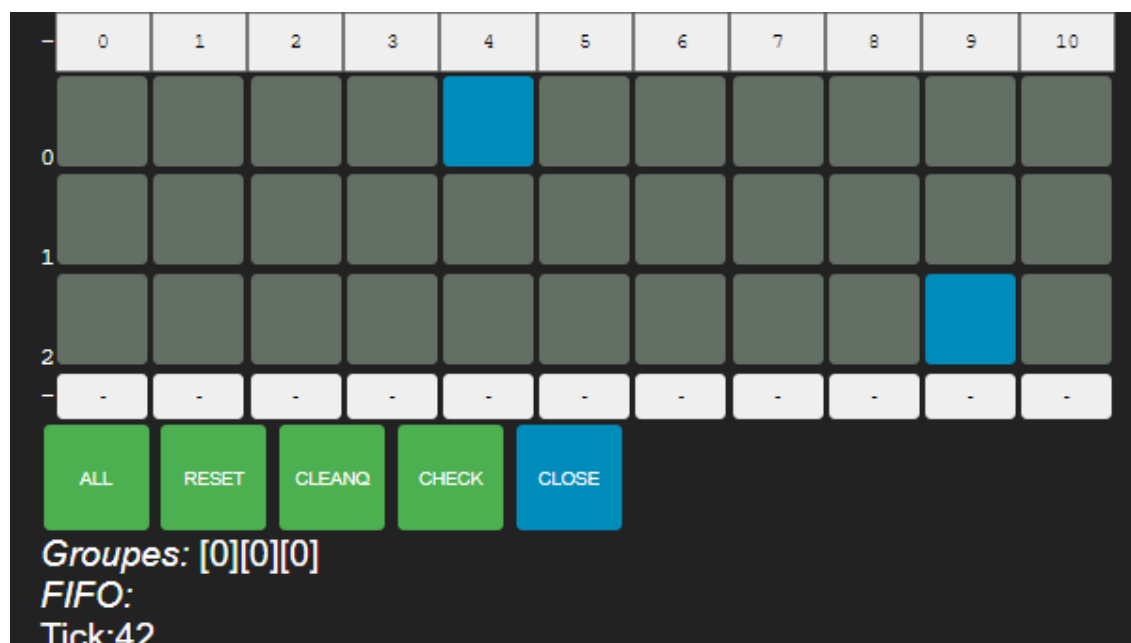
4 USING THE CONTROLLER

The controller is opened from the orchestration window. It gives a view of the groups in the audience and the groups of patterns. It also allows to control in real time the *matrix of possibilities* which is displayed as a table with as rows the groups in the audience as defined in the configuration file of the room and as columns the pattern groups. The controller can activate or deactivate a pattern group by clicking in the matrix. Clicking on the pattern group number activates or deactivates all the groups in the audience for this group.

Caution: The correspondence between the controller indexes and the pattern group numbers is only valid if the group numbers follow each other in the configuration file of the piece. The controller indexes correspond to the row in the pattern group table and not to the group index. (to be reviewed, if possible).

The " ALL " button activates all groups. The " RESET " button deactivates all groups. The " CLEANQ " button clears all queues.

The " CHECK " button displays the pattern configuration file in the Skini console.



The Group, Scrutineer and FIFO displays provide system status.

5 USING THE MIDI CONFIGURATOR

This tool is used to set up the DAW and the pattern configuration file, also called descriptor.

Usually, DAWs have a function to listen to MIDI commands from controllers. To facilitate the implementation of the pieces, Skini can send commands like a controller. The MIDI Configurator

therefore behaves like a MIDI controller. Skini converts the "Clip" commands into a MIDI channel and note. The Configurator allows you to send notes with "Send Clip" and MIDI "Control Changes (CC)" with "Send CC". The first field above "Send CC" is for the CC, the field to the right of it allows you to give a value to the CC. The configurator can be accessed from the orchestration (or with <http://adresse of the server/conf>). The configuration of patterns is presented in the chapter " Pattern configuration ".

Skini note		Send Note		CC command		CC value		Send CC						
IP OSC		OSC Message		OSC Value		Send OSC		CLOSE						
	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration	IP address	Buffer num	Level
1	11	510	0	Rubinstein	Piano1	1	0	1	0	10	4			
2	12	510	0	Kempff	Piano2	1	0	1	0	11	4			
3	13	510	0	Gould	Piano3	1	0	1	0	12	4			
4	14	510	0	Cziffra	Piano4	1	0	1	0	13	4			
5	15	510	0	Abramovitz	Piano5	1	0	1	0	14	4			
6	18	510	0	Levinas	Piano6	1	0	2	0	17	4			
7	19	510	0	Cymerman	Piano7	1	0	2	0	18	4			
8	20	510	0	Cortot	Piano8	1	0	2	0	19	4			
9	21	510	0	Ciccolini	Piano9	1	0	2	0	20	4			
10	22	510	0	Casadesus	Piano10	1	0	2	0	21	4			
11	23	510	0	Boulanger	Piano11	1	0	2	0	22	4			
12	24	510	0	Borchard	Piano12	1	0	2	0	23	4			
13	25	510	0	Pleyel	Piano13	1	0	3	0	24	4			
14	26	510	0	Gaveau	Piano14	1	0	3	0	25	4			
15	27	510	0	Erard	Piano15	1	0	3	0	26	4			

6 USING THE SKINI CLIENT

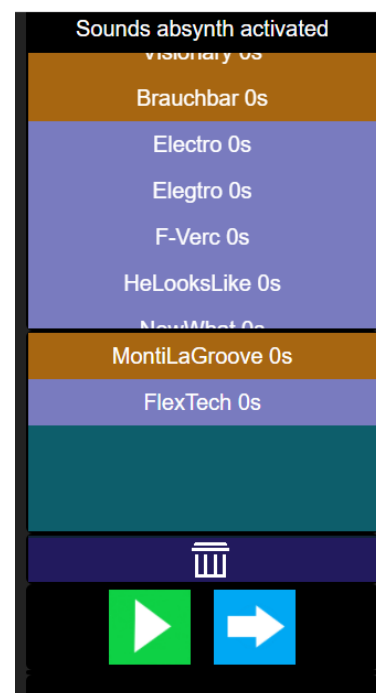
The client, which is called by the URL <http://<serveraddress:port>/skini>, allows audience members to create a list of patterns and send it to the server. The client can feed a list (bottom part of the screen) from the available pattern lists (drag and drop). The green button plays the list locally, the blue button sends the list to the server. The trash can is used to delete items from the list.

Note: The patterns in the list of choices will fill the server queues. If in a list we have patterns for different instruments, the queues will be fed. We will not have the sequence heard locally between the instruments. The sequences are respected by instrument.

This client comes with orchestration functions that allow to dynamically define the length of the list and to empty the list.

Once a list has been sent, the client will not be able to send another one until his requested list has been played completely.

This client coupled with an adequate orchestration allows to create musical games. For example, we can imagine letting groups of clients take turns to design lists of patterns during a period and to note the quality of these lists by giving winners and losers. The "pause/resume" mechanisms of the queues make it possible to manage the design and play periods of the patterns. We can couple a display accordingly on a large screen with the orchestration display tool for example. It is also possible



to communicate with a video game platform such as Unreal Engine via signals for more graphically rich musical games.

Important Note: Do not use queue reordering algorithms with this client if these algorithms can remove patterns from FIFOs. Indeed, the client waits for confirmations on the set of patterns to authorize the sending of a new (or the same) list. If a requested pattern disappears the client will be blocked.

The `cleanQueue` block should be handled with care. Indeed, clients are blocked until all patterns of a list have been played. If the `Fifo`'s are empty, some clients can be blocked because the patterns have disappeared from the `Fifo`'s and will never be played. It is better to do "`cleanChoiceList`" (255) with a "`cleanQueues`".

If the patterns are typed the server will evaluate a score according to the relevance of the succession of types in the list defined by a client. The scoring algorithm is in `websocketserver.js`, in the `computeScore()` function. The scoring rules are to be modified in the source code. The mechanism of evaluation of the scores is based on the pseudos.

The orchestration accesses the current winner with the "`display score`" blocks.

7 USING THE SIMULATOR

The simulator is a tool that allows to test the behavior of a piece before its use with an audience. It also allows the activation of patterns in a random way during a performance with an audience. It has two basic behaviors that are defined in the parameters of the piece with the fields:

Number of client groups
Simulator in a separate Group

The purpose of defining the number of client groups is to give the number of groups of people in the audience that the orchestration can manage independently. The assignment of a group to a member of the audience is done in a cyclic way. Each member is assigned a group according to its predecessor at the time of connection

The `Simulator in a separate Group` parameter, when checked, means that the last client group is reserved for the simulator. The audience will not have access to it. Otherwise, it means that the simulator will be able to behave like any other group in the audience.

The audience simulator is launched either from the blockly programming window or with the command

```
./nodeskini/client/simulatorList/node simulatorList.js
```

The simulator outside the audience on the last "group of people" when `Simulator in a separate Group` is checked, it is launched from the main window or with the command

```
./nodeskini/client/simulatorList/node simulatorList.js -sim
```

The simulator has a mechanism that avoids two successive repetitions of the same pattern over three selections.

The simulator is set in the configuration file of the piece with the lines:

Max Tempo In ms
Tempo Min In ms
Limit Waiting Time (in pulse)

Each call to the server is made at a time defined by:

$\text{tempoInstant} = \text{Math.floor}(\text{Math.random()} * (\text{tempoMax} - \text{tempoMin})) + \text{tempoMin};$

It is therefore a random duration between two limits tempoMax and tempoMin.

Limit Waiting Time is the parameter that defines how long a pattern will wait before the simulator calls the server.

NB: The simulator includes a mechanism to avoid the repetition of the same pattern in a history of 3 previous patterns. This is the `selectRandomInList` function.

8 ORCHESTRATION PROGRAMMING

For the programming of orchestrations, the composer accesses an interface using the *Blockly* solution provided in open source by Google. The handling of Blockly is simple and user-friendly. This interface is used by Scratch, the programming learning tool for children.

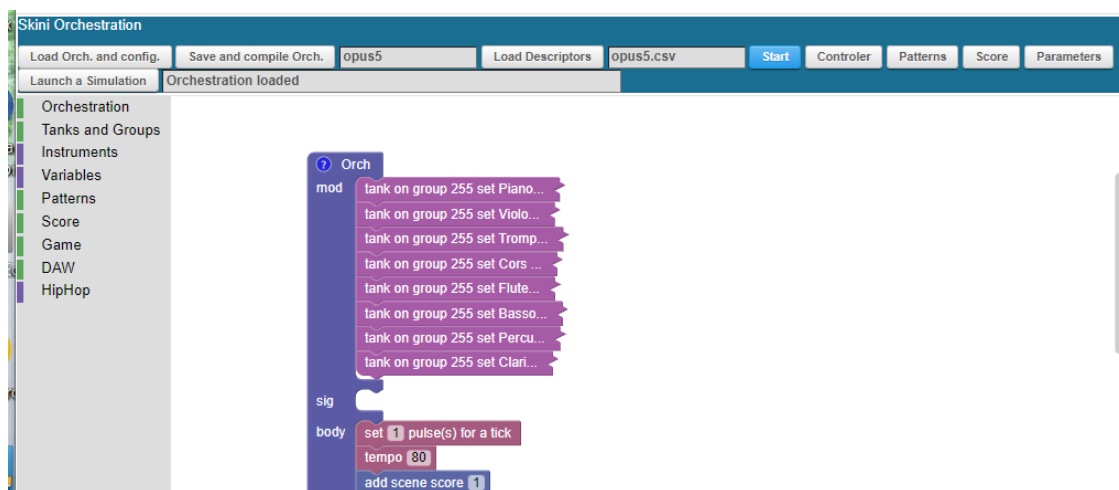
We do not discuss here how Blockly works. A visit to the site

<https://developers.google.com/blockly>

will be completer and more effective than a general presentation in Skini.

8.1 THE SKINI GRAPHICAL INTERFACE

The Blockly interface generates HipHop.js programs without needing to know how to program with this language. The composer can load an orchestration with "Load Orch. and config".



The "Save and Compile" button does two things. It saves the current file in an *xml* file with the name entered in the text field. It creates a HipHop.js file. The "Start" button allows you to launch the orchestration.

"Launch a Simulation" allows you to launch a simulator without using a console.

"Patterns" opens the pattern descriptor configuration window. "Score " opens the pattern block display window and " Controller" the control window. "Parameters " opens the piece parameters window.

Note: The controller gives more information about the active pattern groups and ticks. The information on the groups is visible with the orchestration display (Skini "score" client), if this is foreseen in the room configuration.

We will review the main blocks of the Blockly interface.

Note: "Save and compile" compiles Blockly into a `./myReact/orchestrationHH.js` file. This file is not useful for the composer. The path is set in `websocketServer.js` with the variable `generatedDir`.

8.2 ORCHESTRATION BLOCKS

These are the blocks that define the structure of the orchestration.

The first module is essential. "Orch."



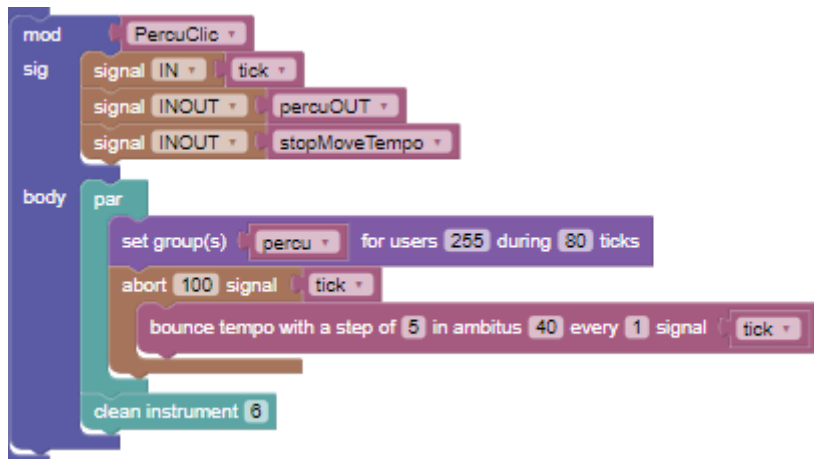
The blockly code is organized in blocks that can be paralleled with the:



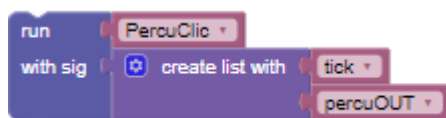
The "mod" field is used to create "Modules", such as the tanks we will see below, which will be called in the body of the orchestration ("Body"). The "sig" field is used to declare signals used in the orchestration.

The use of modules with groups or tanks requires "implicit" signals, i.e. signals that do not appear in the Blockly orchestration. To a group activation corresponds a signal `xxxOUT` where `xxx` is the name of the group and a signal from a group `xxxIN`. There are also default signals in the main program which are not default in the modules and which must be declared as input or output in the modules if necessary: *start*, *halt*, *DAWON*, *tick*, *patternSignal*, *controlFromVideo*, *pulsation*, *midiSignal*, *emptyQueueSignal*, *stopReservoir*, *stopMoveTempo*

Here is an example of a module that uses implicit signals, here *stopMoveTempo* linked to the bounce tempo " block and *percuOUT* linked to the "percu" group.



The module is called with:



8.3 HIPHOP BLOCKS

They correspond to the commands of the synchronous reactive language HipHop. See the Skini tutorial for a complete overview. Here is an example:



By default, the blocks are executed one after the other, but for readability reasons or when you want to put sequences of instructions in parallel, it is possible to group them together with the "seq" block:



The interest of this block is to be able to easily activate the "collapse block" function of Blockly to make the orchestration more synthetic.

8.4 THE PROGRAMMING OF TIME IN A ROOM

The function of time management in Skini is to ensure a good coherence in the sequence of patterns and a great flexibility in programming. It is done according to two main parameters:

- The *pulse* which is provided either by a midi synchronization, or Ableton Link or directly by Skini (cf. the parameterization of a piece)
- The *tick* which is a multiple of the *pulse* and which is used by the mechanism of reading (emptying) the queues.
 - Use of the tick signal

Defining a *tick* as a multiple of the pulse allows us to define the cycle on which the pattern starts in the queues will be synchronized. If we set a *tick* to 4 pulses, each pattern in a queue will be launched at the earliest at the beginning of a cycle of 4 pulses. At the earliest because a pattern can last more than 4 pulses, the next one will have to wait for the end of the previous one. This mechanism avoids overlapping patterns for the same instrument.

A pattern is placed in a queue each time it is selected, and the queues are unstacked at regular intervals that are multiples of the pulse, which we call a *tick*. This interval is fixed with :

set 1 pulse(s) for a tick

For pieces with patterns of the same duration, we can give the *tick* this duration (a certain number of pulses) which ensures that the pattern starts at the same time.

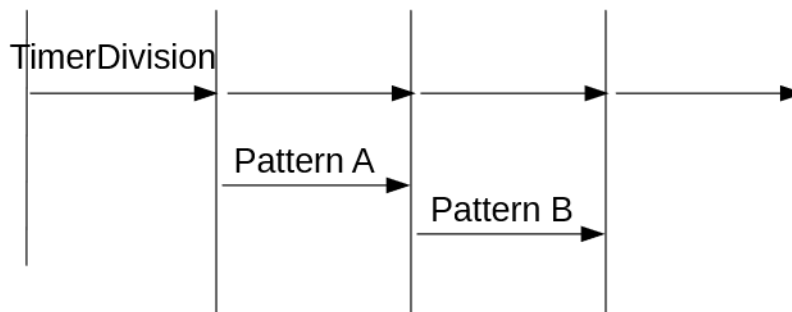
As mentioned above, when you want to introduce patterns of different durations, it is possible to give the *tick* a value corresponding to the duration of the shortest pattern. You have to pay attention to the correspondence between the *tick* duration and the pattern durations. If a pattern has a duration that is not a multiple of a *tick*, Skini will not play it and will give an error message. If you set the *tick* to 4 beats and you have a pattern with 3 beats or 5 beats it will not be played.

The combination of *tick* duration and pattern can lead to complex behaviors. Consider the example where patterns have multiple *tick* durations. Consider patterns that have the same duration, for example 16 pulses and a *tick* equal to 4 pulses. If a pattern A is requested (queued)

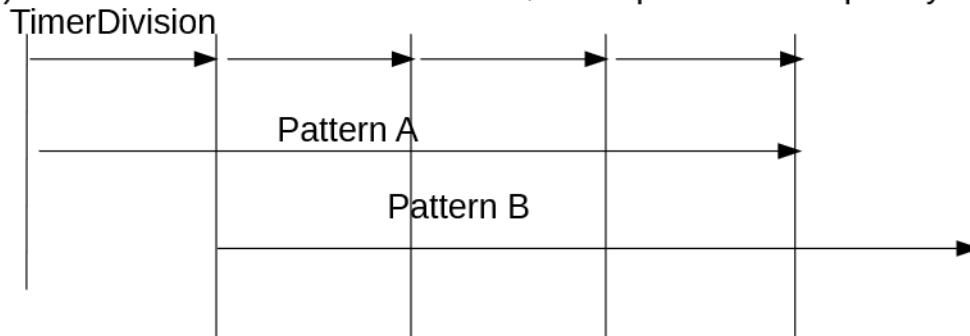
at time T and a pattern B at time T + 5 pulses for example (between T + 5 and T + 8). Pattern B will start a *tick* cycle after pattern A. So the 16-pulse patterns, although all of the same duration, can be shifted in relation to each other according to multiples of the *tick*.

The following diagram illustrates the mechanism. Patterns A and B are on two different instruments. *TimerDivision* is equivalent to *tick*.

1) Pattern de durée = timerDivision, les départs sont synchronisés



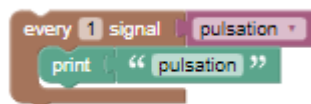
2) Pattern de durée != timerDivision, les départs ne sont pas synchronisés



When using different pattern durations, it is therefore necessary to ensure musical consistency over the duration of the *tick* cycle.

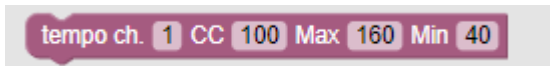
8.4.1 Use of the pulse signal

In a room it is possible to manage the time with pulses. To do this you must activate "Pulsation ON" in the room parameters. It is then possible to manage a pulsation signal (declared by default). Here is an example of the display of a message for each pulse.

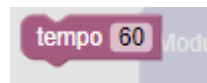


8.5 TEMPO

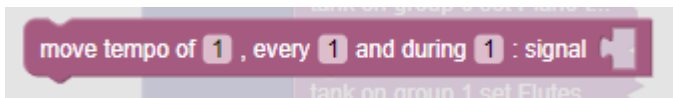
When not using the sync link, changing tempi from the orchestration requires the DAW to be set up to receive Control Changes on the tempo. We have seen that a MIDI control bus is used to send MIDI information to the DAW. It is this bus that is used for tempo controls. In the case of Ableton, the control function needs to declare the parameters used by Live for this control, i.e., a maximum and a minimum value for the tempi. It is the following block that sets these parameters.



Once these parameters are set or directly when using Link. The block

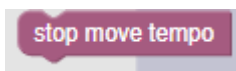


Allows you to intervene on the tempo at any moment of the orchestration. The block



allows an automation of the tempo change. It allows the tempo to be varied by a fixed value at "every" occurrence of the signal given in parameter. The tempo variation is reversed after "during" occurrence of the same signal. This block allows easy integration of tempo movement without programming.

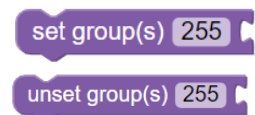
To interrupt the tempo movements, you must apply the block:



8.6 RESERVOIRS AND GROUPS OF PATTERNS

Although both are pattern set processing, they are two quite different classes of processes. The groups are controlled by signals, the activation and deactivation actions are not blocking.

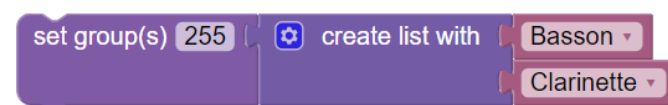
8.6.1 Groups of patterns



The user groups are defined in the configuration file of the room. They are numbered from 0 to 254. The group 255 is in fact all the groups. The following block activates the Basson group for all users:

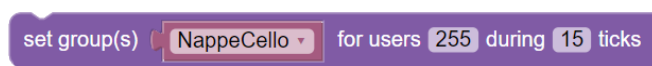


You can activate several groups with lists, ex:



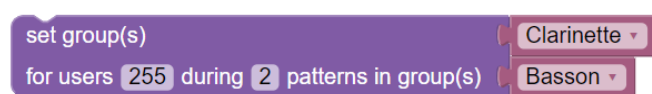
8.6.2 Actions related to groups

Skini provides high level blocks to allow complex actions with groups.

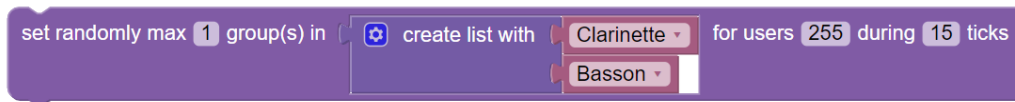


activates one or more groups during a period. Lists can be used instead of a

single group.

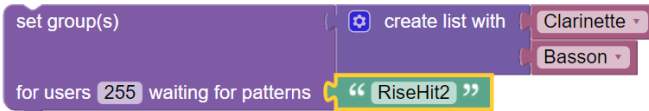


activates one or more groups waiting for patterns played in one or more groups.



allows

you to randomly activate "max" groups from a list during a period.



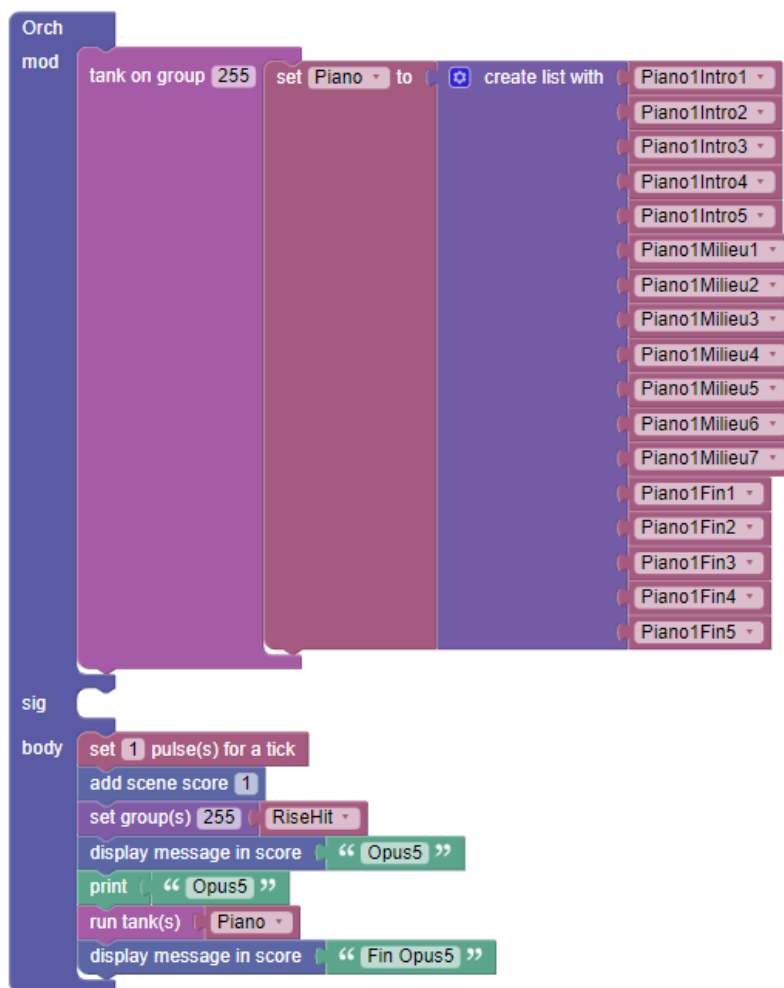
activates one or more groups waiting for one or more specific patterns.

Note: Groups are Blockly variables, patterns here are Blockly strings.

8.6.3 Creation of tanks

A reservoir is a three-level module that is therefore placed in the "Mod. Orchestration". First a list of patterns, then a variable associated with this list. The variable is put in a "tank". A tank is associated with a group of users or all users. Here is an example of a "percussion" tank of percussion patterns assigned to all user groups (255). Patterns are variables or strings containing the name of the patterns as described in the pattern's csv file.

Here is an example of tank use:



With the patterns :

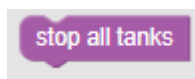
	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration
1	11	510	0	Rubinstein	Piano1	1	0	1	0	10	4
2	12	510	0	Kempff	Piano2	1	0	1	0	11	4
3	13	510	0	Gould	Piano3	1	0	1	0	12	4
4	14	510	0	Cziffra	Piano4	1	0	1	0	13	4
5	15	510	0	Abramovitz	Piano5	1	0	1	0	14	4
6	18	510	0	Levinas	Piano6	1	0	2	0	17	4
7	19	510	0	Cymerman	Piano7	1	0	2	0	18	4
8	20	510	0	Cortot	Piano8	1	0	2	0	19	4
9	21	510	0	Ciccolini	Piano9	1	0	2	0	20	4
10	22	510	0	Casadesus	Piano10	1	0	2	0	21	4
11	23	510	0	Boulanger	Piano11	1	0	2	0	22	4
12	24	510	0	Borchard	Piano12	1	0	2	0	23	4
13	25	510	0	Pleyel	Piano13	1	0	3	0	24	4
14	26	510	0	Gaveau	Piano14	1	0	3	0	25	4
15	27	510	0	Finck	Piano15	1	0	3	0	26	4

And the groups :

	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	Piano1Intro1	10	tank	22	151	1	#b3712d		1
2	Piano1Intro2	11	tank			1	#b3712d		
3	Piano1Intro3	12	tank			1	#b3712d		
4	Piano1Intro4	13	tank			1	#b3712d		
5	Piano1Intro5	14	tank			1	#b3712d		
6	Piano1Milieu1	17	tank			1	#b3712d		
7	Piano1Milieu2	18	tank			1	#b3712d		
8	Piano1Milieu3	19	tank			1	#b3712d		
9	Piano1Milieu4	20	tank			1	#b3712d		
10	Piano1Milieu5	21	tank			1	#b3712d		
11	Piano1Milieu6	22	tank			1	#b3712d		
12	Piano1Milieu7	23	tank			1	#b3712d		
13	Piano1Fin1	24	tank			1	#b3712d		
14	Piano1Fin2	25	tank			1	#b3712d		
15	Piano1Fin3	26	tank			1	#b3712d		
16	Piano1Fin4	27	tank			1	#b3712d		
17	Piano1Fin5	28	tank			1	#b3712d		

8.6.4 Actions on the reservoirs

Tanks are in fact Skini submodules having for parameters patterns. They are blocking in a flow. A tank will stop when it is empty or when it is killed with



There are functions that simplify the management of tanks:



This block will launch one or more tanks given in parameter as a Blockly variable and will "kill" this tank after "during" occurrences of a signal. The next block will leave the patterns of the Basson reservoir available for a maximum of 40 ticks:

run tank(s) Basson during 40 ticks

The following blocks are understood intuitively:

run tank(s) during 1 patterns in group(s)

run randomly max 2 tank(s) in during 1 ticks

run tank(s) waiting pattern(s) played by DAW

This block allows the DAW to take into account the set of specific patterns.

Note: The names of the groups and tanks are in direct correspondence with the configuration file of the music room.

Remarks: All **group** and **reservoir** processing uses **Blockly variables**. Everything that concerns **patterns** directly uses **strings**.

reset orchestration

Resets the orchestration matrix, all active groups are disabled.

8.7 INSTRUMENT QUEUES

Queues are associated with instruments. They are created according to the pattern description in the pattern csv file.

8.7.1 Emptying of queues

These blocks are useful when the composer wants no more pending patterns to be heard for an instrument. He can empty all the queues with :

clean all instruments

The skini client empties its choice list as soon as a clean all queue is performed. It can empty the queue of a particular instrument by giving its index with:

clean instrument 1

The skini client empties its choice list as soon as a clean queue is performed. It does not consider the FIFO number, as does `controleAbleton.js`. (The block does not perform a `cleanChoiceList`).

Attention: the queues, like the groups in the audience, are set with **indexes** that can start from **0**. The score displays are done with parameters that start at 1. (The best score is in 1).

8.7.2 Pause and test queues

pause all instruments

Stops the playing of patterns for all instruments.

resume all instruments

Resumes the playing of the patterns for all instruments.

pause instrument 1

Stops the playing of patterns for an instrument.

resume instrument 1

Resumes the playing of patterns for an instrument.

wait until instrument 1 is empty

Blocks the orchestration until the patterns of an instrument have been played.

8.7.3 Queueing a specific pattern

The composer can make a specific pattern play imperatively with the block.

put pattern in instrument

The parameter of the block is a string with the name of the pattern.

Ex:

put pattern BassonDebut1 in instrument

This block allows you to introduce "sequencer" type elements in a collaborative or generative music orchestration.

Warning: "put pattern" is sensitive to synchronization. Although Skini considers that the system actions are done in zero time, this is not actually the case. However, if in an instant you must receive the synchronization, put the pattern in the FIFO, read the FIFO and send a command to the DAW, Skini may find itself in a situation where the DAW command does not pass at the right synchronization tick. The solution is to introduce a delay at the launch of the pattern by the DAW (launch synchronization). In this case, the DAW launches the command after a short delay which allows the command to be received safely. The DAW then sends a synchronization tick and waits a little before launching the patterns.

8.8 PATTERNS

Skini being mainly intended for processing groups of patterns and tanks, there are few functions dedicated to patterns, here are two of them:

wait for 1 pattern(s) in group NappeAlto

This block will wait for the execution (or the request depending on the setting of the piece) of a certain number of patterns belonging to a group.

wait for pattern (string)

Is a block that waits for the execution (or the request according to the parameterization of the piece) of a certain pattern passed in parameter in the form of character string. For example:

wait for pattern (string) " Percu7 "

8.9 DISPLAY OF THE ORCHESTRATION

The display on a large screen of the orchestration progress is controlled by the orchestration with specific blocks:

add scene score 1

A display can be made on several levels. This block displays one of the levels. The levels are associated with the pattern groups in the room configuration file.

remove scene 1 in score

Make a level disappear

The following block allows you to refresh the display to follow up on operations on the queues for example:

refresh scene score

The following blocks allow you to display and hide a message in a popup window on the big screen:

display message in score

remove message in score

8.10 SKINI GAME

Skini provides a standard audience interface that allows participants to create lists of patterns and send them to the server to be played. The composer can use this interface to invent games, such as "find the right sequence of patterns within a group". The game scenarios are not programmed by the orchestration. They are JavaScript functions on the server.

A basic model is proposed which consists in defining types of patterns and associating notes to the way the participant will organize his lists. The winner will be the one who accumulates the best lists during the play.

What is possible from the orchestration is to act on the lists of the participants by defining their lengths and by emptying them imperatively the lists of a group.

set pattern list length to 3 for group 255

clean choice list for group 255

The following block displays the current best score:

display best score during 2 ticks

To display the other scores (and the best) we use the block

display score of rank 1 during 2 ticks

The "ranking" is counted from 1.

display group score of rank 1 during 2 ticks

set scoring policy 1 Defines the type of algorithm that will calculate the scores. These algorithms can be found in the ./server/computeScore.js file

set scoring class 1 Defines the class considered in the calculation of the score. The *class* is one of the parameters (type in index 7) of the patterns in the csv file describing the patterns. See the ./server/computeScore.js file on the treatment of the classes.

8.11 MIDI COMMAND AND CONTROL CHANGE

It is possible to issue "MIDI Control Changes" (CC) from the orchestration. It is up to the composer to define and parameterize the functions of the CCs in the DAW. The channel on which the CCs will be sent directly to the control must be defined.

sendMidi ch. 1 note 13 vel. 123 This block allows you to send a MIDI command

The choice of MIDI channel depends on how the MIDI assignments to clips in the DAW have been programmed. There is **a difference in the channel seen on Skini which starts at 1 and on DAWs where it can start at 0**. This difference does not require attention outside of these controls. Do not hesitate to test and compare the "sendMIDI" activation with the configuration tool for example.

Notes on MIDI control: The definition of the MIDI bus (MIDI port) associated with the Block is set in the configuration file. Blocks therefore never use this parameter.

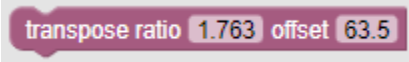
Notes on MIDI commands: In general, the operations performed by the Skini blocks do not intervene on MIDI notes (in the Skini sense) but on groups of patterns, tanks and even patterns

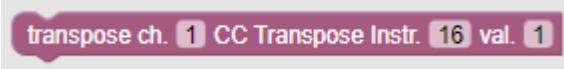
but using their names. This is true except for the sendMIDI and sendCC blocks, which address a clip in the DAW directly.

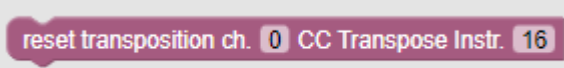
The composer must therefore pay attention to the MIDI channels in the case of blocks acting on Control Changes (CC) and MIDI commands.

8.12 ABLETON LIVE SPECIFIC

These blocks are used to control the transposition tool of Ableton Live. The ratio and offset are to be calculated according to this tool.

 Defines the ratio and offset for the transposition calculation.

 Transposition block, the values are in semitones. The MIDI channel number must be given. The CC command is the one that controls the Ableton transposition tool.

 Block to reset the transposition. The previous command performs an increment.

8.13 OSC COMMANDS FOR RASPBERRY

To send an OSC command from the orchestration you must use the sendOSC block of the DAW menu. Here is an example:



Which sends the message OSC `/level` with the value 100 to the address 192.168.1.34 on the port "Raspberry OSC Port" of the parameters. In the current version only one value is passed associated with the command.

8.14 ADVANCED BLOCKS

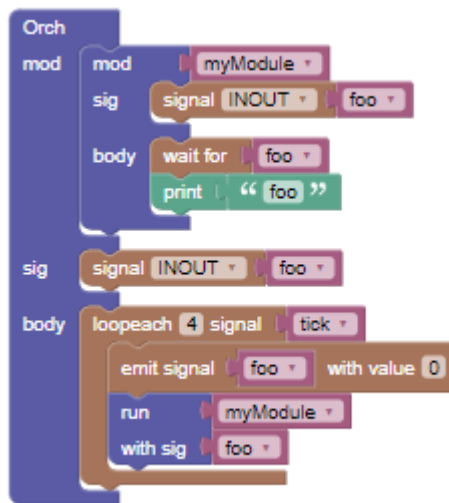
It is possible to create complex orchestrations without knowing the details of HipHop, but for functions based on non-standard blocks there is a series of menus allowing to program in HipHop with Blockly. This assumes that you have the basics of synchronous reactive programming.

These menus are: Signals, Signals Advanced and Module Advanced.

Beyond the tanks, it is possible to create Skini sub-modules.



These sub-modules have input/output signals that must be created in the signal field. The body of the sub-module is in body. Here is an example of a sub-module:



A submodule is executed with the run command:

8.15 CONTROL INTERFACE Z

Skini integrates Interface Z's 16 input (8 ana - 8 num) OSC network card. This card sends OSC messages according to the connected sensors. It operates on a wired internet network. The Skini interface considers the 8 analog inputs.

The configuration of the card is done in *ipConfig.json* with:

- *interfaceZIPaddress*, IP address of the InterfaceZ card.
- *portOSCToInterfaceZData*, for the port that sends the commands from the analog and digital sensors.
- *portOSCToInterfaceZMidi*, for the port that sends commands from the MIDI interface of the card (not used at the moment).
- *portOSCFromInterfaceZ*, for the port on the card that receives MIDI commands.

Example of configuration: The interface is programmed in 192.168.1.250 with data port in 3005 and MIDI port in 3006. The installation works in wired mode on a switch with the IP address 192.168.1.251 for the PC.

The parameterization of the sensors is done in the parameter window:

☒ Sensor OSC

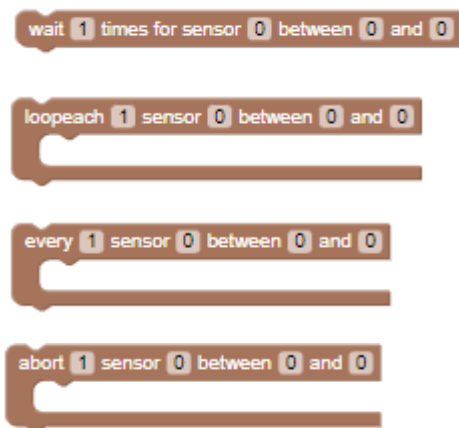
Sensors Temporisation

Sensors Sensibilities

The time delay defines the frequency at which we want to consider the information sent by the sensors. Indeed, the InterfaceZ card sends the state of the sensors in a systematic way at a frequency fixed by a potentiometer which allows to vary the emissions from 100 to 3000 messages per second. In order not to slow down Skini's response time, a *thread* is dedicated to processing these messages. The time delays fix for each sensor the frequency of considering the messages. 15 means that only one message out of 15 is considered. For example, for 100 messages per second from the card, we will have a signal every 150 ms to Skini instead of every 10ms.

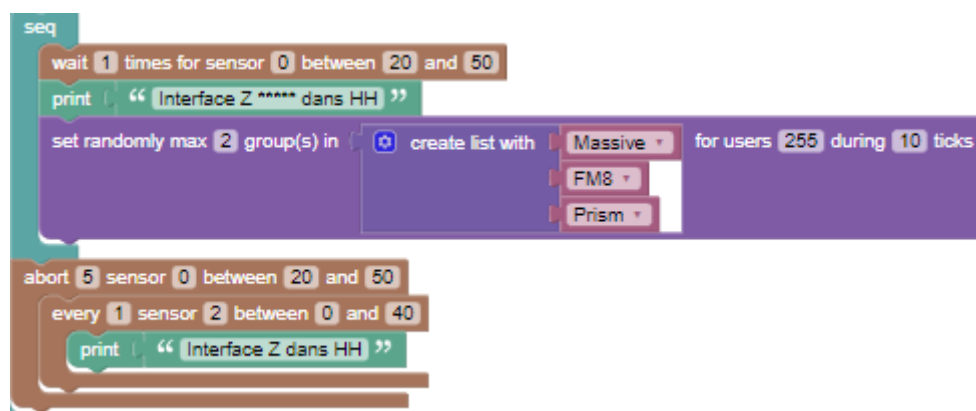
The sensitivity allows you to consider the permanent variations of the analog sensors. If the variation (in plus or minus) with the value considered at the previous time delay is less than the sensitivity, no signal is sent. Here again it is a question of not overloading Skini with irrelevant signal information.

Skini offers a series of blocks to specifically manage Interface Z sensor information. These blocks are dedicated to signals:




They have the same structure. *sensor* is the number of the sensor on the Z interface card. As the values are fluctuating, the signals are considered in a range between a minimum and a maximum.

Programming example:



Order to be reviewed, more operational (1/9/2022)

This command is used to send a MIDI command via the Z interface OSC. This is a very specific use case.

send midi via osc message  ch. 1 note 0 val. 0

8.16 PROGRAMMING OF STINGER TRANSITIONS

The principle is to associate a transition pattern to a group of patterns (or a pattern that is a singleton). In the world of video game audio, this is called a *stinger*. Ideally a *stinger* should be associated with a specific transition from a group of patterns A to a group of patterns B. The difficulty in Skini is that the automaton has no vision on how the FIFOs are filled. The automaton does not know how to identify the "temporal" sequence of two patterns in two different FIFOs.

In the *selection reaction* scenario, the automaton sees how the FIFOs fill up, but there is no way to manage the history of the FIFOs (if we had to create a kind of duplicate of the FIFOs in HH or imagine signals coming from the FIFOs to HH). This management could even be quite complicated if the patterns have variable durations.

In the *runtime reaction* scenario, one would only need a view of the patterns waiting in the FIFOs to locate the right successor to A. That is, a B in a waiting position of at least the duration of A in its FIFO.

In fact, Skini is designed to manage the processes with which the FIFOs fill up, but not their organization, except in the case of priority management in queues explained in chapter Priority in queues, p.37.

First, we will simplify the problem by dealing with the case where we want to associate one or more *stingers* to each pattern. In this case, it is enough to wait for the execution signal of a pattern of A and to launch a pattern S, or to launch a *stinger* which integrates the offset. It is therefore easy to make *stingers* "on exit".

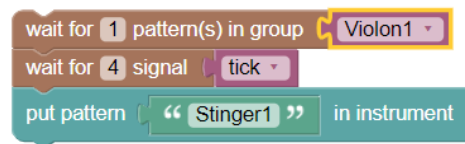
8.16.1 Case of the reaction to the execution

This scenario is quite simple to realize in the case of *runtime reactions*, i.e. when the pattern is launched in the DAW. To manage the time lag of S, a *tick* must be set which allows a time lag to be taken into account when the stinger is launched. Note that *tick* must therefore be at least the duration of this delay (cf. The programming of time in a room, p.23).

A *stinger* can be programmed on the scheme :

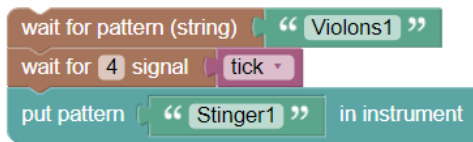
1. Wait for *patternAIN*
2. Wait for *X tick*
3. Put *pattern Stinger* in instrument!

An example for violin patterns of 8 ticks with a stinger starting on the 4th *tick*. Here the *tick* would be worth one beat.



8.16.2 Case of the pattern set return from the DAW

The logic is the same, but the stinger principle is dependent on a particular pattern. This case works even when the reaction is *to the selection* because the pattern signal is emitted by the DAW when the pattern is launched.



Remark: Rather than counting *ticks*, we can in the design of the *stinger* foresee an initial silence to shift the sound. There is no constraint on the duration of the patterns.

8.17 PRIORITY IN QUEUES

Caution: Do not use a FIFO modification algorithm in interactive music if the algorithm can remove patterns from the FIFO queues, this creates blocking situations on clients waiting for a return on the set of requested patterns.

It is possible to activate a queue treatment (FIFO). You have to update the configuration of the room:

Algo Fifo management at 1

(the value 1 assigned allows to consider that we can develop several types of algorithms and give them different identifiers).

For each pattern in the csv file a type must be defined for the pattern at index 7 of each line. We define five types of pattern. D: Start, M: Middle, F: End. N: Neutral (no treatment) and P: Bread (a bread = bad pattern). The type is declared by a numerical value in the pattern csv configuration file: 1 for D, 2 for M, 3 for end and 4 for neutral, 5 for "bread". The type P is used in interactive game contexts, where a player selects a pattern that doesn't sound in the room or in the pattern list of the skini client.)

To improve the structure of musical phrases we look at the state of a queue before adding a pattern. As we write in a FIFO by adding a last element and bind by removing the first one, we scan a FIFO starting from the end (last addition) to intervene on the last patterns put in the FIFO, the most recent ones. Here is the algorithm in place:

A) To add a pattern F in a queue :

1. If there are two Ds in a row, F is inserted between the two.
2. If there are two M's in a row, F is inserted between the two.
3. If the last element of the queue is already an F, we look for a D that is followed by an M, if we find one we put the F to be stacked just after this D.
4. Otherwise we stack F (which gives two F in a row)

We can therefore have sequences of F

B) To add a D pattern to a queue:

1. If there is an F in the queue without a D before it, D is inserted before this F.
2. If in the queue there is an M without D before it, D is inserted before this M.
3. Otherwise we stack D (which gives two D in a row)

We can therefore have sequences of D

C) To add a pattern M in a queue :

1. If in the line there is a D immediately followed by an F we put M between the two.
2. If there are 2 Ds in a row, we put M between them.
3. If in the queue there are 2 F's following each other, we put M between the two.
4. Otherwise we stack M (which gives two M in a row, but is not a problem).

If you don't want to have a sequence of D or F, don't do anything in cases 1.4 and B.3. This is possible in generative music, not in interactive music.

D) For a pattern N no treatment is done.

See in the file `./server/controlDAW.js` the function `ordonneFifo()` to have the details of the algorithm actually in place.

8.17.1 Example of programming a duel of solos (review for Node)

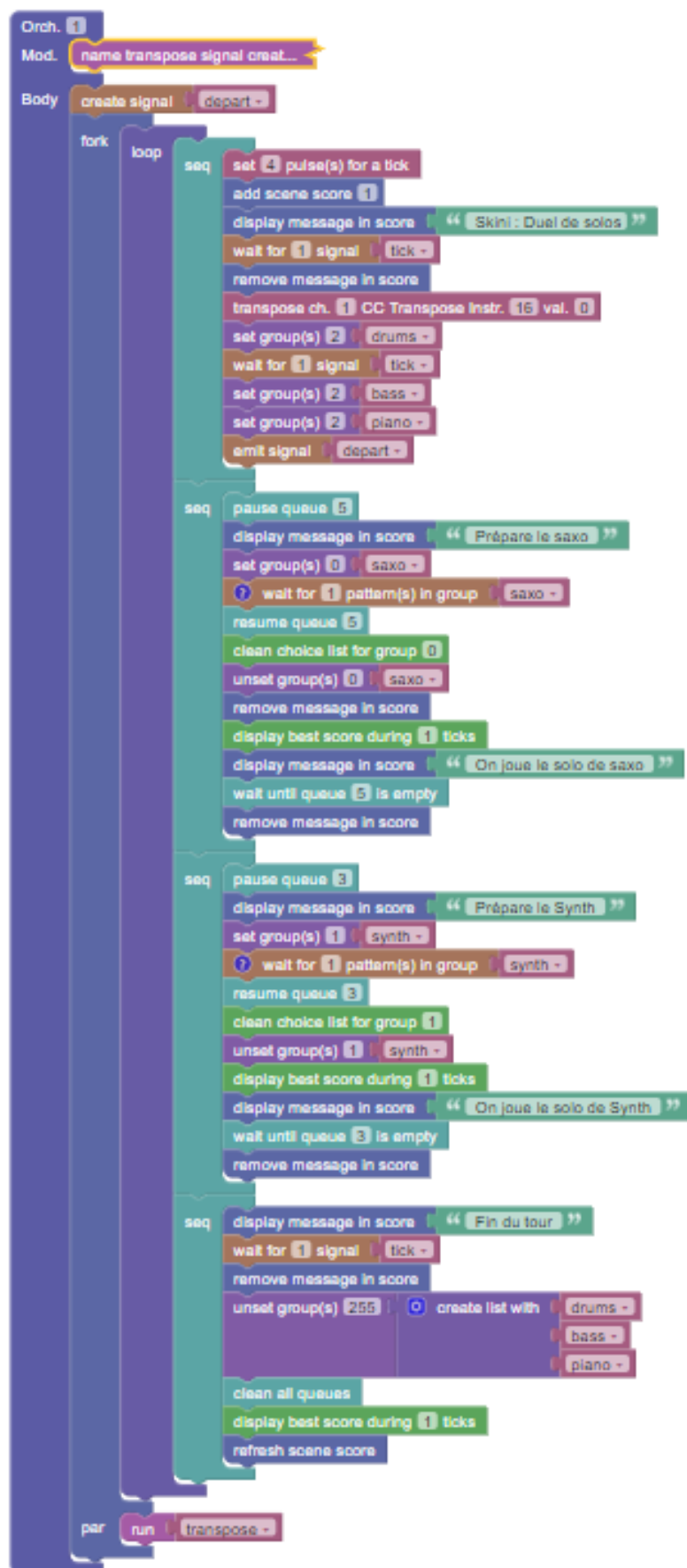
The patterns are used only in groups. The simulator is used on a dedicated group. This is described in the room configuration (funkBitwig) by specifying that the simulator is associated with the last group. Here we have groups 0 and 1 for the players and group 2 for the simulator.

"Number of client group" to 3

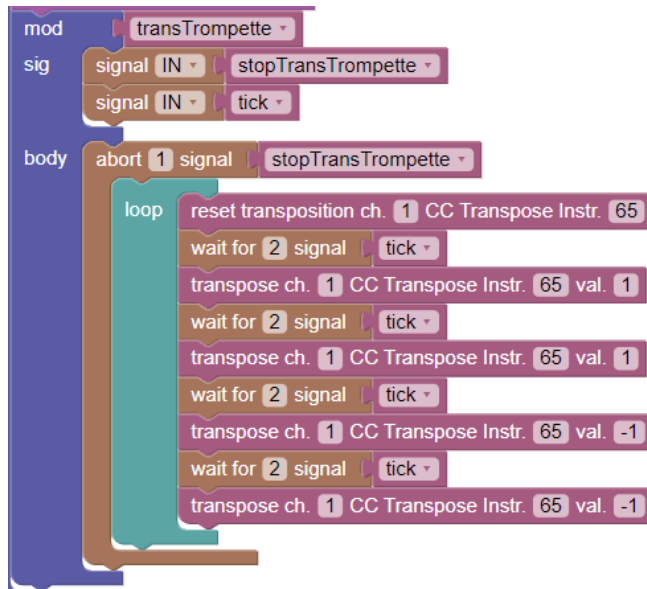
"Simulator in a separate Group" checked

The simulator is launched with the "-sim" option, it plays the rhythmic part independently of the sax and synth soloists.

All patterns have the same length of 4 beats, so it is more resource efficient to take a tick of 4 beats. This makes it easy to synchronize transpositions. The limit would be on tempo changes which are not on the agenda for this funky piece.



8.17.2 Example of transposition in loop

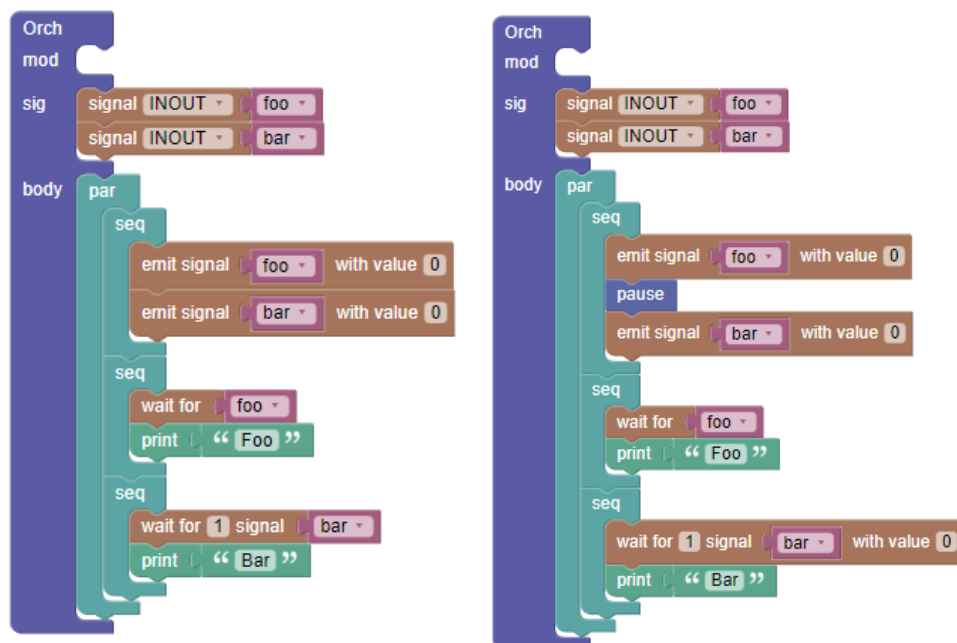


8.18 SOME SUBTLETIES OF SYNCHRONOUS REACTIVE PROGRAMMING

8.18.1 Immediate reaction

Synchronous reactive programming is intuitive because it works pretty much the same way we express. "I am doing this, waiting for that. If something happens then I will stop doing something...". Nevertheless, it's very precise semantics can reveal inconsistencies that are not very visible in our expression habits. This semantics at the initialization of a program makes the difference between a signal considered immediately or after an initial reaction.

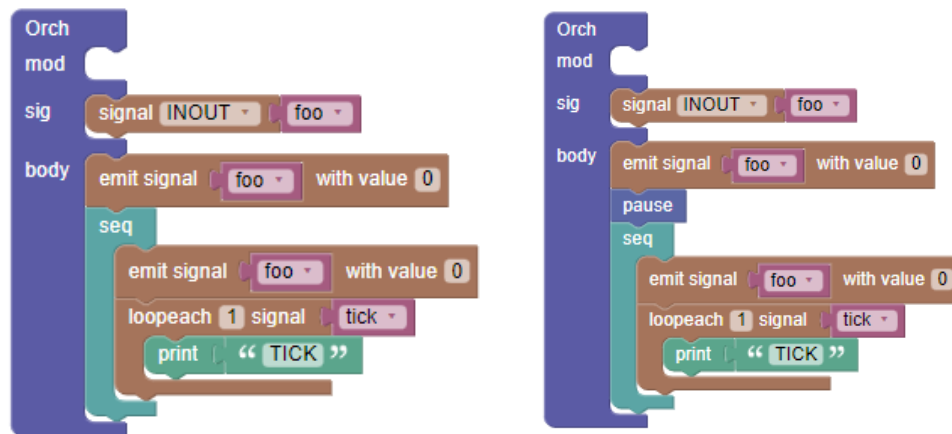
If you run the program on the left, you will see that only "Foo" appears on the console. There is a subtle difference between "wait for" and "wait for 1 signal". The "wait for" reacts immediately, "wait for 1 signal" needs an initial transmission and reaction to start. It is quite rare that in an orchestration this subtlety causes a problem.



In Skini there is only "wait for", "wait for pattern" that reacts to the first emission.

8.18.2 Double emission of the same signal

The program on the left will cause a problem at runtime. "The 'foo' signal will be emitted twice at the same reaction. This message will appear in the console: `TypeError: Can't set single signal 'foo' value more than once.`



The "pause" block in the right-hand program allows you to shift the second transmission by one reaction.

8.19 SPATIALIZATION BY TILING WITH THE PRÉ DEVICE

Here is a way to set up spatialization effects by tiling, i.e. patterns that partially overlap in time.

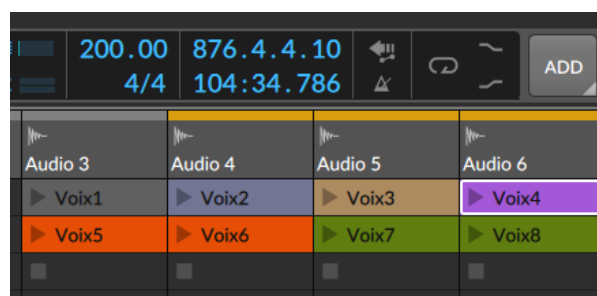
(Example space1.xml, space1.csv, space1.js).

The patterns here all have the same structure with a Fade IN and a Fade OUT to allow overlapping of the patterns. We will take a fast tempo to allow fast overlaps that we will not do here, but that will be possible.



The patterns have a duration of 44 beats with Fade IN and OR of 12 beats each.

Here are the patterns in bitwig Studio. The distribution of the patterns must only take into account the fact that a progress should not be random process is used. take into account when instruments and groups.



Here is an example of a example is given with a would be enough to assign Raspberry Pi to the Patterns.

pattern in interrupted if a This is a point to assigning

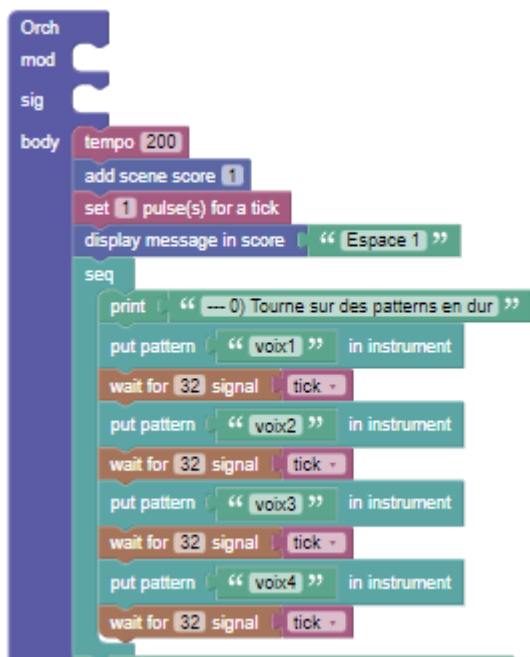
descriptor. The DAW, but it

	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration
1	11	10	0	voix1	voix1	0	0	0	0	0	44
2	12	10	0	voix2	voix2	1	0	0	0	1	44
3	13	10	0	voix3	voix3	2	0	0	0	2	44
4	14	10	0	voix4	voix4	3	0	0	0	3	44
5	15	10	0	voix5	voix5	4	0	0	0	4	44
6	16	10	0	voix6	voix6	4	0	0	0	4	44
7	17	10	0	voix7	voix7	5	0	0	0	5	44
8	18	10	0	voix8	voix8	5	0	0	0	5	44

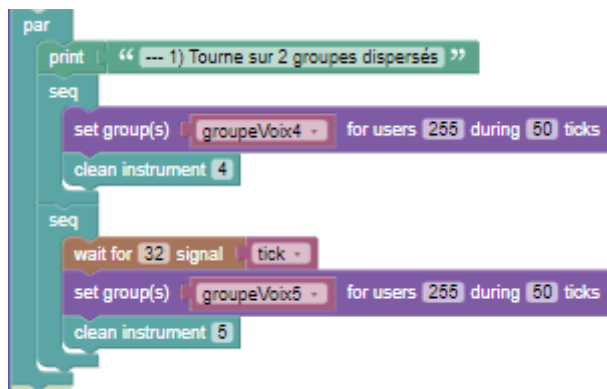
Here is an example of groups:

	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupeVoix0	0	group	200	100	20	#CF1919		1
2	groupeVoix1	1	group	400	100	20	#CF1919		1
3	groupeVoix2	2	group	600	100	20	#CF1919		1
4	groupeVoix3	3	group	800	100	20	#CF1919		1
5	groupeVoix4	4	group	200	300	20	#CF1919		1
6	groupeVoix5	5	group	400	300	20	#CF1919		1

For a "hard" control, i.e. fixed, without random phenomenon. You just must launch the patterns one after the other while waiting for the beginning of the Fade OUT at the 32nd beat.



To "run" two groups with a random selection in each group, simply shift the activation of the groups in parallel.



To run on several groups, you just have to shift them in parallel. The example below gives the same thing as the first "frozen" example because we have only one pattern in each group.



Remark : It is the notion of Skini instrument that makes it easy to realize spatialization by tiling. In an instrument the patterns are queued, so you just have to play on the instrument shifts to link patterns which are in different instruments.

9 WITH MUSICIANS (NOT IN PLACE WITH NODE.JS)

With musicians, Skini sets up a countdown before the first pattern is played in a queue.

In the configuration file of the room, to set up the specificities related to the play with musicians:

```
exports.withMusician = true;
```

This introduces an empty pattern before the first pattern in a FIFO.

WARNING: This offset is based on the *tick* (not the pulse) and must be a multiple of the *tick*, otherwise the player will freeze.

```
exports.decalageFIFOwithMusician = 8;
```

To specify the position of the pattern score files in jpg format, in the configuration file of the room we will have for example :

```
exports.patternScorePath1 = "";
```

These paths are relative to the ". /images" directory.

10 OSC INTERFACE

10.1 PRINCIPLE

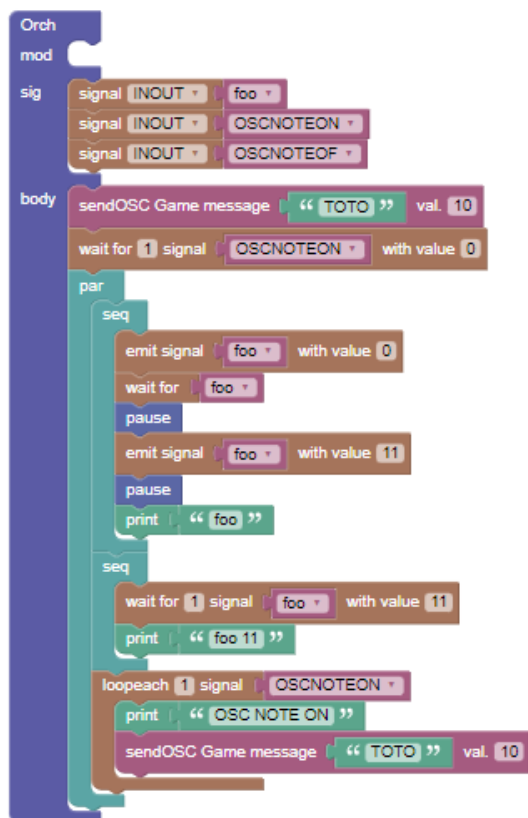
If a game development platform or from a controller can send OSC commands, it is possible to generate signals in the orchestration from external events and conversely to generate OSC messages from the orchestration.

10.2 SINCE THE ORCHESTRATION

It is necessary to create signals corresponding to the received OSC messages. The signals correspond to the messages without the "/" slashes. The message /FOO/BAR will give the signal FOOBAR in Skini. The transmitted message does not have a slash in the header, it is added by Skini.

Note on OSC signals: Signals created for OSC should not have the same names as default signals. It is prudent to define a common format for these signals by adding an *OSC* or *Game* suffix or prefix or whatever.

10.3 IMPLEMENTATION WITH ORCHESTRATION



Example of a program for receiving and transmitting OSC commands:

- from an OSC transmitter to *serverIPAddress* on the *portOSCFromGame*.
- to an OSC receiver in *remoteIPAddressGame* on the *portOSCToGame*.

11 OSC WITH BITWIG STUDIO

It is possible to make Skini communicate in OSC with Bitwig Studio, so without using Processing as an OSC/Midi gateway. The Bitwig controller, Skini_0.control.js, behaves like the Processing gateway for Ableton.

On Bitwig the Skini_0 controller has the same parameters as Processing, for example :

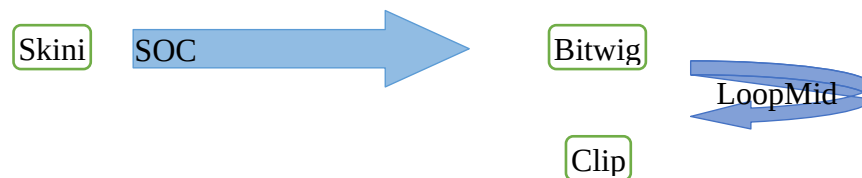
Remote OSC Ip address: 192.168.1.6

Remote OSC port out: 13000

Listening OSC: 12000

Midi information is sent to Bitwig as it is to Processing. So there is no change in Skini in terms of generating Midi commands for patterns or other commands. To keep compatibility with the Processing gateway, Bitwig sends the OSC commands for Midi to itself on a Midi channel in LoopMidi. (I don't know how to send a Midi command directly to the clip triggers). It is therefore necessary to declare a Generic Keyboard controller that receives midi commands from the Skini_0 controller.

So there is a Midi OUT port to give in the Skini_0 controller to LoopMidi, which will be the Midi input port in the generic keyboard.



The Bitwig controller can send Midi commands from one of its controllers to Skini. This is the port given in Midi OUT of the Skini_0 controller. The interpretation of the OSC commands from Midi to Skini can be found in server/midimix.js.

The mapping of OSC commands from Bitwig to Skini is defined in the Skini_0.control.js and midimix.js files.

Moreover, Bitwig emits a tick in OSC `"/BitwigTick"`. The calculation of this tick in the controller is not really canonical. It is based on the Bitwig transport bar.

Example: technoBitwig piece on ABL3.

Conclusion: The only changes to Skini for a first implementation of Bitwig are in midimixi.js and websocketServerSkini.js.

Note: In Bitwig studio there is no notion of bus (midi port). Bitwig receives messages in OSC and reroutes them for control via loopMidi. The bus parameters present in the OSC commands sent to bitwig are therefore not taken into account.

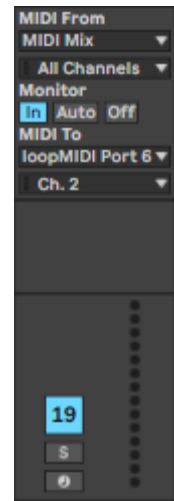
12 OSC WITH MAX4LIVE

It is possible to use Skini and Ableton without using the Processing gateway. For this we use Max for Live (M4L). The implementation consists in creating a Midi track in which the Max patch "SkiniOSC2.amxd" must be loaded. The track can receive a MIDI device as input whose commands will be sent to Skini in OSC.



12.1 CONVERSION TRACK

The Ableton Midi track must be routed as a Midi output to a control channel that will receive Skini's OSC commands converted into Midi. Indeed, it is not possible to talk directly to an Ableton Live control from M4L, you have to go through a virtual cable (LoopMIDI for example). The patch parameters are OSC. The Midi settings are done in the Ableton Live track.



12.2 MIDI ROUTING TRACK



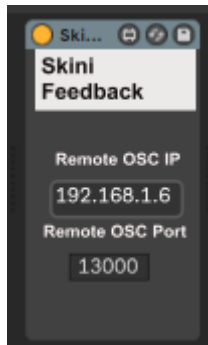
A bug in Ableton Live complicates the operations as soon as you have more than 127 notes associated to the patterns in Skini.

M4L in an Ableton track cannot address anything other than Midi channel number 1. To get around this problem you have to create additional tracks, as many as multiples of 127 among the notes associated with the patterns in the pattern configuration file (.csv).

You have to load the M4L patch "SendToMIDIChannel.amxd" in these additional Midi tracks and give the right channel on the track in "MIDI to". Skini channel 2 corresponds to Ableton channel 3. There is an offset of one unit between Skini and Ableton.



12.3 FEEDBACK TRACK



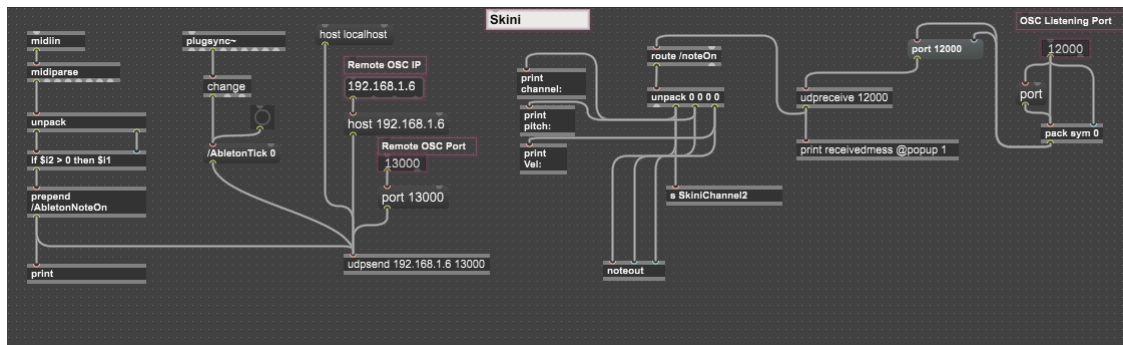
To process the Midi information sent by Ableton when a pattern is played (this function is only useful if you use the *patternSignal* signal in the orchestration), you need to add a Midi track with a *SkiniAbletonFeedback.amxd* M4L patch. This track has as input port (MIDI From) a Midi port that receives the "Remote Controls". For example :

The "remote controls" are sent to Skini in OSC. Here again we are limited to 127 Skini notes. To overcome this problem, it is easier to use Processing, given the level of complexity of parameterization reached in Ableton.

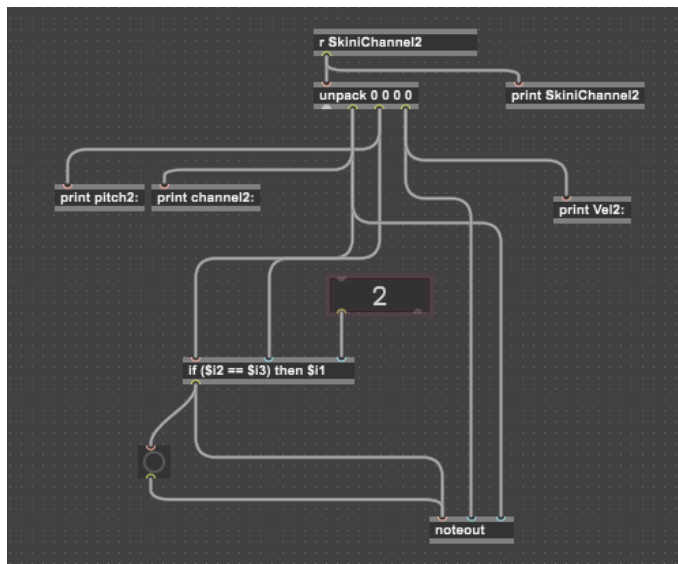


12.4 M4L PATCHES

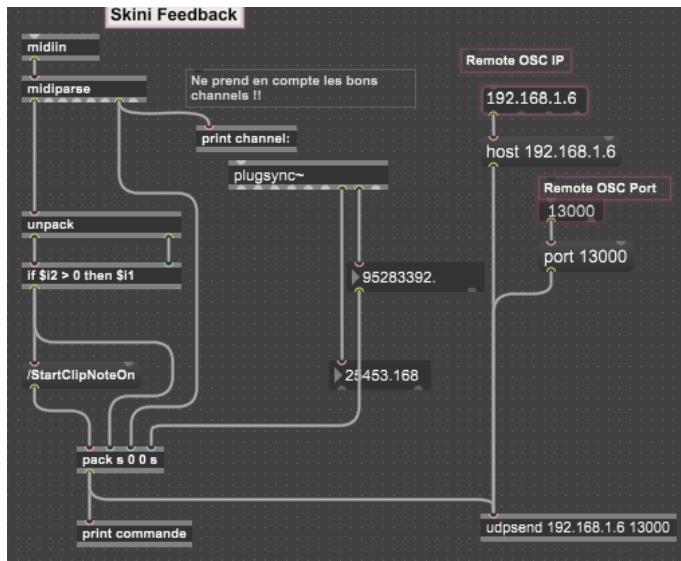
Here is the *SkiniOSC2.amxd* patch:



The *SendToMIDIChannel.amxd* patch:



The *SkiniAbletonFeedback.amxd* patch:



12.5 CONCLUSION OSC SKINI WITH ABLETON

Using Ableton Live with Skini is possible without Processing, it becomes complex as soon as you have more than 127 notes of Patterns in Skini because of a design problem in M4L. This is another demonstration of the lack of maturity of M4L.

As an example, Ableton or the *technoDemoOSC.als* piece.

Remarks:

1) The assignment of MIDI commands to clips in Ableton processes the channels starting from 0. Skini converts the Midi channels from 1. So we see that Skini's commands actually start on channel 2 of Ableton's Midi control port. If you process less than 127 Skini notes, you can use only one *Midi routing* track by assigning *channel 2* to the control Midi port.

2) The *SkiniOSC2.amxd noteout* does not take into account the Midi channel given to it but only the Midi channel at the output of the track. The way out of this is to use Max/Msp's *send/receive*. The *SkiniOSC2.amxd* patch sends the Midi command to the *SendToMIDIchannel.amxd* patch which compares the received channel with its parameter. This patch loses Midi notes for no reason, *send/receive* doesn't work sometimes or *noteout* doesn't do anything and there is no *try* and *catch* mechanism to know why.

13 PATTERN CONTROL ON RASPBERRY

To use Skini to control the Raspberry project "Pré" (by Jean-Luc Hervé) with OSC, there are 3 parameters to modify in the configuration of the room:

"Use Raspberries", checked;
Play Buffer Message" with "test" checked;
"Raspberry OSC Port" set to 4000;

The *Use Raspberries* parameter allows you to disable playing on the Raspberry globally and play the patterns on the DAW. *playBufferMessage* is the OSC message (without /) that the Raspberry understands to play a sound file. *raspOSCPort* is the UDP port used by OSC for the Raspberries

We extend the pattern descriptor with three pieces of information:

- The IP address of the Raspberry that should play the pattern
- The number of the sound in the Raspberry corresponding to the pattern (field " Buffer num ").
- The sound level of the pattern (0 to 128)

If the "buffer num" field is not filled in, the pattern is considered to be played by the DAW.

It is thus possible to play patterns in the same room with the DAW and the Raspberries. It is also possible to test a pattern on the DAW before playing it with a Raspberry.

NB for developer: The *queue* stores the IP address of the Raspberry at index 10, the buffer number at index 11 and the level at index 12. In the pattern *configuration* the Raspberry IP address is in index 11, the buffer in index 12 and the level in index 13.

14 ANNEXES

14.1 SKINI WITH ABLETON LIVE

We have often used Ableton Live as a DAW for our developments. Any DAW that can associate a MIDI command to a clip without synchronization constraints is generally suitable. Ableton Live offers the possibility of associating MIDI commands to a large number of parameters, tempo, MAX/MSP, recording commands etc. Its use with Skini is therefore very rich and easy to implement with the Skini configurator.

For our development the patterns were designed in Ableton, mostly in MIDI format. Ableton allows the conversion of MIDI clips into sounds if needed, but then we lose the MIDI processing like transpositions, mode conversions etc.

For our pieces we do not use global quantization in Ableton. The synchronization of the clips is done by Skini.

As presented in this documentation, the configuration of the MIDI ports does not present any difficulties, we will pay attention to the IN port used for Skini commands to Ableton which must

authorize Remote Controls ("Télec.") and the OUT port used by Ableton to send MIDI messages to Skini (Processing gateway) which must also authorize Remote Controls. The configuration of the ports is done between lines 214 to 256 of Processing's " Skini sequencer " and Ableton's Preferences/MIDI.

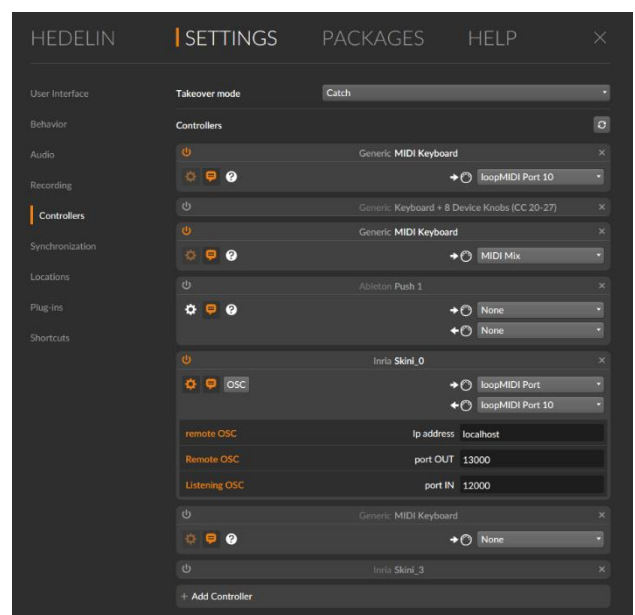
You need to enable Ableton's MIDI sync on the OUT port corresponding to the Processing gateway's IN port for Ableton's tempo to control Skini.

In the Windows version with LoopMIDI, port 12 is used by Ableton to inform Skini of the launched clip (port 13 is used to pass MIDI controls from the video in Reaper). These ports are "hard wired" in the *OSCMidi* tab of the Processing gateway.

14.2 SKINI WITH BITWIG STUDIO

With Bitwig studio it is also possible to communicate in OSC and thus to do without the Processing gateway. A Bitwig controller has been developed for this purpose (`./bitwig/Skini_0/Skini_0.control.js`). It receives Skini commands in OSC form and in order not to change the pattern declaration mode, Bitwig Studio reroutes these OSC commands to a virtual MIDI interface (on windows LoopMidi for example) which sends back the received commands in MIDI.

The figure on the right shows an example of a Bitwig Studio configuration.



14.3 EXAMPLES OF ORCHESTRATIONS

findThePercuNode.xml : is an example of a game where the audience has to find the correspondence between patterns and a sound ambiance. The corresponding Bitwig studio session is worldMusicGame. The example in Bitwig studio uses VST from Native Instruments

opus5.xml: is an example that uses mostly reservoirs with MIDI transposition commands. The example in Ableton uses VST from Native Instruments. The Live set is opus5.

14.4 ORGANIZATION OF THE FILE SYSTEM

From the directory where Skini is installed we have the directories :

client : The different clients of skini are located in subdirectories.

docs : follow-up of the thesis and the Skini project

images: in subdirectories by pieces we have here the partitions of the patterns as jpg files.

pieces : In this directory we have the configuration files of the pieces and the csv files of patterns definition

Processing : contains Processing programs, *sequencerSkini* is the one commonly used.

sequencesSkini : contains the patterns saved by the distributed sequencer.

server : contains the Hop.js and HipHop.js files of Skini and the general configuration.

sounds: the mp3 sound files of the patterns are organized in subdirectories.

blockly_hop : which contains blockly programs and orchestrations in xml format

bitwig : contains extensions for Bitwig Studio

14.5 SCORE RECORDING IN FINALE

You cannot extract MIDI directly from Ableton if you are afraid of losing the MIDI transposition commands sent directly by Processing. If there are no pattern modifications in Live, you can directly complete the MIDI tracks in Live and export them to Finale. If you want to use pattern modifications, you should apply the procedures below.

The prerequisite in this process is that there are no tempo changes in the Skini session. These changes are to be made in the final score. Tempo changes are not important because this is not a sound recording but a written transcription.

Think of providing an option in the automatons to deactivate the changes of tempos.

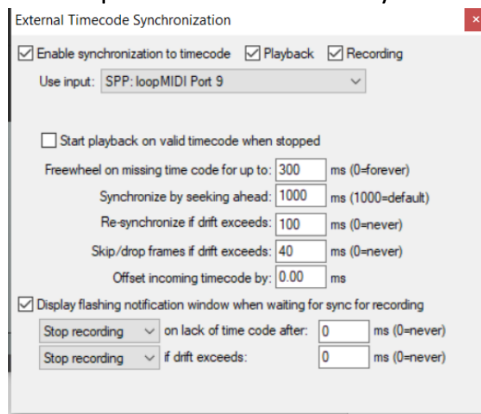
Be careful also with the duration of the patterns. Here you must necessarily activate the global quantization of Ableton Live to avoid MIDI micro-shifts that are impossible to process. Setting the global quantization is the same as setting the pattern length in Skini.

With these precautions the transcription from Skini to Finale takes a few minutes.

14.5.1 From an Ableton recording

- 1) Save the original Ableton Live file
- 2) Copy it to another file to prepare the MIDI conversion
- 3) Replace the VST of each track with a MIDI OUT channel.
Simply delete the VST to allow MIDI output on a Track.
Pay attention to the output port used for the MIDI sync.
- 4) Remove tempo variations (if any)
- 5) Configure the MIDI channels in Reaper (port 9 is usually used for MIDI sync).

6) Put Reaper in Slave with MIDI sync



- 7) Set Ableton to MIDI Sync transmission (check Ableton and Reaper sync ports)
- 8) Record in Reaper by setting the same tempo as Ableton Live. Be careful to have Ableton Live with a global quantization.
- 9) In Reaper format the MIDI tracks, remove the blank at the start of the recording, quantize the MIDI...
- 10) Export each MIDI track from Reaper independently
- 11) Load MIDI tracks recorded by Reaper into Finale
- 12) Format in Finale.

12.a.i. Directly from Skini

The settings and constraints on tempo variations are the same as above. You can directly record the progress of a Skini session in Reaper without recording it in Live.

Be careful to have a global quantization in Ableton Live to avoid small MIDI shifts or drifts from Skini. It should be possible to adjust the delay on the MIDI sync in Live to be perfectly in phase between Live and Reaper.

14.6 SWITCH FROM FINALE TO MICROSOFT WORD OR POWERPOINT

- Export the score in PDF format in Finale
- Load the PDF into INKSCAPE via import, taking the options that include fonts (not the default option)
- In INKSCAPE save in simple SVG format.
- Drag SVG to Microsoft

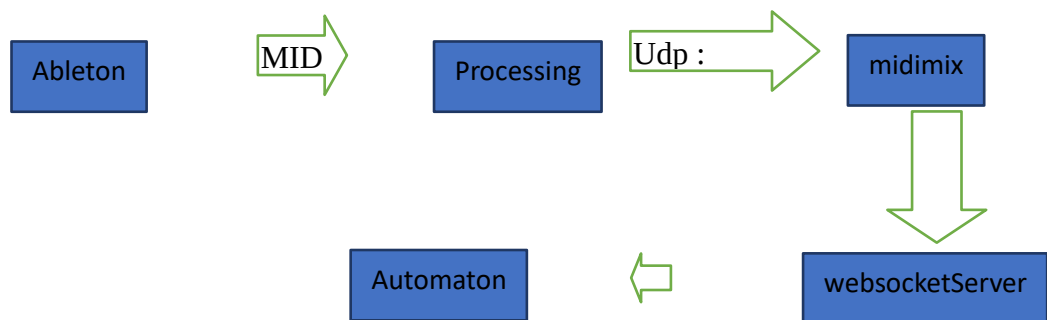
○ RECEIVE MIDI CLIP LAUNCH INFO FROM ABLETON

- You need IP addresses in IPconfig, not localhost.
- Set an unused MIDI out channel to "Telec" (remote). This means that there is a control surface on the port (in opus1 and 2 it is the loopMIDI Port out 12).
- The correspondence must be done in Processing (sequencerSkini.pde) in the table "myBusIn" and in the tests of origin of the MIDI commands (OSCMidi). Processing translates the commands coming from the MIDI port into OSC "StartClip" commands.
- Ableton sends the MIDI commands for launching clips to this channel.

Remark : In " track " mode on a MIDI out channel Ableton sends the MIDI notes they process but not the CCs. If you want to retrieve CCs, they would have to come from the clips

The MIDI commands sent by Ableton follow a weird protocol that is processed by `midimix.js` which is the program that processes OSC data from Processing (the name is not great, it goes back to the Golem).

`Midimix` is created in `golem.js`. It passes the information to `websocketServer`, it is `websocketServer` which informs the `automaton`. We pass by `websocketServer` only to reach the `automaton`.



14.7 RECORDING OF LIVE PATTERNS IN THE DISTRIBUTED SEQUENCER

(to do, give the structure of the programs.).

15 AN EXAMPLE OF A PIECE: OPUS5

This piece uses most of the basic principles of Skini.

