Skini and Node.JS

Instructions for use and implementation

V 1.0

B. Petit, 9/2021

bertrand@hedelin.fr

Table des matières

1 INTRODUCTION

Skini is a collaborative and generative music composition platform. This document[1] 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 2021 version. To understand the functioning of Skini, it is advisable to refer to the different articles published on this platform (Programming journal 2020, ICMC 2021, NIME 2019). If you can read French the thesis « *Temps et Durée : de la programmation synchrone à la composition musicale* » is the most complete document describing the Skini approach.

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 in the form of groups. 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, measuring the duration, listening to MIDI or OSC information.

The orchestration can be seen as a "super sequencer" bringing interaction functions based on queues and bringing complex automatisms to commercial DAWs. The orchestration can also

[1] This platform is developed by a French speaker, so don't be afraid if you find English mistakes in this document. You will notice that several elements may use French terminology, they are not mistakes 😊. But It should not cause you to much troubles because remember that most of the technical English terms came from French… If you are a Latin language speaker you should globally feel at home.

control the queuing of patterns as the audience would. It is possible to issue MIDI, note or "control changes" commands directly from the orchestration.

The orchestration is written in HipHop.js language using graphical programming for this Node.js version. A basic knowledge of synchronous programming is better to start designing a first orchestration. With a good mastery of Skini it will be possible to produce rich and inconceivable pieces otherwise.

1.4 Using a Digital Audio Workstation

In this document we will only deal with *Ableton Live* and *Bitwig Studio*. The use of another DAW will be based on the same principles as those proposed for one or this DAWs.

The composer must create patterns (clips). To each of these patterns the composer will associate a Skini note in a pattern configuration file. 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, 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. It is therefore necessary that the dimensions of the groups are compatible with the possible display on an interface for the audience. The composer must also find a good balance between short patterns that will make the interaction dynamic 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 have been issued, you still must create the configuration files as described in chapter 2 on configurations, and then move on to orchestration.

1.5 Playing with musicians

In the version 1.0 of Skini on Node.js this function is not yet implemented but Skini can very soon be used to dialogue with musicians 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 sub-directory 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 CONFIGURATION

The directories in this document 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.

Note: The only package which depends on the operating system is “midi”. The default one included in the source is for Windows. You can install the good one for MacOs or Linux with npm.

2.2 INSTALLATION OF PROCESSING FOR 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 by putting the good parameter in the configuration file of the piece of music.

If the communication with the DAW is done in MIDI and you use a computer that hosts the Node.js server, and another computer that 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 must install the libraries oscP5, TheMidiBus, and WebSockets (use menu of Processing: add a tool -> libraries).

2.3 NETWORK CONFIGURATION

2.3.1 IP Configuration

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

Example of ipConfig.json :

```
{
        "remoteIPAddressImage": "192.168.82.96",
        "remoteIPAddressSound": "localhost",
        "remoteIPAddressLumiere": "192.168.82.96",
        "remoteIPAddressGame": "192.168.82.96",
        "serverIPAddress": "localhost",
        "webserveurPort": 8080,
        "websocketServeurPort": 8383,
        "InPortOSCMIDIfromDAW": 13000,
        "OutPortOSCMIDItoDAW": 12000,
        "distribSequencerPort": 8888,
        "outportProcessing": 10000,
        "outportLumiere": 7700,
        "inportLumiere": 9000
}
```

`serverIPAddress:` Skini server address.

`remoteIPAddressLumiere`: for a specific usage with QLC using OSC

`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 specific to the GOLEM show.

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

For standard use, it is sufficient to update `remoteIPAddressAbleton` et `serverIPAddress`.

2.4 MIDI Configuration for Node.js

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:

```
[
  {
    "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 controler"
  }
]
```

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

The "spec" field defines the usage with:

- - “clipToDaw” defines the port that allows the DAW to receive MIDI commands from Skini.
- “syncFromDAW" defines the port that will receive MIDI synchronization from the DAW.

- “controler" corresponds to a port with an associated MIDI controller (keyboard, PAD …).

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.

2.5 Configuration of the musical pieces

The command to launch Skini is of the type*: node skini. <piece description >*

Ex : node skini ./pieces/opus1.js

We will now review the main parameters of the piece description file. The configuration files for the explanations are in the directory *./pieces* but you can put them where you want.

For the description of the patterns we have the *configClips* parameter. Here is an example:

```
exports.configClips = "pieces/trouveLaPercu.csv";
```

Which defines the pattern configuration file *‘trouveLaPercu.csv’*. We will see later (chapter: Patterns configuration) the creation of this file of descriptors.

The parameter `directMidiON` of line :

```
exports.directMidiON = true;
```

true means that the communication between Skini and the DAW is done via MIDI. false means that 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.

The `reactOnPlay` parameter defines how to make the orchestration automaton react. By default, it is on selection. With `reactOnPlay=true` it is when the pattern is played. This has an important impact on the way the automaton is thought. *Stingers* are only possible with `reactOnPlay=true`. Ex :

exports.reactOnPlay = false;

The parameter `soundFilesPath1` of line

```
exports.soundFilesPath1 = "opus1";
```

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

The parameter :

```
exports.nbeDeGroupesClients = 2;
```

Sets the number of groups of people in the audience that the orchestration can handle. With the following parameter, `simulatorInAseperateGroup` set to `true`.

```
exports.nbeDeGroupesClients = 2;
exports.simulatorInAseperateGroup = true;
```

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 higher than 2, because the last group will be the simulator's one.

If the simulatorInAseperateGroup parameter is missing, it means that there is no group dedicated to the simulator.

The parameter algoGestionFifo allows to activate processing algorithms on the queues (cf. chapter Instruments queuing p.22).

```
exports.algoGestionFifo = 0;
```

The naming of the pattern groups is done via an array. The names of the pattern groups are used to create the HipHop signals of the orchestration. Here is an example:

```
exports.groupesDesSons= [
  // Pour group: nom du groupe (0), index du groupe (1), type (2), x(3), y(4), nbe d'éléments(5), color(6), prédécesseurs(7), n° de scène graphique
  ["groupe1",   0, "group", 170, 100, 20, rouge, [], 1 ],  //0 index d'objet graphique
  ["groupe2",   1, "group", 20, 240, 20, bleu, [], 1 ],     //1
  ["groupe3",   2, "group", 170, 580, 20, vert, [], 1 ],  //2
  ["groupe4",   3, "group", 350, 100,  20, gris, [], 1 ],   //3
  ["groupe5",   4, "group", 20, 380, 20, violet, [], 1 ], //4
  ["groupe6",   5, "group", 350,580, 20, terre, [], 1 ],  //5
  ["groupe7",   6, "group", 540,100, 20, rose, [], 1 ],   //6
  ["derwish",   7, "group", 740,480, 20, rose, [], 1 ],
  ["gaszi",     8, "group", 540,580, 20, rose, [], 1 ],
  ["djembe",    9, "group", 740,200, 20, rose, [], 1 ],
  ["piano",    10,"group", 740,340, 20, rose, [], 1 ]
];
```

The parameter : `exports.synchoOnMidiClock = true;` With the value true it means that the synchronization is provided by the DAW.

The presence of musicians must be specified with the lines:

```
exports.avecMusicien = true;
exports.decalageFIFOavecMusicien = 4;
exports.patternScorePath1 = "";
```

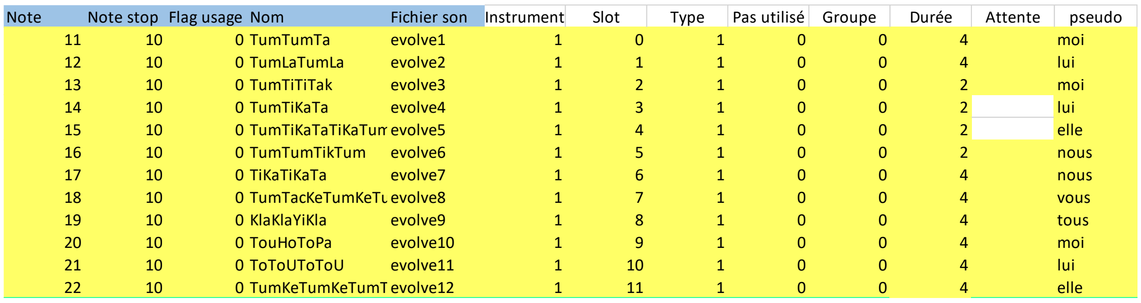
`decalageFIFOavecMusicien` gives the pulse count before the first pattern is played. Indeed, a musician needs to prepare himself before playing a pattern, unlike a DAW. This parameter introduces a systematic delay if there is no pattern in the queue for the instrument concerned. If there are patterns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

`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 Patterns Configuration

It is done with the help of csv files that are located in the directory *"./pieces"*. These are the files that were declared in the configuration of the piece in JavaScript. It is easy to edit this file in a spreadsheet and export it in csv format (utf8 if you want to use accents).



Note	Note stop	Flag usage	Nom	Fichier son	Instrument	Slot	Type	Pas utilisé	Groupe	Durée	Attente	pseudo
11	10	0	TumTumTa	evolve1	1	0	1	0	0	4		moi
12	10	0	TumLaTumLa	evolve2	1	1	1	0	0	4		lui
13	10	0	TumTiTiTak	evolve3	1	2	1	0	0	2		moi
14	10	0	TumTiKaTa	evolve4	1	3	1	0	0	2		lui
15	10	0	TumTiKaTaTiKaTum	evolve5	1	4	1	0	0	2		elle
16	10	0	TumTumTikTum	evolve6	1	5	1	0	0	2		nous
17	10	0	TiKaTiKaTa	evolve7	1	6	1	0	0	4		nous
18	10	0	TumTacKeTumKeTu	evolve8	1	7	1	0	0	4		vous
19	10	0	KlaKlaYiKla	evolve9	1	8	1	0	0	4		tous
20	10	0	TouHoToPa	evolve10	1	9	1	0	0	4		moi
21	10	0	ToToUToToU	evolve11	1	10	1	0	0	4		lui
22	10	0	TumKeTumKeTumT	evolve12	1	11	1	0	0	4		elle

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:

```javascript
var channel = Math.floor(note / 127) + 1;
note = note % 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 works because Ableton Live (and no doubt other DAWs as well) allows MIDI commands to be associated with patterns without any constraint on the channels. This method allows us to get rid of tedious channel management.

The *Note Stop* column, is the MIDI note 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 with 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 the patterns to a specific instrument which corresponds to a MIDI instrument or a musician. There is no correspondence between these numbers and a MIDI configuration.

The *slot* column relates to ongoing developments in the recording of patterns in Live. 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 be able to reorganize the queues; the description of this functionality is in the chapter *8.13 Priority in queues*.

The *group* column is in correspondence with the indexes of the groups described in the JavaScript configuration file in the table *groupsSounds*. It is this parameter that associates the pattern to a group and allows the patterns to be made available via the orchestration.

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

2.7 CONFIGURATION FOR USING 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 with the url :

http://*'ip of the server:8080'*/score

The parameterization of this display is done in the configuration file of the piece. Below we will see an excerpt of the configuration of a graph 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 bit different for groups and tanks. In brackets it is the index of the table of the line.

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

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

To define the graphic, it is advised to draw the score with a draw tool (Libreoffice for example) and to transfer the coordinates of the rectangles in this file.

The configuration of the reservoirs is more complex than that of the groups. A reservoir 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, for example line 18 to 21.

Care must be taken with the numbering of the predecessors. A group or a tank can be used as a predecessor. In the example configuration file we have added at the end of the line the number of the group as a predecessor. Up to line 18 everything is simple. On line 18 we start a tank that ends on line 21. These 4 lines are in the same "predecessor" of value 11. So in line 22 begins the "predecessor" of value 12.

```
1.  exports.groupesDesSons = [
2.
3.    // Pour group: nom du groupe (0), index (1), type (2), x(3), y(4), nbe d'élé
    ments(5), color(6), prédécesseurs(7), n° de scène graphique(8)
4.    // Pour tank:  nom du groupe(0), index(1), type(2), x(3), y(4), numéro du ta
    nk(5), color(6), prédécesseurs(7), n° de scène graphique(8)
5.    ["violonsEchelle", 0,  "group",  323,  176, 12, ocre, [3,8], 1],  //0
6.    ["violonsChrom",1, "group",  800,  180, 16, ocre, [9], 2],  //1
7.    ["violonsTonal",2,  "group",  450,  25, 24, ocre, [30], 3],  //2
8.    ["altosEchelle", 3,  "group",  200,  114, 12, violet, [5], 1],    //3
9.    ["altosChrom", 4,  "group",  800,  270, 16, violet, [9], 2], //4
10.   ["cellosEchelle", 5,  "group",  52,  173, 10, vert, [], 1], //5
11.   ["cellosChrom", 6, "group",  800,  360, 16, vert, [9],2],  //6
```

```
    ["cellosTonal", 7,  "group",  650,  100, 8, vert, [2], 3],   //7
    ["ctrebassesEchelle", 8,  "group",  200,  244, 12, bleu, [5], 1], //8
    ["ctrebassesChrom", 9,  "group",  590,  430, 16, bleu, [25, 29], 2], //9
    ["ctrebassesTonal", 10,  "group",  650,  160, 6, bleu, [2], 3],  //10
    ["trompettesEchelle1", 11, "tank",  273,  374, 1, orange, [5,0], 1], //11
    ["trompettesEchelle2", 12,  "tank",  200,  10, 1, orange, [],],
    ["trompettesEchelle3", 13,  "tank",  200,  10, 1, orange, [],],
    ["trompettesEchelle4", 14,  "tank",  200,  10, 1, orange, [],],
    ["trompettesTonal1", 15,  "tank",  650,  280, 2, orange, [2], 3], //12
    ["trompettesTonal2", 16,  "tank",  200,  100, 2, orange, []],
    ["trompettesTonal3", 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).
- Possibly Processing from MIT which will bridge the gap between OSC and MIDI.

To launch Skini you need to run:

- >node skini <piece descriptor full path>

With electronics you must then:

1. If you are using the OSC/MIDI gateway, start Processing with the program *sequencerSkini.pde*. The Processing console will indicate that the gateway has been connected to the server. 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 controller in a web browser with http://*IP of the server :8080*/contr. Skini is ready for performance.

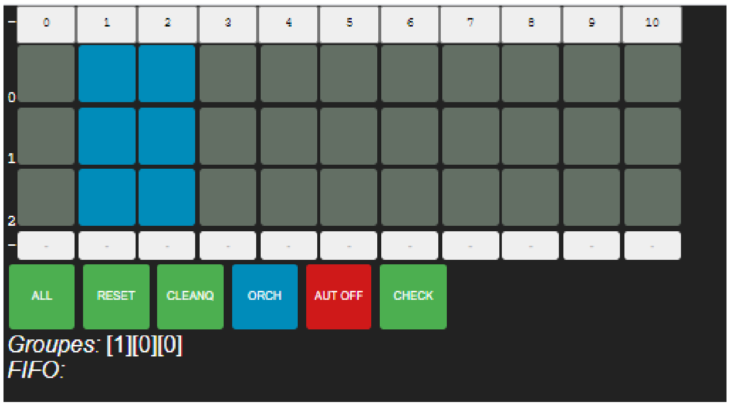
4 THE CONTROLLER

The controller is the "master of ceremonies" interface of a Skini performance. 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 piece and as columns the groups of patterns. 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 piece configuration file. 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 " LOAD " button loads the orchestration currently being edited. The orchestration is launched with " ORCH ON ".

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



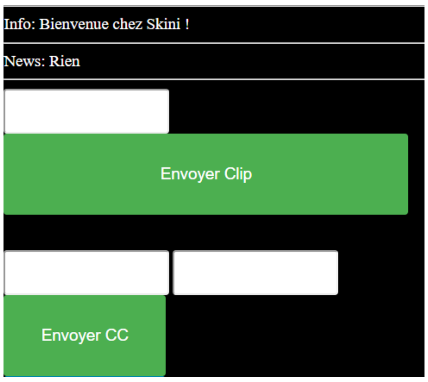
Groupes: [1][0][0]
FIFO:

The Group and FIFO displays provide system status.

CAUTION : There are two levels of indication of a piece in progress. One level in the configuration file and one level at the launch of the piece (node skini <the piece>). If the orchestration defined in the configuration file does not correspond to the launch, at the time of loading (ORCH ON) we will get an error message. No problem, just launch the Blockly editor to load the right piece, the one that matches the Node.js command, and save it. We can then click " ORCH ON " without error. If an error persists, you just have to restart Skini.

5 MIDI CONFIGURATOR

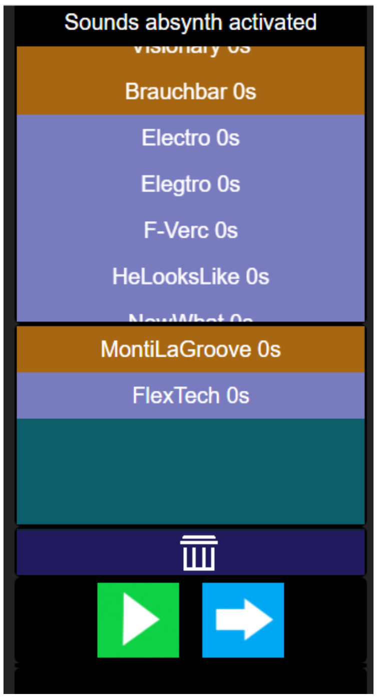
This tool is used to set up the DAW. Usually, DAWs have a function to listen to MIDI commands from MIDI controllers. To facilitate the implementation of the piece, Skini can send commands like a MIDI controller. Skini converts the "Clip" commands into a MIDI channel and note. The Configurator allows you to send notes with "Envoyer Clip" and MIDI "Control Changes (CC)" with "Envoyer CC". The first field above "Envoyer CC" is for the CC, and the field to the right of it allows you to give a value to the CC. The configurator is accessed with the address *http://IP of server:8080/conf*.


Info: Bienvenue chez Skini !

News: Rien

6 STANDARD 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 of choices.



Note: The patterns in the list of choices will fill the server's queues. If in a list we have patterns for different instruments, the queues will be fed. So 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.

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

This client coupled with an adequate orchestration allows to create musical games. For example, one can imagine letting groups of clients take turns to design lists of patterns during a period and to rate 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 in programming must 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 may 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 the file *websocketserver.js*, in the *computeScore()* function in file *./serveur/computeScore.js.* The scoring rules are to be modified in the source code. The scoring mechanism is based on *pseudos*.

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

7 SIMULATION

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 configuration file of the piece with the commands:

```
exports.nbeDeGroupesClients = 2;
exports.simulatorInAseperateGroup = false;
```

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 parameter *simulatorInAseperateGroup* when set to true, means that the last client group is reserved for the simulator. The audience will not have access to it. When set to false it means that the simulator will be able to behave like any group of the audience.

The hearing simulator is launched with the command:

```
./client/simulateurListe/node simulateurListe.js
```

The simulator outside the audience on the last "group of people" when *simulatorInAseperateGroup* is true is launched with the command:

```
./client/simulateurListe/node simulateurListe.js -sim
```

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

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

```
var tempoMax =  500; // En ms
var tempoMin = 100; // En ms
var limiteDureeAttente = 30; // En seconde
```

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

tempoInstant = Math.floor((Math.random() * (tempoMax - tempoMin)) + tempoMin);

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

limitWaitingTime is the parameter that defines the waiting time of a pattern beyond which the simulator will not call the server.

The simulator includes a mechanism to avoid the repetition of the same pattern in a history of 3 previous patterns. It is the function *selectRandomInList*.

8 PROGRAMMING ORCHESTRATIONS

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.

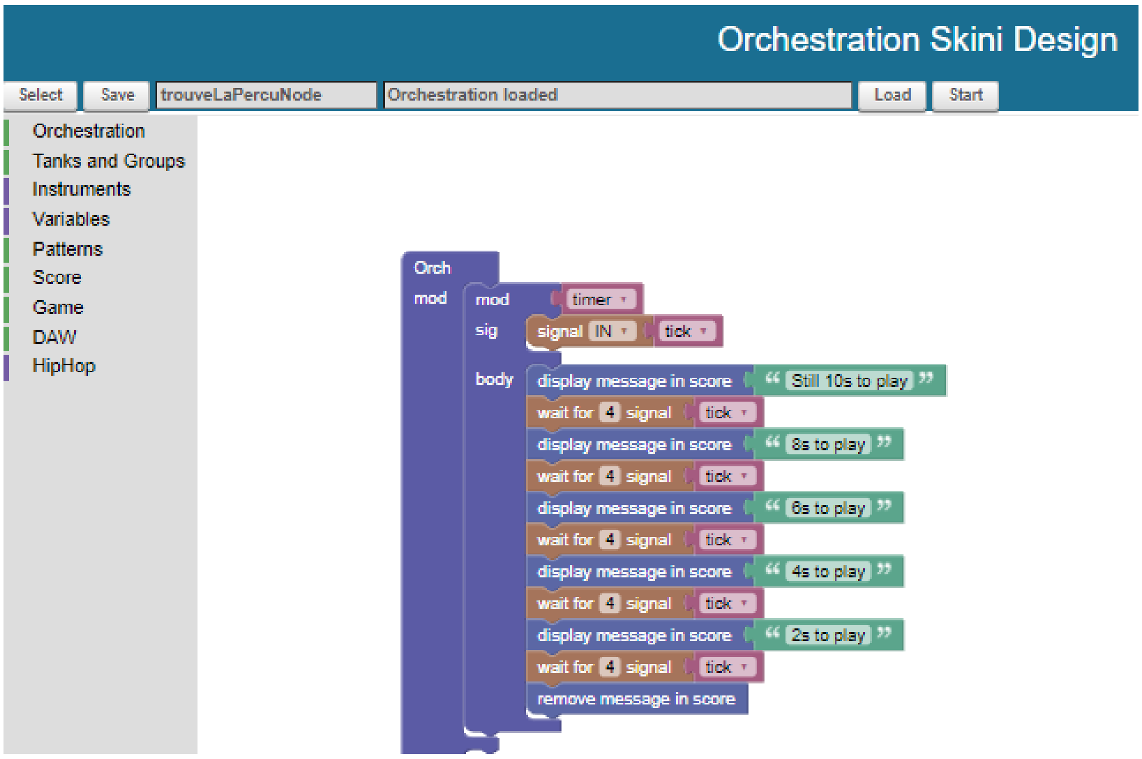
We don't talk about how Blockly works here. A visit to the site:

https://developers.google.com/blockly

will be more complete and effective than a general presentation for Skini.

8.1 GRAPHICAL INTERFACE OF SKINI

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



```
Orch
mod   mod   timer
      sig   signal IN  tick
      body  display message in score "Still 10s to play"
            wait for 4 signal tick
            display message in score "8s to play"
            wait for 4 signal tick
            display message in score "6s to play"
            wait for 4 signal tick
            display message in score "4s to play"
            wait for 4 signal tick
            display message in score "2s to play"
            wait for 4 signal tick
            remove message in score
```

The "Save" 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. To load the orchestration, we click on "Load". "Start" allows to launch the orchestration without using the controller.

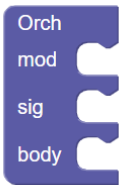
Remark: In development phase, the controller is not necessary, it is possible to compile and launch an orchestration from the Blockly interface. Nevertheless, the controller gives more information about the active pattern groups and ticks. The information on the groups is also visible with the orchestration display on a big screen (using the Skini "score" client), if this is defined in the configuration of the piece.

We will review the main blocks of the Blockly interface.

Note: "Save" compiles Blockly into a ./myReact/orchestrationHH.js file, which is the HipHop.js AST of the Blockly program. This file is not useful for the composer.

8.2 MODULES

These are the blocks defining the structure of the orchestration.



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

The blockly code is organized in blocks that can be paralleled.

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.

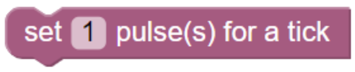
The interest of this block is to be able to easily activate the "collapse block" function of Blockly to make the orchestration more synthetic.

Resets the orchestration matrix, all active groups are disabled.

8.3 TIME PROGRAMING

The patterns are triggered by a queue mechanism. This mechanism has two roles. It avoids overlapping patterns for the same instrument and it allows to define a synchronization strategy for all patterns.

A pattern is placed in a queue each time it is selected. The queues are unstacked at regular intervals multiple of the pulse, which is called a tick. This interval is fixed with a signal emitted in the automaton with:



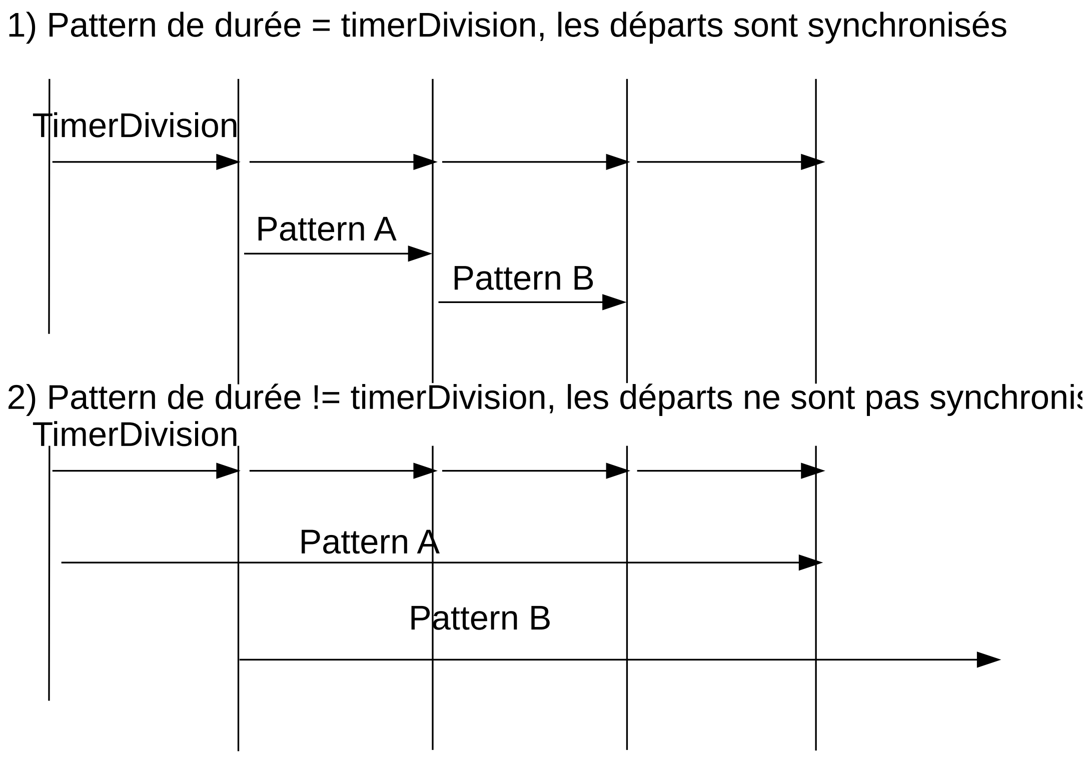
For pieces with homogeneous pattern durations, i.e., with the same duration for all patterns, you must give the tick this duration if you want the pattern starts at the same time.

When you want to introduce patterns with different durations, it is possible to give the tick a value corresponding to the shortest duration of a pattern. This means that the patterns will all be synchronized with each other on the value of the tick.

To understand the mechanism, you must understand that the queues are read at each cycle of the tick.

In the case where patterns have multiple durations of the tick. It is necessary to pay attention to the way in which the patterns can be chained. If all patterns have the same duration, for example 16 pulses and the tick is equal to 4 pulses. If a pattern “A” is called at time T and a pattern “B” at time T + 5 pulses for example. Pattern B will start a tick cycle after pattern A because the pattern starts are synchronized by the tick. Thus, the 16-pulse patterns, though they are of the same duration, can be shifted by 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
TimerDivision
Pattern A
Pattern B
2) Pattern de durée != timerDivision, les départs ne sont pas synchronis
TimerDivision
Pattern A
Pattern B

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

Note: It is possible to modify the tick duration with set pulse during a piece.

It is also possible to use the pulse provided by the DAW in an orchestration and not the ticks to measure the time, you must declare:

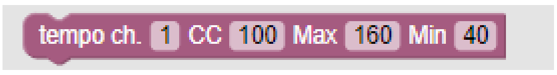
```
exports.pulsationON = true;
```

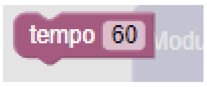
in the configuration file of the piece and include a signal *pulsation* “in” in the orchestration.

8.4 TEMPO

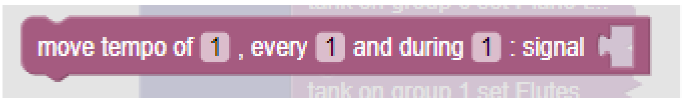
The modification of tempi from the orchestration requires the DAW to be set up so that it can receive Control Changes on the tempo. We have seen that a MIDI control bus is used to send

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.


tempo ch. 1 CC 100 Max 160 Min 40

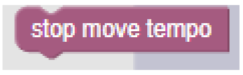
Once these parameters are set. The block: 
tempo 60

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


move tempo of 1 , every 1 and during 1 : signal

allows an automation of the tempo change. It allows the tempo to vary 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 the easy integration of tempo movement without programming.

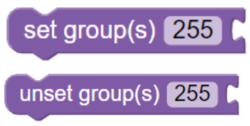
To interrupt the tempo movements, you must apply the block:


stop move tempo

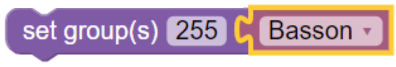
8.5 Tanks and patterns groups

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

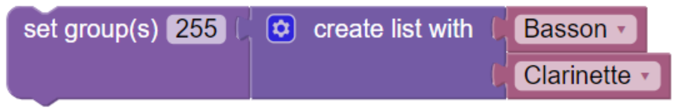
8.5.1 Patterns groups


set group(s) 255
unset group(s) 255

The user groups are defined in the configuration file of the piece. 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:

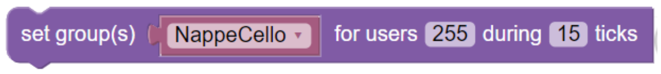

set group(s) 255 Basson

You can activate several groups with lists, ex :

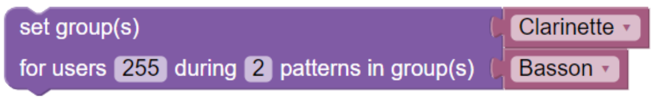

set group(s) 255 create list with Basson Clarinette

8.5.2 Actions for groups

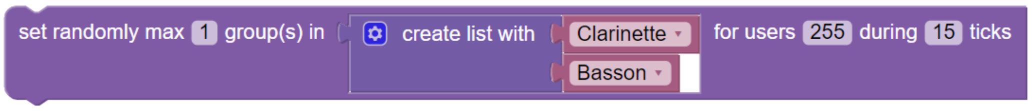
Skini provides high level blocks to allow complex actions with groups.


set group(s) NappeCello for users 255 during 15 ticks

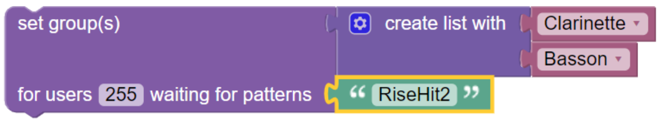
activates one or more groups during a period. Lists can be used instead of a single group.


```
set group(s) Clarinette
for users 255 during 2 patterns in group(s) Basson
```

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


```
set randomly max 1 group(s) in create list with Clarinette Basson for users 255 during 15 ticks
```

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

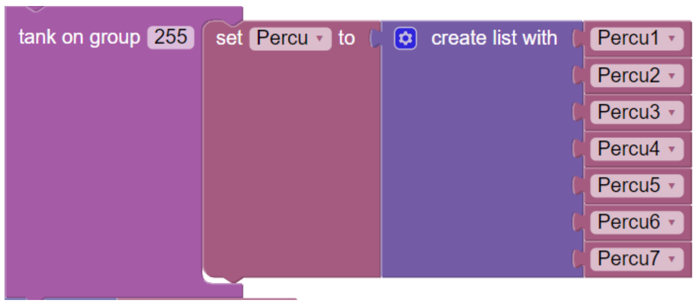

```
set group(s) create list with Clarinette
                              Basson
for users 255 waiting for patterns "RiseHit2"
```

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

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

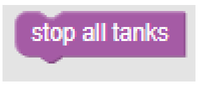
8.5.3 Creating tanks

A tank is a block with three levels. 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.

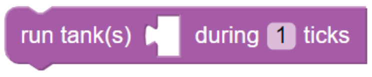

```
tank on group 255 set Percu to create list with Percu1
                                                Percu2
                                                Percu3
                                                Percu4
                                                Percu5
                                                Percu6
                                                Percu7
```

8.5.4 Actions on tanks

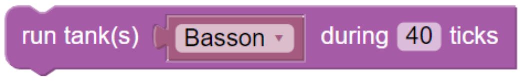
Tanks are in fact Skini submodules with pattern parameters. They are blocking in a flow. A tank will stop when it is empty or when it is killed with:


```
stop all tanks
```

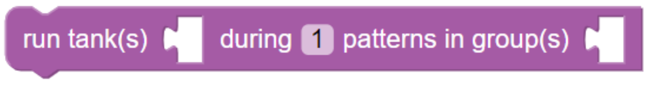
There are functions that simplify the management of tanks:

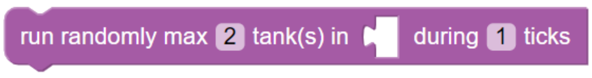

```
run tank(s) during 1 ticks
```

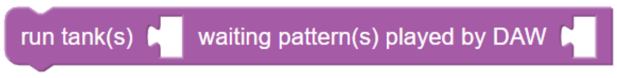
This block will launch one or more tanks given in parameter as a Blockly variable and will "kill" this tank after "during" occurrences of a signal. The following block will leave the patterns of the Basson reservoir available for a maximum duration of 40 ticks:


```
run tank(s) Basson during 40 ticks
```

It is easy to understand how the following blocks work:







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

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

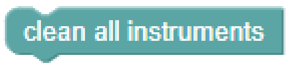
Remarks: *group* and *tank* processing uses Blockly *variables*. Everything that directly concerns *patterns* uses *strings*.

8.6 Instruments queuing

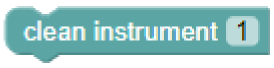
Queues are associated with instruments. They are created according to the pattern description in the pattern csv file.

8.6.1 Emptying the 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:



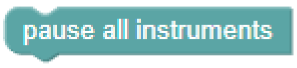
The skini client empties its choice list as soon as a clean all queue is performed. It can empty the queue of a particular instrument by giving its index with:



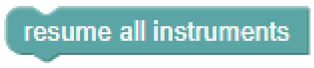
The skini client empties its choice list as soon as a *clean instrument* is performed. It does not consider the FIFO number. (This block does not perform a *cleanChoiceList*).

Caution: 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).

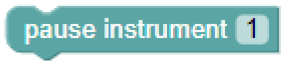
8.6.2 Pausing and testing queues



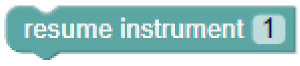
Stops the playing of patterns for all instruments.



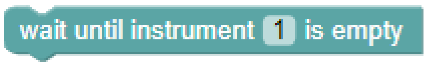
Resumes the playing of the patterns for all instruments.



Stops the playing of patterns for an instrument.



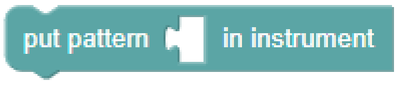
Resumes the playing of patterns for an instrument.



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

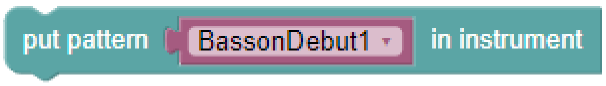
8.6.3 Queuing a specific pattern

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



The block parameter is a string with the pattern name.

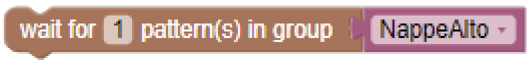
Ex :



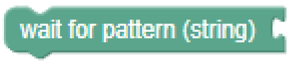
This block allows you to introduce "sequencer" type elements in a collaborative or generative music orchestration.

8.7 Patterns

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

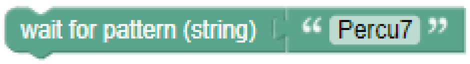


This block will wait for the execution (or the request depending on the setting of the piece) of several patterns belonging to a group.



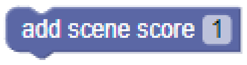
Is a block that waits for the execution (or the request according to the parameterization of the piece) of a certain pattern passed in parameter in the form of character string.

For example:

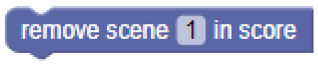


8.8 Orchestration display

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

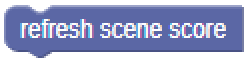

add scene score 1

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 piece configuration file.

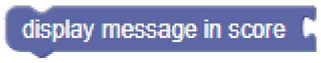

remove scene 1 in score

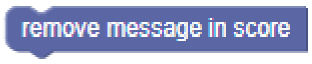
Make a level disappear.

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


refresh scene score

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


display message in score

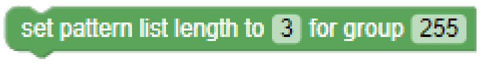

remove message in score

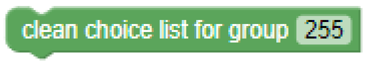
8.9 SKINI GAMES

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.

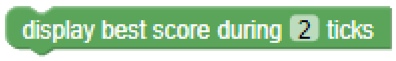
A basic model is proposed which consists in defining types of patterns and associating notes to the way the participant will organize his lists. The winner will be the one who accumulates the best lists during the play.

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

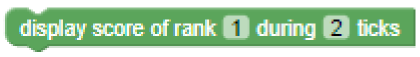

set pattern list length to 3 for group 255


clean choice list for group 255

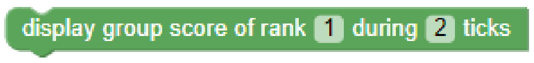
The following block displays the current best score:

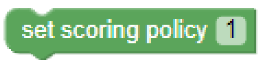

display best score during 2 ticks

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

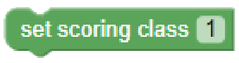

display score of rank 1 during 2 ticks

The " ranking " starts at 1.


display group score of rank 1 during 2 ticks



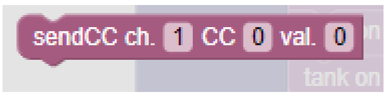
Defines the type of algorithm that will calculate the scores. These algorithms can be found in the *./server/computeScore.js* file.



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 file *./serveur/computeScore.js* on the treatment of the classes.

8.10 MIDI Control Change

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



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 modify this parameter.

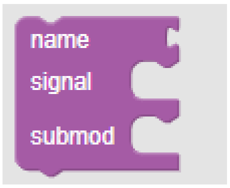
Remarks on MIDI commands: The operations performed by the blocks do not intervene on MIDI notes (in the Skini sense) but on groups of patterns, tanks and even patterns but using their names. The composer can play on MIDI channels in the case of blocks acting on Control Changes (CC).

8.11 Advanced blocks

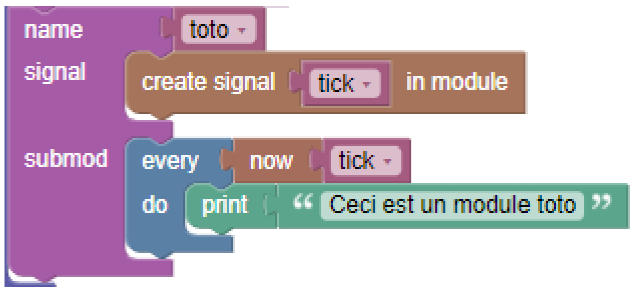
It is possible to create complex orchestrations without knowing the details of synchronous reactive programming, but for functions based on non-standard blocks there is a series of menus allowing to program in HipHop with Blockly. This assumes that you have the basics of synchronous reactive programming.

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

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

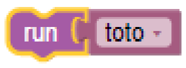


These sub-modules have input/output signals that must be created in the signal field. The body of the submodule is in *submod*. Here is an example of a submodule:



```
name    toto
signal  create signal tick in module
submod  every now tick
        do  print " Ceci est un module toto "
```

A submodule is executed with the run command:



```
run toto
```

8.12 Programming « stingers »

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 spot the "temporal" sequence of two patterns in two different FIFOs.

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 the queues explained in chapter Priority in queues, p.28.

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.

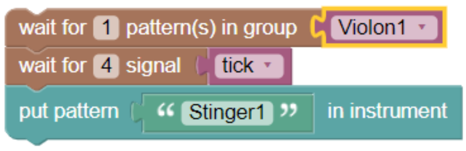
8.12.1 Case of the reaction to the execution

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 S delay, we need to set a tick that allows us to consider a delay when the stinger is launched. Note that tick must therefore be at least the duration of this delay (cf. Time programing, p.18).

A stinger can be programmed on the scheme:

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

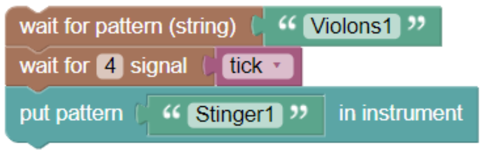
An example for violin patterns of 8 ticks with a stinger starting on the 4th tick. Here the tick would be worth one beat.



```
wait for 1 pattern(s) in group Violon1
wait for 4 signal tick
put pattern "Stinger1" in instrument
```

8.12.2 Feedback of a pattern set 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 done at the selection because the pattern signal is emitted by the DAW when the pattern is launched.



```
wait for pattern (string) "Violons1"
wait for 4 signal tick
put pattern "Stinger1" in instrument
```

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.

8.13 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 process a queue (FIFO). You must add the following command in the configuration file of the piece:

```
exports.algoGestionFifo = 1;
```

(value 1 assigned here just show that we can implement 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 patterns. 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 piece or 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 we play 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 an example of algorithm:

A) To add a pattern F to a queue:

1. If there are 2 Ds in the queue that follow each other, we insert F between the two.
2. If there are 2 M in a row, we insert F between them.
3. If the last element in 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)

So we can have sequences of F

B) To add a pattern D in a queue:

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

We can thus have sequences of D

C) To add a pattern M in a queue:

1. If in the queue there is a D immediately followed by an F we put M between the two.

2. If in the queue there are 2 Ds which follow each other, we put M between the two.

3. If there are 2 Fs in a row, 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, you don't have to 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 *./serveur/controleDAW.js* the function *ordonneFifo*() to have the detail of the algorithm actually in place.

8.13.1 Example of programming a game “choose the good instrument”

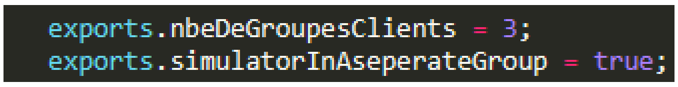
The file to load is ./myReact/trouveLaPercuNode.xml. This game use Bitwig studio with OSC communication, see the config file ./pieces/trouveLaPercuNode.js.

The principle is that two people, using the “standard” skini client will alternatively have to build lists of pattens which fit the best to music played in background. The one who did the best choices wins. For that we define “types” for the patterns. These are defined in the file ./pieces/trouveLaPercu.csv (generated from ./pieces/trouveLaPercu.xlsx).

The *scoring policy* which defines the rules is set to 2. (The score in computed in *./serveur/computeScore.js).*

For the game we defined several groups of patterns which mix “good” and “wrong” patterns according to the “scoring class”. If a selected pattern is in the good class, it means with a type equal to the class, the player wins points.

The patterns are used only in groups. The simulator is used on a dedicated group. This is described in the configuration file 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.

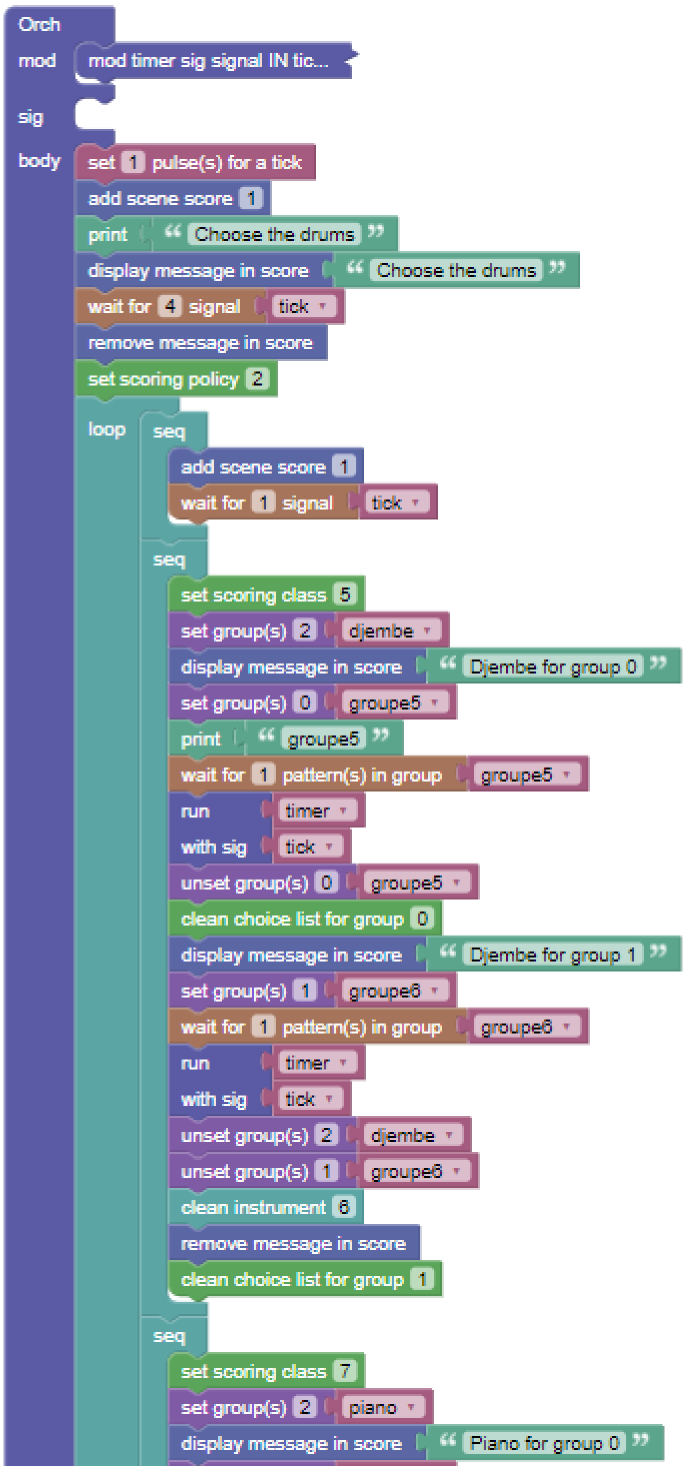


```
exports.nbeDeGroupesClients = 3;
exports.simulatorInAseperateGroup = true;
```

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 economical in resources 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.

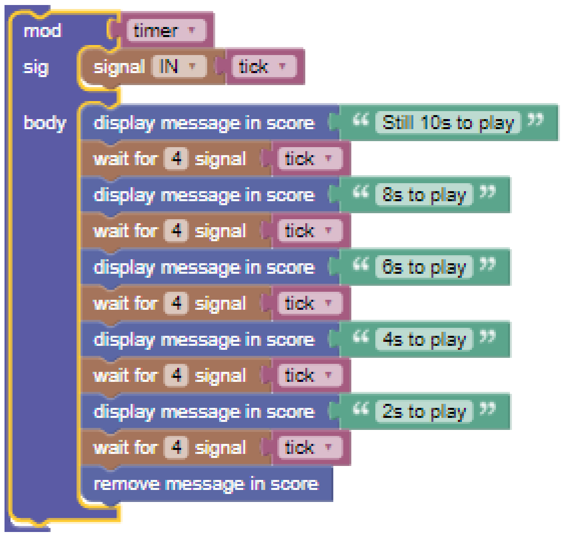
Here is the beginning of the Blockly program:



```
Orch
mod   mod timer sig signal IN tic...
sig
body  set 1 pulse(s) for a tick
      add scene score 1
      print " Choose the drums "
      display message in score " Choose the drums "
      wait for 4 signal tick
      remove message in score
      set scoring policy 2
      loop  seq
                add scene score 1
                wait for 1 signal tick
            seq
                set scoring class 5
                set group(s) 2 djembe
                display message in score " Djembe for group 0 "
                set group(s) 0 groupe5
                print " groupe5 "
                wait for 1 pattern(s) in group groupe5
                run timer
                with sig tick
                unset group(s) 0 groupe5
                clean choice list for group 0
                display message in score " Djembe for group 1 "
                set group(s) 1 groupe6
                wait for 1 pattern(s) in group groupe6
                run timer
                with sig tick
                unset group(s) 2 djembe
                unset group(s) 1 groupe6
                clean instrument 6
                remove message in score
                clean choice list for group 1
            seq
                set scoring class 7
                set group(s) 2 piano
                display message in score " Piano for group 0 "
```

8.13.2 Example of displaying timing timing data on the big screen

Here is the code of the module not displayed because "collapsed" in the previous example.
This module is named "timer", is called by the "run" block with a parameter "tick".



```
mod   timer
sig   signal IN  tick
body  display message in score  " Still 10s to play "
      wait for 4 signal  tick
      display message in score  " 8s to play "
      wait for 4 signal  tick
      display message in score  " 6s to play "
      wait for 4 signal  tick
      display message in score  " 4s to play "
      wait for 4 signal  tick
      display message in score  " 2s to play "
      wait for 4 signal  tick
      remove message in score
```

9 WITH MUSICIANS (TO BE UPDATED)

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

In the configuration file of the piece, to set up the specificities related to playing with musicians:

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

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

WARNING: This shift is done according to 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 piece we will have for example:

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

These paths are relative to the "./images" directory
Ces chemins sont relatifs au répertoire « ./images ».

10 OSC Interface OSC with Unreal Engine 4 (to be updated)

10.1 Principle

When a game development platform can send OSC commands, it is possible to generate signals in the orchestration from game events, and conversely to generate OSC events in the game from the orchestration.

10.2 From orchestration

For the implementation of the signals for the exchanges between Skini and the game platform we use a mechanism similar to the one used for the creation of the signals from the piece configuration file, here by means of two string arrays. One array for incoming OSC messages and one for outgoing OSC messages. There is a direct correspondence between the OSC messages and the HipHop.js signals. The outgoing and incoming signals can have values. For signals received by the orchestration, it is the react() that directly assigns the values from the received OSC commands.

The declarations of the signals are created in the file:

`.\serveur\autocontroleur\automateInt.js`.

The signal listeners are created when the orchestration automaton is created in :`.\serveur\autocontroleur\automateInt\groupeCliensSons.js`.

Sending OSC messages to the game platform is done with the *sendOSCGame(message, value)* function in ./server/logosCandMidiLocal.js called in ./server/autocontroller/customerGroupSounds.js when listeners are created.

For the signals entering the orchestration, the functions handling the reception of OSC commands must have access to the automaton HipHop.js to call the *automatonPossibleMachine.react()* function as does websocketServerSkini.js. The management of OSC exchanges is done in a specific module *.\serveur\gameOSC.js* which is initialized in websocketServerSkini.js because it is in this module that the orchestration automaton is created.

The orchestration implementation for the game does not need to allow the possibility of launching several automata in the same Skini session. There is therefore only one set of "OSC" signals per Skini session.

Note on OSC signals: Signals created in the configuration file for OSC should not have the same names as signals used in orchestration. It is prudent to define a common format for these signals by adding an OSC or Game suffix or prefix or other.

10.3 Implementation with orchestration

The implementation is very simple, the mechanism for creating signals for orchestration is set up if the signal arrays exports.gameOSCIn = [] and exports.gameOSCOut = [] are present in the configuration file of the piece.

Ex :

```
exports.gameOSCIn = [ "porte1", "porte2"];
```

```
exports.gameOSCOut = ["jumpOsc", "climbOsc" ];
```

It is also necessary that in this file there is the line: exports.`remoteIPAdressGame = ipConfig.remoteIPAddressGame;`

And that in ipConfig.js "remoteIPAddressGame" is set to the address of the game platform.

Ex : `"remoteIPAddressGame": "192.168.1.6"`

11 OSC AVEC 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:

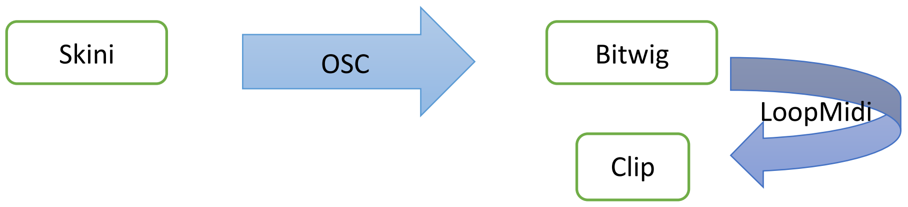
Remote OSC Ip address: 192.168.1.6

Remote OSC port out: 13000

Listening OSC : 12000

Midi information is sent to Bitwig as it is to Processing. There is no modification in Skini for the generation of 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 the midi commands from the Skini_0 controller.

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 correspondence between OSC commands from Bitwig to Skini is defined in the files Skini_0.control.js and midimix.js.

In addition, Bitwig issues 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.

Conclusion: The only modifications made to Skini for a first implementation of Bitwig are in midimix.js and in 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 considered.

12 Annexes

12.1 Skini with Ableton Live

We mainly 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 good enough. 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.

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 ("Téléc.") 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.

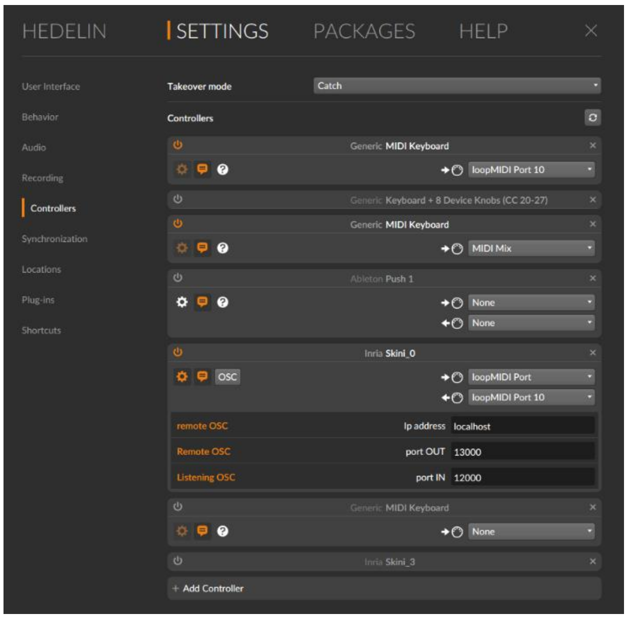
You have to activate the MIDI synchronization of Ableton on the OUT port corresponding to the IN port of the Processing gateway so that the tempo of Ableton controls Skini.

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.

12.2 Skini avec Bitwig Studio

With Bitwig studio it is 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 Bitwig Studio configuration.



Takeover mode: Catch

Controllers

Generic MIDI Keyboard
→ loopMIDI Port 10

Generic Keyboard + 8 Device Knobs (CC 20-27)

Generic MIDI Keyboard
→ MIDI Mix

Ableton Push 1
→ None
← None

Inria Skini_0
→ loopMIDI Port
← loopMIDI Port 10
remote OSC — Ip address: localhost
Remote OSC — port OUT: 13000
Listening OSC — port IN: 12000

Generic MIDI Keyboard
→ None

Inria Skini_3

12.3 EXAMPLES OF ORCHESTRATIONS

trouveLaPercuNode.xml : is an example of a game where the audience has to find the correspondence between patterns and a sound ambiance**.**

opus5Node.xml : : is an example of a configuration that uses mainly reservoirs with MIDI commands for transposition, mode change.

12.4 FILE SYSTEM ORGANISATION

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

client : The different skini clients are located in subdirectories.

docs

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

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

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

serveur : 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