

---

# **Skini sur Node.JS**

## **Mode d'emploi et implémentation**

B. Petit-Hédelin, 2/2026

## Table des matières

1	Introduction .....	5
1.1	Processus de composition.....	5
1.2	Patterns.....	5
1.3	Orchestration .....	5
1.4	Les files d'attente .....	6
1.5	Des pièces avec une DAW .....	6
1.6	Des pièces avec des musiciens (à porter dans node.js) .....	6
2	La configuration .....	7
2.1	Installation .....	7
2.2	Installation de Processing pour la passerelle osc/MIDI (obsolete) .....	7
2.3	Configuration.....	7
2.3.1	Configuration IP et répertoire .....	7
2.3.2	Accès aux pièces et descripteurs des patterns .....	8
2.4	Configuration MIDI .....	8
2.5	Configuration des pièces.....	10
2.5.1	Façon de recevoir les commandes MIDI .....	11
2.5.2	Mode de réaction.....	11
2.5.3	Les fichiers sons pour les clients et les clients .....	11
2.5.4	Synchronisation .....	11
2.5.5	Groupes de patterns .....	12
2.5.6	Musiciens (Pas en place) .....	12
2.6	Configuration des patterns .....	13
2.7	Configuration de l'affichage de l'orchestration sur grand écran .....	14
3	Lancer Skini .....	16
4	Utilisation du contrôleur .....	17
5	Utilisation du configurateur MIDI .....	17
6	Utilisation du client SKINI .....	18
7	Utilisation du simulateur .....	19
7.1	Spécialisation du simulateur .....	19
7.2	Activation .....	19
7.3	Paramétrage.....	20
7.4	Gestion des types de patterns dans le simulateur .....	20
7.5	Exemple d'utilisation de listes de type : le würfelspiel .....	21
8	Programmation graphique des orchestrations.....	22

8.1	L'interface graphique de Skini .....	23
8.2	Les bloc d'orchestration.....	23
8.3	Les blocs Hiphop.....	24
8.4	La programmation du temps dans une pièce .....	24
8.4.1	Utilisation du signal tick .....	25
8.4.2	Utilisation du signal pulsation .....	26
8.5	Tempo.....	26
8.6	Réservoirs et groupes de patterns .....	27
8.6.1	Groupes de patterns .....	27
8.6.2	Actions liées aux groupes.....	27
8.6.3	Création des réservoirs .....	28
8.6.4	Actions sur les réservoirs.....	30
8.7	Les files d'attentes des instruments .....	31
8.7.1	Vidage des files d'attentes .....	31
8.7.2	Mise en pause et test des files d'attente .....	32
8.7.3	Mettre un pattern spécifique en files d'attente .....	32
8.8	Patterns.....	33
8.9	Affichage de l'orchestration.....	33
8.10	Jeu Skini.....	33
8.11	Commande et Control Change MIDI.....	34
8.12	Spécifique Ableton Live.....	35
8.13	Envoi de Commandes OSC vers Raspberry .....	35
8.14	Les blocs avancés.....	35
8.15	Composition de pièces polyphoniques .....	36
8.16	Contrôle Interface Z.....	38
8.17	Programmation des transitions « stingers » .....	39
8.17.1	Cas de la réaction à l'exécution .....	40
8.17.2	Cas du retour de jeu de pattern depuis la DAW .....	40
8.18	Priorité dans les files d'attente .....	41
8.18.1	Exemple de programmation d'un jeu : trouve la percu .....	42
8.18.2	Exemple de transposition en boucle.....	44
8.19	Quelques subtilités de la programmation réactive synchrone .....	45
8.19.1	Réaction immédiate.....	45
8.19.2	Double émission d'un même signal .....	45

8.20	Pour la musique générative .....	46
8.21	Spatialisation par tuilage avec le dispositif Pré.....	47
9	Programmation textuelle des orchestrations .....	51
9.1	Interface de contrôle pour la programmation textuelle.....	51
9.2	Les fonctions de Skini via HipHop.js (à faire) .....	52
10	Avec des musiciens (pas en place avec Node.js).....	52
11	Interface OSC .....	53
11.1	Principe .....	53
11.2	Depuis l'orchestration.....	53
11.3	Mise en œuvre avec l'orchestration.....	54
12	OSC avec Bitwig Studio .....	55
13	OSC avec Max4Live.....	56
13.1	Piste de conversion.....	56
13.2	Piste de routage Midi.....	56
13.3	Piste de feedback .....	57
13.4	Les patches M4L.....	58
13.5	Conclusion OSC Skini avec Ableton .....	59
14	Contrôle de patterns sur Raspberry .....	60
15	Annexes .....	60
15.1	Skini avec Ableton Live.....	60
15.2	Skini avec Bitwig Studio .....	61
15.3	Exemples d'orchestrations.....	61
15.4	Organisation du système de fichier .....	61
15.5	Enregistrement de partition dans Finale .....	62
15.5.1	A partir d'un enregistrement d'Ableton.....	62
15.5.2	Directement depuis Skini .....	63
15.6	Passer de Finale à Microsoft Word ou PowerPoint.....	63
15.7	Recevoir les infos MIDI de lancement de clip d'Ableton .....	63
15.8	Enregistrement de patterns en Live dans le séquenceur distribué.....	65
16	Un exemple de pièce : Opus5 .....	66
17	Une pièce avec contrôle des transposition directement avec CC Midi.....	70

# 1 INTRODUCTION

---

Skini est une plateforme hybride de composition de musique collaborative et générative basée sur la programmation modulaire. Ce document aborde les points techniques relatifs à la mise en œuvre de Skini sur Node.js et la création d'orchestration. Pour comprendre le fonctionnement de Skini il est conseillé de se reporter au document « *Temps et Durée : de la programmation synchrone à la composition musicale* » ou aux différents articles parus sur cette plateforme (Programming journal 2020, ICMC 2021, NIME 2019).

Ce document comprend aussi en annexe quelques procédures relatives à l'utilisation de Finale et d'Ableton Live avec Skini.

## 1.1 PROCESSUS DE COMPOSITION

Skini a été conçu pour composer de la musique qui sera exécutée en interaction avec une audience ou produite automatiquement par des processus aléatoires. La solution comporte un serveur Web qui intègre des modules d'orchestration écrits au moyen d'un outil de programmation graphique qui est une couche d'abstraction au-dessus du langage HipHop.js. Il est aussi possible de programmer en HipHop.js et JavaScript sans utiliser l'interface de programmation graphique.

La méthode de composition repose sur trois concepts de base : les patterns, les files d'attente et l'orchestration. Ces éléments sont décrits dans la thèse « Le temps et la durée : de la programmation réactive synchrone à la composition musicale ». Nous n'abordons ici que la dimension pratique de l'outil, c'est-à-dire sa mise en œuvre.

## 1.2 PATTERNS

Le compositeur peut créer les patterns sans contraintes particulières de la part de Skini. Ils seront vus comme des éléments activés par une commande MIDI et ayant des durées définies en nombre de pulsations. Les patterns sont mis à la disposition de l'audience sous forme de groupes<sup>1</sup>. Il n'y a pas de contrainte sur les tailles des groupes de patterns.

## 1.3 ORCHESTRATION

Le compositeur va définir la façon dont les groupes sont mis à la disposition de l'audience au moyen de l'orchestration. L'orchestration permet *d'activer* et de *désactiver* des groupes de patterns. Parmi les informations qui lui permettent d'évoluer nous avons, l'écoute des sélections de groupe par l'audience, la mesure de la durée, l'écoute d'informations MIDI ou OSC.

L'orchestration peut être vue comme un « super séquenceur » apportant des fonctions d'interaction basées sur des *files d'attente* et apportant des automatismes complexes à des DAW du commerce. L'orchestration peut aussi commander la mise en file d'attente de patterns

---

<sup>1</sup> Si le compositeur souhaite mettre à la disposition de l'audience ou du simulateur un pattern particulier, il peut créer un groupe avec un seul élément par exemple.

sans audience. Il est possible d'émettre des commandes MIDI, note ou « control changes » directement depuis l'orchestration.

L'orchestration est écrite en langage HipHop.js sous forme graphique. Une connaissance élémentaire de la programmation synchrone est nécessaire pour se lancer dans la conception d'une première orchestration. Avec une bonne maîtrise de HipHops.js il sera possible de produire des pièces riches et inconcevables autrement.

## 1.4 LES FILES D'ATTENTE

Les patterns ne sont pas joués immédiatement quand ils sont sollicités par une audience ou un processus aléatoire (simulateur ou générateur). Comme Skini fonctionne en temps réel, les files d'attente garantissent qu'aucune sollicitation ne sera perdue. Il peut donc y avoir un décalage temporel plus ou moins grand pour entendre un pattern selon l'état des files d'attente associées à chaque « instrument ». Les files d'attente, selon le rythme auquel elle sont remplies, permettent des traitements particuliers comme la vérification de la compatibilité entre pattern par exemple.

## 1.5 DES PIECES AVEC UNE DAW

Skini est un système hybride qui nécessite un outil pour produire du son. Nous n'aborderons dans ce document le cas d'Ableton Live et celui de Bitwig Studio. L'utilisation d'une autre DAW reposera sur les mêmes principes que ceux proposés pour l'une ou l'autre solution.

Le compositeur doit créer des patterns (des clips). A chacun de ces patterns le compositeur associera une note Skini. Cette note Skini sera convertie et envoyée vers la DAW à partir de commandes émises par l'audience ou d'un processus aléatoire.

Le compositeur peut créer autant de patterns qu'il le souhaite tout en gardant en tête que ces patterns seront organisés en groupes et que ce seront ces groupes qui seront mis à la disposition de l'audience ou du processus aléatoire. Pour une pièce collaborative, il faut donc que les dimensions des groupes soient compatibles avec l'affichage possible sur une interface pour l'audience. Il faut aussi que le compositeur trouve un bon équilibre entre des patterns courts qui vont dynamiser l'interaction et des patterns longs qui vont stabiliser le discours musical.

Dans le cas d'une interaction avec l'audience, pour chaque patterns le compositeur devra créer un *fichier son*, mp3 ou wav, dont le nom est associé au pattern dans le fichier de configuration des patterns. Par défaut il s'agit de fichier mp3 qui seront téléchargés par l'audience pour écoute avant une sélection.

Une fois les patterns créés et les commandes MIDI associées, il reste à créer les fichiers de configuration comme décrit au chapitre des configurations, puis passer à l'orchestration.

## 1.6 DES PIECES AVEC DES MUSICIENS (A PORTER DANS NODE.JS)

Skini peut être utilisé pour dialoguer avec des musiciens, avec des Raspberry Pi possédant le client adapté, avec ou sans synthétiseurs. La démarche est identique à celle d'une DAW. Les patterns activés ne consistent plus à émettre une commande MIDI mais à afficher sur des clients dédiés aux musiciens des partitions préalablement déposées dans un sous répertoire du répertoire « ./image ». Ce sous-répertoire est configuré dans le fichier de configuration de la

pièce. Ces fichiers sont au format jpg. Les noms des fichiers jpg sont les mêmes que ceux des sons associés aux patterns.

## 2 LA CONFIGURATION

---

Les répertoires sont référencés par rapport au répertoire principal où est installé Skini.

### 2.1 INSTALLATION

Il suffit d'installer Node.js qui est une solution très largement utilisée et de copier les fichiers Skini dans un répertoire. S'il manquait des packages au moment du lancement de Skini, Node.js le signalerait avec des messages d'erreur. Il suffit alors d'installer les packages avec npm.

### 2.2 INSTALLATION DE PROCESSING POUR LA PASSERELLE OSC/MIDI (OBSOLETE)

La passerelle OSC/MIDI est utilisée lorsque que la DAW ne comprend que les commandes MIDI et que le serveur Skini n'est pas sur la même machine que la DAW. Si le serveur Node.js et la DAW sont sur le même ordinateur il n'est pas nécessaire d'installer la passerelle entre OSC et MIDI et donc cette étape n'est pas nécessaire. Vous pouvez utiliser Skini en accédant directement au port MIDI de votre machine.

Donc si la communication avec la DAW se fait en MIDI et que vous utilisez un ordinateur qui héberge le serveur Node.js et un autre ordinateur qui héberge la DAW, il faudra installer Processing ([www.processing.org](http://www.processing.org)) les deux ordinateurs pourront communiquer en OSC et il faudra la passerelle. Dans Processing il faudra installer les librairies oscP5, TheMidiBus, WebSockets (menu : ajouter un outil -> librairies).

**Attention :** Au 1/12/2025, cette passerelle n'est plus utilisée. Le paramètre busMidiDAW est forcé à 1. Si la passerelle devait être utilisée il faudra revoir l'affectation de ce paramètre dans la fenêtre « Parameters ».

### 2.3 CONFIGURATION

Il y a deux fichiers de configuration à mettre à jour avant de lancer Skini. Un concerne la configuration réseau, l'autre la configuration MIDI.

#### 2.3.1 Configuration IP et répertoire

Elle se fait avec le fichier « ./serveur/ipConfig.json »

Exemple d'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.     "webserveurPort": 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` : adresse du serveur.

`remoteIPAddressLumiere` : pour un usage avec QLC pour un dialogue en OSC

`remoteIPAddressDAW` : IP du serveur Processing pour les commandes MIDI vers Ableton ou aune autre DAW.

`remoteIPAddressSound` : IP du serveur Processing pour les commandes MIDI vers REAPER, c'est pour le spectacle GOLEM.

`remoteIPAddressImage` : IP du serveur Processing pour une Visualisation sur grand écran.

Pour un usage standard, il suffit de mettre à jour `remoteIPAddressAbleton` et `serverIPAddress`.

Le changement de port pour les Websockets « `websocketServeurPort` », se fait avec le script powershell ***changePortSkini.ps1***. (Sinon il faut repasser browserfity sur les clients).

### 2.3.2 Accès aux pièces et descripteurs des patterns

Les paramètres `sessionPath` et `piecePath` font partie du paramétrage système et se trouve donc au même niveau que le réseau. `sessionPath` définit le répertoire des paramètres de la pièces et `piecePath` ceux de l'orchestration Blockly ou HipHop.js. Il n'est pas possible d'accéder au chemin complet depuis un navigateur, c'est pour cela que ces paramètres sont nécessaires.

**Attention** : Il faut donc relancer Skini si l'on change de répertoire pour une pièce dans `ipConfig.js`.

## 2.4 CONFIGURATION MIDI

La configuration des ports MIDI se fait à l'aide du fichier `./serveur/midiConfig.json`. Ce fichier définit les bus MIDI en fonction de la configuration de l'ordinateur. Ce fichier est utilisé par Skini et Processing.

Voici un exemple en utilisant LoopMIDI sur Windows :



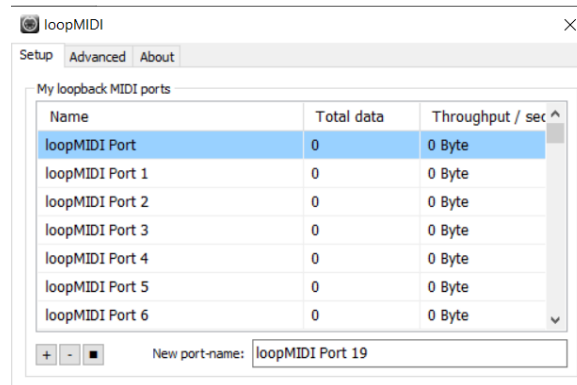


Figure 1 : Le câble virtuel LoopMIDI

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

Le champ « type » définit s'il s'agit d'un port IN ou OUT.

Le champ « spec » définit l'usage avec :

- « clipToDaw » définit au port qui permet à la DAW de recevoir les commandes MIDI de Skini.
- « syncFromDAW » définit le port qui va recevoir la synchronisation MIDI de la DAW.
- « clipFromDAW » définit le port via lequel la DAW envoie les message de départ des clips.

- « controle » correspond à un port auquel est associé un contrôleur MIDI (clavier, PAD...). n'est pas nécessaire au fonctionnement de base de Skini.

Le champ « name » contient le nom du port MIDI sur l'ordinateur. Ici il s'agit de ports sur l'interface LoopMIDI.

Le champ « comment » permet de commenter l'utilisation du port.

Voici un exemple de configuration des ports MIDI correspondant à l'exemple ci-dessus dans Ableton.



Le port 6, en IN pour Ableton et OUT pour Skini permet le contrôle depuis Skini.



Le port 9 envoie la synchronisation MIDI à Skini. Ceci n'est utilisé que si on utilise le mode de synchronisation MIDI ou MIDI/OSC avec une passerelle. Avec Link ce n'est pas nécessaire.



Le port 12 est utilisé pour les informations de « retour » d'Ableton, quand un clip est joué.



Pour plus de détail, notamment avec Bitwig Studio, voir les annexes de ce document.

**Remarque :** Le paramètre *busMidiDAW* du fichier de configuration d'une pièce, n'est utile que dans le cas de l'utilisation de la passerelle *Processing*. C'est un index qui permet à *Processing* de trouver le port MIDI pour émettre les *noteOn* vers la DAW. Si on n'utilise pas la passerelle *Processing*, *busMidiDAW* n'est pas utilisé. L'utilisation de la passerelle dans le cas de *Node.js* n'intervient que lorsque l'on a la DAW et le serveur sur deux machines différentes car on peut parler MIDI depuis *node.js*. En 12/2025 la passerelle n'est plus utilisée car avec *Node.js* MIDI fonctionne sans avoir besoin de passer par un intermédiaire. La passerelle était nécessaire dans la première version de Skini sous *hop.js*.

## 2.5 CONFIGURATION DES PIÈCES

Les pièces sont configurées à partir d'une fenêtre ouverte par le bouton « Parameters » de la fenêtre principale Blockly. (Les patterns sont configurés en cliquant sur le bouton « Patterns »).

**Pour les curieux :** La pièce est configurée dans un fichier JavaScript, ce fichier est chargé en même temps que l'orchestration. Il est sélectionnée à partir de la fenêtre de programmation de l'orchestration. Le nom de ce fichier doit correspondre à l'orchestration. Par exemple pour l'orchestration *opus1.xml* on doit avoir le fichier de configuration *opus1.js*.

Nous allons passer en revue les paramètres de configuration de la pièce.

### 2.5.1 Façon de recevoir les commandes MIDI

Pour le paramètre « Direct Midi » coché signifie que la communication entre Skini et la DAW se fait via MIDI. Sinon la communication entre Skini et la DAW se fait via OSC. OSC est utilisé avec la passerelle Processing essentiellement pour Ableton Live, Bitwig Studio peut communiquer directement en OSC moyennant l'installation du contrôleur Skini.

### 2.5.2 Mode de réaction

Le paramètre « React on Play » définit la façon de faire réagir l'automate d'orchestration. Par défaut c'est à la sélection. Avec « React on Play » coché c'est au moment où se joue le pattern. Ceci a un impact important sur la façon de penser l'automate. Les *stingers* (pour des transitions) ne sont possibles qu'avec « React on Play » actif.

### 2.5.3 Les fichiers sons pour les clients et les clients

Le paramètre Sound Files Path définit le chemin des fichiers sons, associés aux patterns, qui sont téléchargés par les clients depuis le répertoire . « \sounds ».

Le paramètre « Number of client groups » fixe le nombre de groupes de personnes dans l'audience que l'orchestration peut gérer.

Avec le paramètre « Simulator in a separate Group » nous introduisons la possibilité de dédier un groupe au simulateur. C'est-à-dire que les patterns disponibles pour le simulateur ne seront pas vu par l'audience. Pour ceci il faut un nombre de groupe supérieur à 2, car le dernier groupe sera celui du simulateur. Si le paramètre « Simulator in a separate Group » n'est pas coché, c'est qu'il n'y a pas de groupe dédié au simulateur.

Le paramètre « Algo Fifo management » permet d'activer avec un entier un algorithme de traitement sur les files d'attente (voir plus bas).

### 2.5.4 Synchronisation

Il existe 4 modes de synchronisation possibles. Par Midi, par Midi via OSC, par Ableton Link et en local avec un worker node.js. Il ne faut avoir qu'un seul mode à la fois.

Le mode Midi est le plus simple, il fonctionne avec toutes les DAW qui peuvent émettre une synchro Midi. Il est possible de communiquer en utilisant OSC avec Bitwig studio car il y a un contrôleur de Skini qui permet cela. Pour Ableton il faudra utiliser la passerelle Processing. Ableton Link est très simple et permet de se synchroniser en réseau.

Le worker de Node.js est utile pour des projets sans DAW, donc avec Musicien ou Raspberry qui ne donnent pas de synchro, et sans synchro externe Ableton Link venant d'un quelconque logiciel. C'est surtout un outil de test quand on n'a pas de DAW.

Pour choisir le mode de synchronisation nous avons les paramètres :

- *Syncho On Midi Clock* est utilisé pour une synchronisation MIDI venant de la DAW
- *Syncho Link* est utilisé pour une synchronisation venant du protocole Ableton Link.
- *Syncho Skini* est utilisé pour une synchronisation depuis Skini gère sa propre synchronisation selon un le paramètre *timer* qui donne le tick en ms. Dans ce cas on ne sait pas changer le tempo dans l'orchestration.

Si toutes ces synchronisations sont inactives Bitwig peut envoyer la synchro en OSC via le contrôleur Skini de Bitwig.

**Attention :**

- 1) Il faut relancer Skini quand on change de mode de synchronisation.
- 2) Il ne faut pas avoir la synchro OSC de Bitwig via le contrôleur Skini0 en même temps qu'une autre synchro. On recevrait trop de messages de synchronisation potentiellement décalés et en double.

### 2.5.5 Groupes de patterns

Le nommage des groupes de patterns se fait via un tableau accessible depuis la fenêtre principale et le bouton « Parameters ». Les noms des groupes de patterns sont utilisés pour la création des signaux HipHop de l'orchestration. Voici un exemple :

	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 Musiciens (Pas en place)

La présence de musiciens doit être spécifiée avec les lignes :

```
1. exports.avecMusicien = true;  
2. exports.decalageFIFOavecMusicien = 4;  
3. exports.patternScorePath1 = "";
```

decalageFIFOavecMusicien donne le décompte de pulsations avant le jeu du premier pattern. En effet, un musicien a besoin de se préparer avant de jouer un pattern contrairement à une DAW. Ce paramètre introduit un décalage systématique si aucun pattern ne se trouve en file

d'attente pour l'instrument concerné. S'il y a des patterns en file d'attente, le client musicien affiche le pattern suivant celui en cours. Ce qui permet au musicien de ne pas être surpris.

Les `patternScorePath1` est le sous-répertoire du répertoire « `./images` » où se trouvent les partitions de l'orchestration.

Dans le client musicien il faut se logger avec le numéro de l'instrument. C'est ce qui permet à au serveur.js de gérer les messages comme des images.

## 2.6 CONFIGURATION DES PATTERNS

Elle se fait à partir de la fenêtre ouverte depuis la page d'orchestration en cliquant sur « Patterns ». (Ceci génère un fichier csv qui se trouvent dans le répertoire spécifié dans la configuration de la pièce.)

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	Vert. type	Group Index	Duration	IP address	Buffer num	Level
1	11	10	0	canon5-1	canon5-1	0	0	0	1	1	8			
2	12	10	0	canon5-2	canon5-2	0	0	0	1	1	8			
3	13	10	0	canon5-3	canon5-3	0	0	0	1	1	8			
4	14	10	0	canon5-4	canon5-4	0	0	0	1	1	8			
5	15	10	0	canon5-1-suite	canon5-1-suite	0	0	0	1	1	8			
6	16	10	0	canon5-2-suite	canon5-2-suite	0	0	0	1	1	8			
7	17	10	0	canon5-3-suite	canon5-3-suite	0	0	0	1	1	8			
8	18	10	0	canon6-1	canon6-1	1	0	0	2	2	8			
9	19	10	0	canon6-2	canon6-2	1	0	0	2	2	8			
10	20	10	0	canon6-4	canon6-4	1	0	0	0	2	8			
11	21	10	0	canon6-1-suite	canon6-1-suite	1	0	0	2	2	8			
12	22	10	0	canon6-2-suite	canon6-2-suite	1	0	0	2	2	8			

La première colonne donne la note MIDI correspondant au pattern dans la DAW. Ces notes ne correspondent pas exactement à des notes MIDI. En effet, pour simplifier le codage nous n'avons pas imposé de limites sur ces numéros contrairement au standard MIDI qui ne permet que 128 notes sur un canal. On peut aller au-delà de la limite de 127. La transcription en note MIDI consiste à appliquer le calcul suivant :

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

On voit donc que la notion de canal MIDI est comprise dans la *note* Skini. L'équivalence entre notes Skini et note MIDI n'est donc immédiate que pour les notes < 127. Ceci permet d'associer des commandes MIDI à des patterns sans contrainte sur les canaux. Cette méthode nous permet de nous affranchir d'une gestion fastidieuse des canaux.

La colonne **Note Stop**, est la note MIDI permettant d'interrompre un pattern en cours. Ceci est propre au DAW comme Ableton Live.

**Flag Usage**, n'est pas un paramètre, c'est un outil pour le moteur Skini.

La colonne **Text** donne les textes qui seront associés à chaque pattern pour les différents clients. La colonne *Fichier son*, donne les noms de fichiers son, qui sont dans les répertoires définis dans le fichier de configuration JavaScript, par défaut ce sont des fichiers mp3. Pour utiliser des fichiers *wave*, il faut ajouter une extension « *.wav* » aux noms des fichiers.

La colonne **Instrument** associe les patterns à un instrument spécifique qui correspond à un instrument MIDI ou un musicien. Il n'y a pas de correspondance entre ces numéros et une configuration MIDI.

La colonne **Slot** est relative à des développements en cours sur l'enregistrement de patterns en Live. Ils peuvent être ignorés pour des sessions Skini sans enregistrement Live.

La colonne **Type** permet de qualifier un pattern pour pouvoir réorganiser les files d'attente, la description de cette fonctionnalité est dans le chapitre 8.18, « Priorité dans les files d'attente ».

La colonne **Vert. Type** permet de définir des types verticaux (ou harmonique) pour ne rendre possible que le jeu de patterns compatibles entre eux. La fonctionnalité est décrite dans le chapitre « Composition de pièces polyphoniques » page 36.

La colonne **Group Index** est en correspondance avec les index des groupes décrits dans la configuration des groupes (parameters). C'est ce paramètre qui associe le pattern à un groupe et permet la mise à disposition des patterns via l'orchestration.

La colonne **Durée (Duration)** définit la longueur du pattern en nombre de **pulsations** émises par la synchronisation en général MIDI.

La colonne **IP address** permet d'associer un pattern à une commande OSC lorsque Skini fonctionne avec des équipements distribués comme des Raspberries Pi possédant une application adaptée. Toujours avec des Raspberries Pi **Buffer Num** est un paramètre de la commande OSC qui définit le buffer associé au pattern. **Level** est utilisé pour définir le niveau sonore propre à ce pattern. Si le champ **buffer num** est vide le pattern est activé sur la DAW. Il est possible d'envoyer des commandes OSC depuis cette fenêtre pour tester les patterns distribués.

## 2.7 CONFIGURATION DE L'AFFICHAGE DE L'ORCHESTRATION SUR GRAND ECRAN

Il est possible d'afficher un déroulement de l'orchestration dans un navigateur. Skini propose un affichage à partir de boîtes connectées. Les groupes sont représentés par des rectangles et les réservoirs par des rectangles avec des bords arrondis. La taille des réservoirs se réduit au fur et à mesure qu'ils se vident. On accède à un visuel de l'orchestration depuis la fenêtre de programmation ou avec l'url :

<http://serveur/score>

Le paramétrage de cet affichage se fait depuis la fenêtre « Parameters » de la pièce. Plus bas nous verrons un extrait de paramétrage d'un graphique pour l'opus1. Pour chaque groupe de patterns nous avons des champs donnant la position des rectangles, une couleur, une liste de prédécesseurs c'est-à-dire de groupes pointant vers le groupe de la ligne et un numéro de scène. Le format est un peu différent pour des groupes et des réservoirs. Entre parenthèses il s'agit de l'index du tableau de la ligne.

Pour les groupes : nom du groupe (0), index (1), type (2), x(3), y(4), nbe d'éléments(5), color(6), prédécesseurs(7), n° de scène graphique(8)

Pour les réservoirs : nom du réservoir (0), index (1), type (2), x(3), y(4), numéro du tank(5), couleur(6), Previous(7), n° de scène graphique(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

Le paramétrage des réservoirs est plus complexe que celui des groupes. Un réservoir est un ensemble de groupes. Un réservoir est défini par le champ en cinquième position. Tous les groupes/patterns d'un réservoir ont le même numéro de réservoir.

Il faut faire attention à la numérotation des prédécesseurs. Un groupe ou un réservoir peuvent être utilisé comme prédécesseur.

Dans la configuration ci-dessous générée au format texte et non à partir du tableur des paramètres de Skini nous avons ajouté en fin de ligne en commentaire le numéro du groupe en tant que prédécesseur. Jusqu'à la ligne 11 tout est simple. A la ligne 12 nous commençons un réservoir qui se termine en ligne 15. Ces 4 lignes sont dans le même « prédécesseur » de valeur 11. Donc en ligne 16 commence le « prédécesseur » de valeur 12. (Notons que les couleurs sont exprimées au format hexadécimal).

```

1. ["violonsEchelle", 0, "group", 323, 176, 12, ocre, [3,8], 1], //0
2. ["violonsChrom", 1, "group", 800, 180, 16, ocre, [9], 2], //1
3. ["violonsTonal", 2, "group", 450, 25, 24, ocre, [30], 3], //2
4. ["altosEchelle", 3, "group", 200, 114, 12, violet, [5], 1], //3
5. ["altosChrom", 4, "group", 800, 270, 16, violet, [9], 2], //4
6. ["cellosEchelle", 5, "group", 52, 173, 10, vert, [], 1], //5
7. ["cellosChrom", 6, "group", 800, 360, 16, vert, [9], 2], //6
8. ["cellosTonal", 7, "group", 650, 100, 8, vert, [2], 3], //7
9. ["ctrebassesEchelle", 8, "group", 200, 244, 12, bleu, [5], 1], //8
10. ["ctrebassesChrom", 9, "group", 590, 430, 16, bleu, [25, 29], 2], //9
11. ["ctrebassesTonal", 10, "group", 650, 160, 6, bleu, [2], 3], //10
12. ["trompettesEchelle1", 11, "tank", 273, 374, 1, orange, [5,0], 1], //11
13. ["trompettesEchelle2", 12, "tank", 200, 10, 1, orange, [], ],
14. ["trompettesEchelle3", 13, "tank", 200, 10, 1, orange, [], ],
15. ["trompettesEchelle4", 14, "tank", 200, 10, 1, orange, [], ],
16. ["trompettesTonal1", 15, "tank", 650, 280, 2, orange, [2], 3], //12
17. ["trompettesTonal2", 16, "tank", 200, 100, 2, orange, [], ],
18. ["trompettesTonal3", 17, "tank", 200, 100, 2, orange, [], ]

```

L'affichage de l'orchestration est déclenché par l'orchestration.

**Note :** Le client score utilise la version JavaScript de Processing, P5js. Le code source se trouve dans `./Processing/P5js/score/score.js`

### 3 LANCER SKINI

Pour que Skini fonctionne avec des musiciens sans électronique, seul Node.js est nécessaire. Avec de l'électronique il faut :

- Une Digital Audio Workstation (par exemple Ableton Live ou Bitwig Studio).
- Eventuellement Processing du MIT qui va faire la passerelle entre OSC et MIDI.

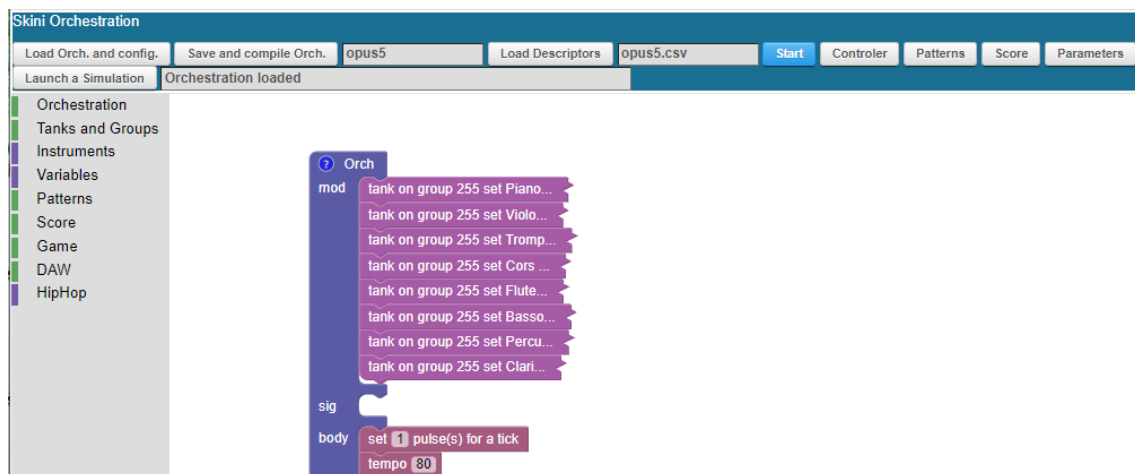
Pour lancer Skini il faut :

- Lancer : **node skini.mjs**

Avec électronique il faut ensuite :

- 1 Si on utilise la passerelle OSC/MIDI, lancer *Processing* avec le programme *sequenceurSkini.pde*. La console Processing signale que la passerelle s'est connectée sur le serveur. Rappelons qu'avec Bitwig studio et le contrôleur Skini\_0, on n'utilise pas cette passerelle.
- 2 Lancer et charger les patterns dans la DAW. Lancer la lecture sur la DAW pour activer la synchronisation MIDI si celle est utilisée. L'affichage de Processing doit défiler ce qui signifie que Skini est en route.

Il est possible à présent de charger la programmation de l'orchestration dans un navigateur Web avec `http://IP du serveur :8080/block`. Skini est prêt pour une performance.



La fenêtre d'orchestration permet d'accéder au contrôleur, à la configuration et à l'affichage d'un suivi graphique de la pièce (score).



## 4 UTILISATION DU CONTROLEUR

Le contrôleur est ouvert à partir de la fenêtre d'orchestration. Il donne une vision des groupes de l'audience et des groupes de patterns. Il permet aussi de contrôler en temps réel la *matrice des possibles* qui est affichée sous forme d'un tableau avec comme lignes les groupes dans l'audience tel que défini dans le fichier de configuration de la pièce et comme colonnes les groupes de patterns. Le contrôleur peut activer ou désactiver un groupe de patterns en cliquant dans la matrice. En cliquant sur le numéro du groupe de pattern on active ou désactive tous les groupes de l'audience pour ce groupe.

**Attention :** La correspondance entre les index du contrôleur et les numéros de groupe de patterns n'est valable que si les numéros de groupe se suivent dans le fichier de configuration de la pièce. Les index du contrôleur correspondent à la ligne dans le tableau des groupes de patterns et non à l'index du groupe. (à revoir, si possible).

Le bouton « ALL » active tous les groupes. Le bouton « RESET » désactivent tous les groupes. Le bouton « CLEANQ » vide toutes les files d'attente.

Le bouton « CHECK » affiche le fichier de configuration des patterns dans la console Skini.



Les afficheurs Groupes, Scrutateurs, FIFO donnent des états du système.

## 5 UTILISATION DU CONFIGURATEUR MIDI

Cet outil est utilisé pour paramétrer la DAW et le fichier de configuration des patterns, aussi appelé descripteur.

En général les DAW possèdent une fonction d'écoute des commandes MIDI venant de contrôleurs. Pour faciliter la mise en œuvre des pièces, Skini peut envoyer des commandes comme un contrôleur. Le configurateur MIDI se comporte donc comme un contrôleur MIDI. Skini

convertit les commandes « Clip » en canal et note MIDI. Le configurateur permet d'envoyer des notes avec « Envoyer Clip » et des « Control Changes (CC) » MIDI avec « Envoyer CC ». Le premier champ au-dessus de « Envoyer CC » est pour le CC, le champ à droite de ce dernier permet de donner une valeur au CC. On accède au configurateur depuis l'orchestration (ou avec [http://adresse\\_du\\_serveur/conf](http://adresse_du_serveur/conf)). La configuration des patterns est présentée au chapitre « Configuration des patterns ».

Skinini 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 UTILISATION DU CLIENT SKINI

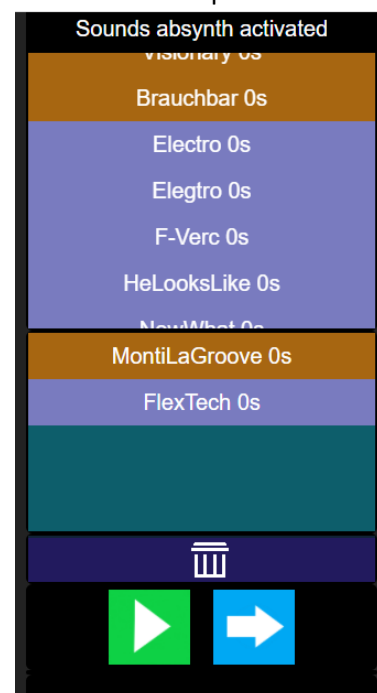
Le client qui s'appelle par l'URL [http://<adresse\\_serveur:port>/skini](http://<adresse_serveur:port>/skini) permet aux membres de l'audience de créer une liste de patterns et de l'envoyer au serveur. Le client peut alimenter une liste (partie inférieure de l'écran) à partir des listes de patterns disponibles (drag and drop). Le bouton vert joue la liste en local, le bouton bleu envoie la liste sur le serveur. La poubelle sert à supprimer des éléments de la liste de choix.

**Remarque :** Les patterns de la liste des choix vont remplir les files d'attente du serveur. Si dans une liste nous avons des patterns pour des instruments différents, se seront des files d'attente qui seront alimentées. On aura donc pas la séquence entendue en local entre les instruments. Les séquences sont respectées par instrument.

Ce client s'accompagne de fonctions d'orchestration permettant de définir dynamiquement la longueur de la liste et de vider la liste.

Quand une liste a été envoyée, le client ne pourra pas en envoyer une autre tant que sa liste demandée n'aura pas été jouée complètement.

Ce client couplé avec une orchestration adéquat permet de créer des jeux musicaux. Par exemple on peut imaginer de laisser à tour de rôles des groupes de clients concevoir des listes de patterns durant une période et noter la qualité de ces listes en donnant des gagnants et des perdants. Les mécanismes de « pause/resume » des files d'attente permettent de gérer



des périodes de conception et de jeu des patterns. On pourra coupler un affichage en conséquence sur un grand écran avec l'outil d'affichage de l'orchestration par exemple. Il est aussi possible de communiquer avec une plateforme de jeu vidéo comme Unreal Engine via des signaux pour des jeux musicaux plus riches en graphisme

**Remarque importante :** Ne pas utiliser les algorithmes de réorganisation des files d'attente avec ce client si ces algorithmes peuvent enlever des patterns des FIFOs. En effet, le client attend des confirmations sur le jeu des patterns pour autoriser l'envoi d'une nouvelle (ou de la même) liste. Si un pattern demandé disparaît le client sera bloqué.

Le bloc `cleanQueue` est à manipuler avec précaution. En effet, les clients sont bloqués tant que tous les patterns d'une liste n'ont pas été joués. Si les Fifo dont vidéos certains clients peuvent être bloqués puisque les patterns ont disparu des Fifo et ne seront jamais joués. Il vaut mieux faire « `cleanChoiceList` » (255) avec un « `cleanQueues` ».

Si les patterns sont typés le serveur évaluera un score selon la pertinence de la succession des types dans la liste définie par un client. L'algorithme de notation se trouve dans `websocketserver.js`, dans la fonction `computeScore()`. Les règles de notation sont à modifier dans le code source. Le mécanisme d'évaluation des scores repose sur les pseudos.

L'orchestration accède au gagnant en cours avec les blocs « `display score` ».

## 7 UTILISATION DU SIMULATEUR

Le simulateur était initialement un outil qui permettait de tester le comportement d'une pièce avant son utilisation avec une audience. Il permet aussi l'activation de patterns de façon aléatoire durant une performance avec audience. Il a donc deux comportements de base qui se définissent dans les paramètres de la pièce avec les champs :

Number of client groups  
Simulator in a separate Group

La définition du nombre de groupes de clients a pour but de donner le nombre de groupes de personnes dans l'audience que l'orchestration pourra gérer de façon indépendante. L'attribution d'un groupe à un membre de l'audience se fait de façon cyclique. Chaque membre se voit attribuer un groupe suivant son prédécesseur au moment de la connexion.

### 7.1 SPECIALISATION DU SIMULATEUR

Le paramètre `Simulator in a separate Group` quand il est coché, signifie que le dernier groupe client est réservé au simulateur. L'audience n'y aura pas accès. Sinon il signifie que le simulateur pourra se comporter comme d'importe quel groupe de l'audience.

### 7.2 ACTIVATION

Le simulateur d'audience se lance soit depuis la fenêtre de programmation blockly soit avec la commande

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

Le simulateur en dehors de l'audience sur le dernier « groupe de personnes » quand Simulator in a separate Group est coché, il se lance depuis la fenêtre principale ou avec la commande

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

Le simulateur comporte un mécanisme qui évite deux répétitions successives du même pattern sur trois sélections.

### 7.3 PARAMETRAGE

Le simulateur se paramètre dans le fichier de configuration de la pièce avec les lignes :

```
Tempo Max En ms  
Tempo Min En ms  
Limit Waiting Time (in pulse)
```

Chaque appel au serveur se fait à un instant défini par :

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

Il s'agit donc d'une durée aléatoire entre deux limites tempoMax et tempoMin.

Limit Waiting Time (limite de Duree d'Attente) est le paramètre qui définit la durée d'attente d'un pattern au-delà de laquelle le simulateur ne fera pas appel au serveur.

NB : Le simulateur comporte un mécanisme pour éviter la répétition d'un même pattern dans un historique de 3 précédents patterns. Il s'agit de la fonction selectRandomInList.

### 7.4 GESTION DES TYPES DE PATTERNS DANS LE SIMULATEUR

Il est possible de demander au simulateur de traiter les listes de patterns à solliciter. C'est-à-dire de leur appliquer un ordre défini selon une liste. Par exemple si nous avons 5 types de patterns (0,1,2,3,4), nous pouvons demander au simulateur d'envoyer à Skini des listes de patterns classer selon la série (2,3,0,4). Quand le simulateur recevra de Skini une liste de patterns, il en choisira au hasard dans cette liste en classant ses choix selon la série (2,3,0,4). Si un des types manque, la classification se fera sans le type manquant. Si le simulateur reçoit un ensemble de pattern ne comprenant aucun type 0, il appliquera un classement selon (2,3,4).

La fonction est activée depuis l'orchestration avec :

```
activate Type list in simulator
```

La série des types est définie par le bloc :

```
set type list for simulator " 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 "
```

Quand on utilise cette gestion des types la définition de « pattern list length » du bloc

```
set pattern list length to 7 for group 255
```

n'est pas pris en compte. La liste des patterns envoyée sera au maximum de la longueur de la liste des types définies.

**Attention** : Il ne faut pas utiliser la gestion des types dans le simulateur avec un algorithme de FIFO management de Skini (dans les paramètres « Algo Fifo managements » doit être à 0). Les deux techniques utilisent les *types des patterns* mais avec des usages différents qui peuvent être en conflit.

**Remarque** : la gestion des types horizontaux est plus efficace depuis le simulateur qu'avec l'algorithme de gestion des FIFO, car il est possible de gérer plus de 4 types. La différence sera liée au type de performance. L'algorithme d'organisation des FIFO fonctionnera en interaction avec une audience, pas celui du simulateur.

## 7.5 EXEMPLE D'UTILISATION DE LISTES DE TYPE : LE WÜRFELSPIEL

Nous avons utilisé la gestion des types pour mettre en œuvre un trio de Stadler, créé au début du XIXème siècle pour un würfelspiel. Nous avons créé les patterns de Stadler dans Ableton Live selon le modèle proposé par Stadler :

4 MIDI	5 MIDI	6 MIDI	7 MIDI	8 MIDI	9 MIDI
▶ T72	▶ T56	▶ T75	▶ T40	▶ T83	▶ T18
▶ T6	▶ T62	▶ T39	▶ T73	▶ T3	▶ T45
▶ T59	▶ T42	▶ T54	▶ T16	▶ T28	▶ T62
▶ T25	▶ T74	▶ T1	▶ T68	▶ T53	▶ T38
▶ T81	▶ T14	▶ T65	▶ T29	▶ T37	▶ T4
▶ T41	▶ T7	▶ T43	▶ T55	▶ T17	▶ T27
▶ T89	▶ T26	▶ T15	▶ T2	▶ T44	▶ T52
▶ T13	▶ T71	▶ T80	▶ T61	▶ T70	▶ T94
▶ T36	▶ T76	▶ T9	▶ T22	▶ T63	▶ T11
▶ T5	▶ T20	▶ T34	▶ T67	▶ T85	▶ T92
▶ T46	▶ T64	▶ T93	▶ T49	▶ T32	▶ T24
▶ T79	▶ T84	▶ T48	▶ T77	▶ T96	▶ T86
▶ T30	▶ T8	▶ T69	▶ T57	▶ T12	▶ T51
▶ T95	▶ T35	▶ T58	▶ T87	▶ T23	▶ T60
▶ T19	▶ T47	▶ T90	▶ T33	▶ T50	▶ T78
▶ T66	▶ T88	▶ T21	▶ T10	▶ T91	▶ T31

Nous avons défini 16 types numérotés de 0 à 15 qui correspondent chacun à une ligne de ce tableau. Pour chaque type nous avons 6 patterns. Pour chaque type nous définissons un tank avec les 6 patterns d'une ligne, par exemple pour le type 0 :



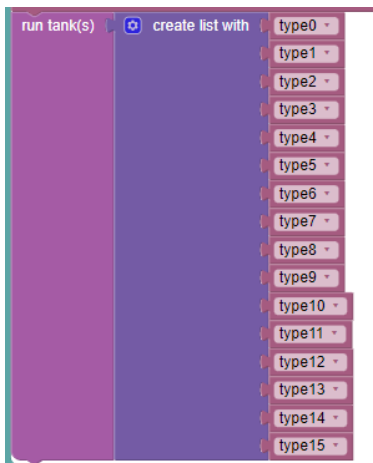
Le trio consiste à jouer successivement les patterns selon une liste de types qui correspond aux lignes du tableau. Nous la définissons ainsi :

set type list for simulator “ 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ”

Comme nous utilisons des tanks pour chaque ligne, aucun pattern n'est rejoué. La programmation Skini du würfelspiel consiste à faire correspondre :

- les champs **Group** du descripteur des patterns avec les champs **Index** des paramètres,
- les champs **Types** du descripteur des patterns avec les champs **Tanks nb** des paramètres,
- les champs **Text** du descripteur des patterns avec les champs **Group** des paramètres,

Puis simplement à activer tous les tanks et tient une instruction.



A la différence du würfelspiel de Stadler, il n'y aura aucune répétition entre les différentes séquences du trio.

**Remarque :** La mise en œuvre de ce würfelspiel est un exemple d'utilisation Skini pour la génération de musique tonale. La liste des types suit une structure harmonique, ici elle assez simple mais elle pourrait être enrichie ou modifiée en cours de pièce.

## 8 PROGRAMMATION GRAPHIQUE DES ORCHESTRATIONS

Pour la programmation des orchestrations le compositeur accède à une interface utilisant la solution *Blockly* fournit en open source par Google. La manipulation de Blockly est simple et conviviale. C'est cette interface qui est utilisée par Scratch l'outil d'apprentissage de la programmation pour les enfants.

Nous n'abordons pas ici le fonctionnement de Blockly. Un passage sur le site

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

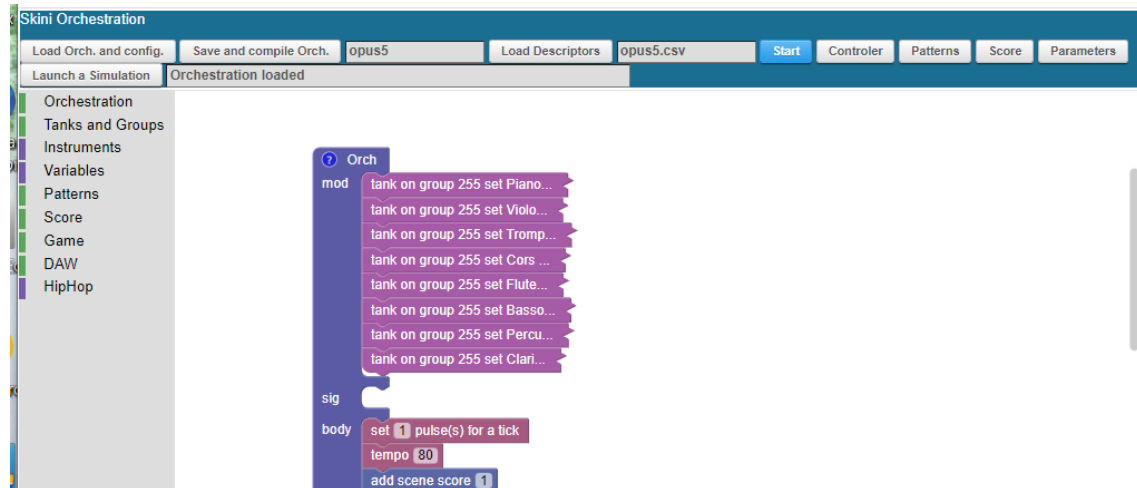
sera plus complet et efficace qu'une présentation générale dans le cadre de Skini.

Les programmes blockly ont une extension *xml*. Toute une série de tutoriels Skini avec blockly sont fourni dans le répertoire : *./pieces/tutos/*

Plusieurs exemples de programme plus complexes se trouvent dans les sous-répertoires du répertoire *pieces*.

## 8.1 L'INTERFACE GRAPHIQUE DE SKINI

L'interface Blockly génère des programmes HipHop.js sans avoir besoin de savoir programmer avec ce langage. Le compositeur peut charger une orchestration avec « Load Orch. and config ».



Le bouton « Save and Compile » fait deux choses. Il enregistre le fichier en cours dans un fichier *xml* avec le nom entré dans le champ texte. Il crée un fichier *HipHop.js*. « Start » permet de lancer l'orchestration.

« Launch a Simulation » permet de lancer un simulateur sans passer par une console.

« Patterns » ouvre la fenêtre de configuration des descripteurs de patterns. « Score » ouvre la fenêtre d'affichage des blocs de patterns et « Controler » la fenêtre de contrôle. « Parameters » ouvre la fenêtre des paramètres de la pièce.

**Remarque :** Le contrôleur donne plus d'information sur les groupes de patterns actifs et les ticks. Les informations sur les groupes sont visibles avec l'affichage de l'orchestration (client « score » de Skini), si ceci est prévu dans la configuration de la pièce.

Nous allons passer en revue les principaux blocs de l'interface Blockly.

**Remarque :** « Save and compile » compile Blockly dans un fichier *./myReact/orchestrationHH.js*. Ce fichier n'est pas utile pour le compositeur. Le chemin est fixé dans *websocketServer.js* avec la variable *generatedDir*.

## 8.2 LES BLOC D'ORCHESTRATION

Il s'agit des blocs définissant la structure de l'orchestration.

Le premier module est indispensable. « Orch ».



Le code blockly est organisé dans des blocs qui peuvent être mis en parallèle avec le bloc :



Le champ « mod » est utilisé pour la création de « Modules », comme les réservoirs que nous verrons ci-après, qui seront appelés dans le corps de l'orchestration (« Body »). « sig » permet de déclarer des signaux utilisés dans l'orchestration.

L'utilisation des modules avec des groupes ou des tanks fait appel à des signaux « implicites » c'est-à-dire qui n'apparaissent pas dans l'orchestration Blockly. A une activation de groupe correspond un signal xxxOUT où xxx est le nom du groupe et un signal venant d'un groupe xxxIN. Il y a aussi des signaux par défaut dans le programme principal qui ne sont pas par défaut dans les modules et qu'il faudra déclarer en entrée ou sortie dans les modules si besoin : *start*, *halt*, *DAWON*, *tick*, *patternSignal*, *controlFromVideo*, *pulsation*, *midiSignal*, *emptyQueueSignal*, *stopReservoir*, *stopMoveTempo*.

### 8.3 LES BLOCS HIPHOP

Ils correspondent aux commandes du langage réactif synchrone HipHop. Voir les tutoriels Skini pour une vision complète. Voici par exemple :



Par défaut les blocs sont exécutés les uns après les autres, mais pour des soucis de lisibilité ou lorsque que l'on souhaite mettre en parallèle des séquences d'instructions, il est possible de les regrouper avec le bloc « seq » :

L'intérêt de ce bloc est de pouvoir facilement activer la fonction « collapse block » de Blockly pour rendre l'orchestration plus synthétique.



Pour mettre des instructions en parallèle, il faut commencer par un bloc « fork » et placer à sa suite un ou plusieurs blocs « par ».

On peut enchâsser les uns dans les autres des blocs « seq », « fork » et « par ».

### 8.4 LA PROGRAMMATION DU TEMPS DANS UNE PIECE

La gestion du temps dans Skini a pour fonction d'assurer une bonne cohérence dans l'enchaînement des patterns et une grande souplesse de programmation. Elle se fait selon deux principaux paramètres :

- La *pulsation* qui est fournie soit par une synchronisation midi, soit Ableton Link ou bien directement pas Skini (cf. le paramétrage d'une pièce)
- Le *tick* qui est un multiple de la *pulsation* et qui est utilisé par le mécanisme de lecture (vidage) des files d'attente.



#### 8.4.1 Utilisation du signal tick

Le fait de définir un *tick* comme un multiple de la pulsation permet de définir le cycle sur lequel vont être synchronisés les débuts de pattern qui sont dans les files d'attente. Si nous fixons un *tick* à 4 pulsations, chaque pattern dans une file d'attente sera lancé au plus tôt au début d'un cycle de 4 pulsations. Au plus tôt car un pattern pouvant durer plus de 4 pulsations le suivant devra attendre la fin du précédent. Ce mécanisme permet d'éviter le chevauchement de patterns pour un même instrument.

On place donc un pattern dans une file à chaque fois qu'il est sélectionné et les files d'attente sont dépilés à intervalle régulier multiple de la pulsation que l'on appelle un *tick*. Cet intervalle est fixé avec :

set 1 pulse(s) for a tick

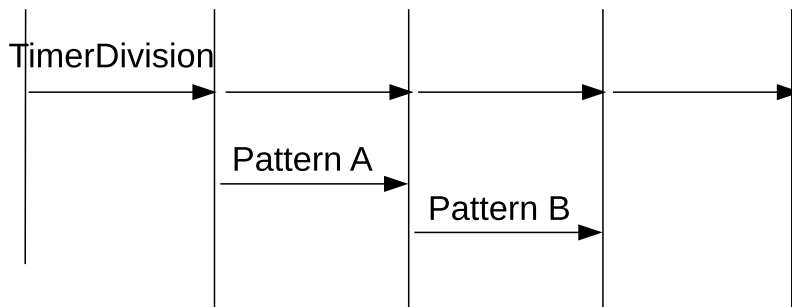
Pour des pièces avec des patterns de même durée, on peut donner au *tick* cette durée (un certain nombre de pulsations) ce qui assure que les départs de patterns se feront tous aux mêmes instants.

Comme évoqué ci-dessus, quand on souhaite introduire des patterns de durées différentes, il est possible de donner au *tick* une valeur correspondant à la durée du plus court pattern. Il faudra faire attention à la correspondance entre la durée du *ticks* et les durées de patterns. Si un pattern a une durée non multiple d'un *tick* Skini ne le jouera pas et donnera un message d'erreur. Si on fixe le *tick* à 4 pulsations et que l'on a un pattern à 3 pulsations ou à 5 pulsations il ne sera pas joué.

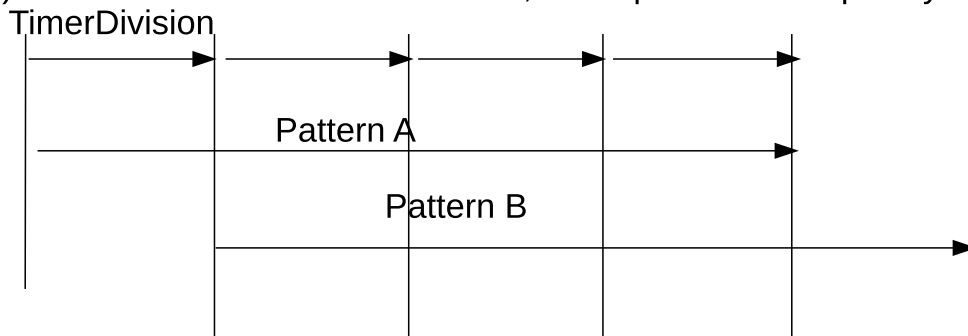
La combinaison de durée de *tick* et de pattern peut conduire à des comportements complexes. Prenons l'exemple où des patterns ont des durées multiples du *tick*. Prenons des patterns qui ont la même durée, par exemple 16 pulsations et un *tick* égal à 4 pulsations. Si un pattern A est sollicité (mis en file d'attente) à l'instant T et un pattern B à l'instant T + 5 pulsations par exemple (entre T + 5 et T + 8). Le pattern B démarrera un cycle de *tick* après le pattern A. Donc les patterns de 16 pulsations bien que tous de mêmes durées pourront être décalés les uns par rapport aux autres selon des multiples du *tick*.

Le schéma suivant illustre le mécanisme. Les patterns A et B sont sur deux instruments différents. *TimerDivision* est équivalent à *tick*.

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



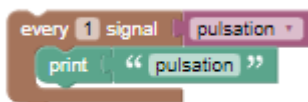
2) Pattern de durée != timerDivision, les départs ne sont pas synchronisés



Lors de l'utilisation de durées de patterns différentes il faut donc s'assurer de la cohérence musicale sur la durée du cycle du *tick*.

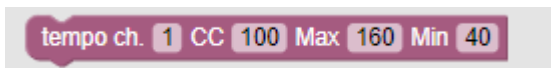
#### 8.4.2 Utilisation du signal pulsation

Dans une pièce il est possible de gérer le temps avec les pulsations. La pulsation correspond à la synchronisation MIDI pour l'équivalent d'une noire. Pour ceci il faut activer « Pulsation ON » dans les paramètres de la pièce. Il est alors possible de gérer un signal pulsation (déclaré par défaut). Voici par exemple l'affichage d'un message à chaque pulsation.

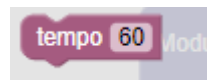


### 8.5 TEMPO

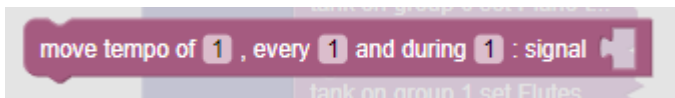
Quand on n'utilise pas la synchro Link, la modification des tempi depuis l'orchestration nécessite un paramétrage de la DAW qui lui permette de recevoir les Control Change sur le tempo. Nous avons vu qu'un bus MIDI de contrôle permettait d'envoyer les informations MIDI vers la DAW. C'est ce bus qui est utilisé pour les contrôles de tempo. Dans le cas d'Ableton, la fonction de contrôle nécessite de déclarer les paramètres utilisés par Live pour ce contrôle, c'est-à-dire une valeur max et une valeur min pour les tempi. C'est le bloc suivant qui fixe ces paramètres.



Une fois ces paramètres fixés ou directement quand on utilise Link. Le bloc

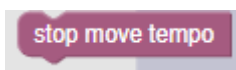


Permet d'intervenir sur le tempo à n'importe quel moment de l'orchestration. Le bloc



permet une automatisation du changement de tempo. Il permet de faire varier le tempo d'une valeur fixée à « every » occurrence du signal donnée en paramètre. La variation de tempo est inversée au bout de « during » occurrence de ce même signal. Ce bloc permet l'intégration facile de mouvement de tempo sans programmer.

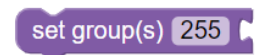
Pour interrompre les mouvements de tempo, il faut appliquer le bloc :



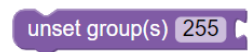
## 8.6 RESERVOIRS ET GROUPES DE PATTERNS

Bien qu'il s'agisse dans les deux cas de traitements d'ensemble de patterns il s'agit de deux classes de processus assez différents. Les groupes sont contrôlés à l'aide de signaux, les actions d'activation et de désactivation ne sont pas bloquantes.

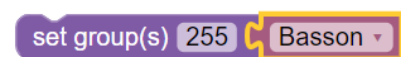
### 8.6.1 Groupes de patterns



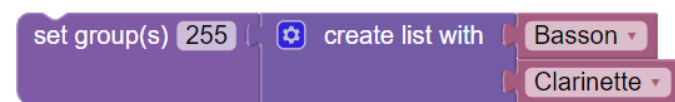
Les groupes d'utilisateurs sont définis dans le fichier de configuration de la pièce. Ils sont numérotés de 0 à 254. Le groupe 255 est en fait



l'ensemble des groupes. Le bloc suivant active le groupe Basson pour tous les d'utilisateurs :

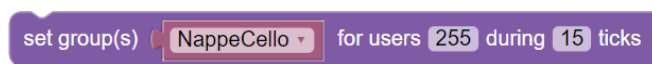


On peut activer plusieurs groupes avec des listes, ex :



### 8.6.2 Actions liées aux groupes

Skini fournit des blocs de haut niveau pour permettre des actions complexes avec des groupes.



active un ou plusieurs groupes durant un période. On peut utiliser des listes

au lieu d'un groupe seul.

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

Clarinette  
Basson

active un ou plusieurs groupes en attente de patterns joués dans un ou plusieurs groupes.

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

permet d'activer de façon aléatoire « max » groupes parmi un liste durant une période.

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

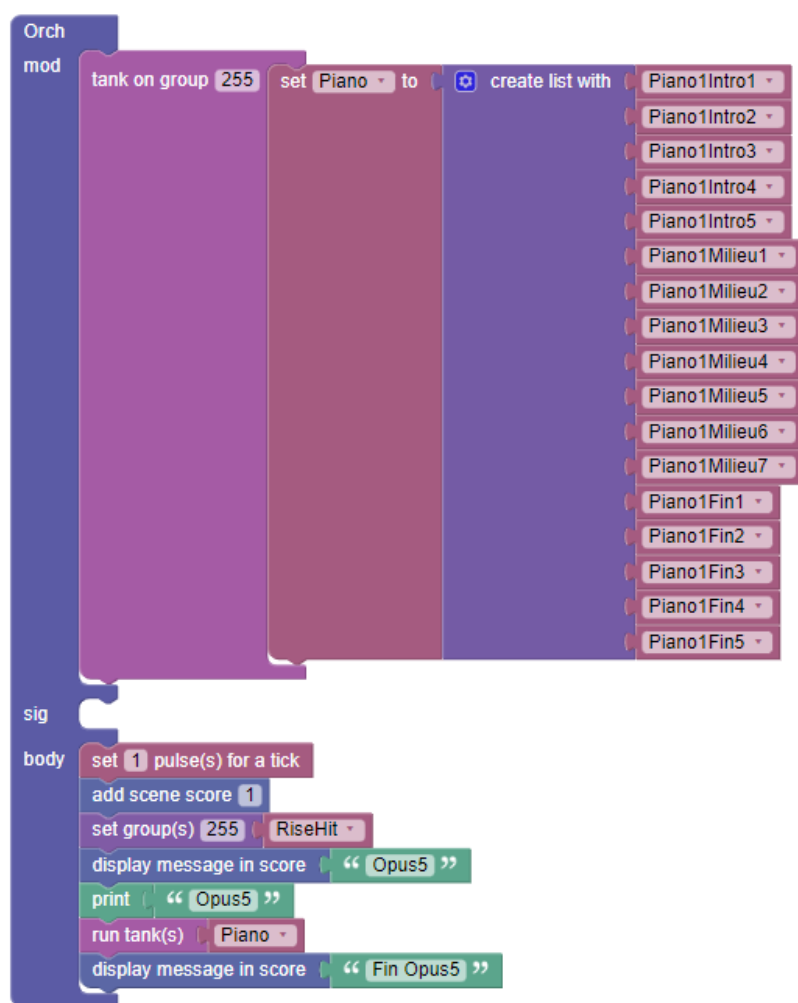
active un ou plusieurs groupe dans l'attente d'un ou plusieurs patterns spécifiques.

**Remarque :** Les groupes sont des variables Blockly , les patterns ici des « strings » Blockly.

### 8.6.3 Création des réservoirs

Un réservoir est un module à trois niveaux qui se place donc dans le champ « Mod. » De l'orchestration. Tout d'abord une liste de patterns, puis une variable associée à cette liste. La variable est mise dans un « tank ». Un réservoir est associé à un groupe d'utilisateur ou tous les utilisateurs. Voici un exemple de réservoir « percussion » de patterns de percussions assigné à tous les groupes d'utilisateur (255). Les patterns sont des variables ou des chaînes de caractères contenant le nom des patterns tels que décrit dans le fichier csv des patterns.

Voici un exemple d'utilisation de réservoir :



Avec les 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	Prokofiev	Piano15	1	0	2	0	26	4

Et les groupes :

	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 sur les réservoirs

Les réservoirs (tanks en anglais) sont en fait des sous-modules Skini ayant pour paramètres des patterns. Ils sont bloquants dans un déroulement. Un réservoir s'arrêtera une fois qu'il sera vide ou qu'on l'aura tué avec

stop all tanks

Il existe des fonctions simplifiant la gestion des réservoirs :

run tank(s) during 1 ticks

Ce bloc lancera un ou plusieurs réservoirs donnés en paramètre sous forme de variable Blockly et « tuera » ce réservoir au bout de « during » occurrences d'un signal. Le bloc suivant laissera les patterns du réservoir Basson disponibles pendant une durée maximale de 40 tick :

run tank(s) Basson during 40 ticks

On comprend de façon intuitive le fonctionnement des blocs suivants :

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

ce bloc permet de prendre en compte le jeu de patterns spécifiques par la DAW.

**Remarque :** Les noms des groupes et des réservoirs sont en correspondance directe avec le fichier de configuration de la pièce de musique.

**Remarques :** Tous les traitements des **groupes** et des **réservoirs** utilisent des **variables Blockly**. Tout ce qui concerne directement des **patterns** utilise des **chaines de caractère** (strings).

reset orchestration

Réinitialise la matrice de l'orchestration, tous les groupes actifs sont désactivés.

**Remarque sur l'utilisation des réservoirs avec le paramètre *react on play* activé :** On rappelle, que le paramètre *react on play* actif signifie qu'un signal concernant la commande d'exécution d'un pattern est émis au moment où le pattern est joué et non pas au moment où la commande est mise en file d'attente. Ceci a un impact sur la façon de gérer le temps dans les orchestrations et ceci a un impact sur le fonctionnement des réservoirs. Pour qu'un réservoir soit pleinement effectif, dans le sens où un pattern ne peut pas être commandé 2 fois, *react on play* doit être inactif. En effet si la réaction doit se produire au moment du lancement d'un pattern et que la commande a été placée dans une file où se trouvent déjà des commandes, il se peut que ce même pattern soit sollicité plusieurs fois, car il n'aura pas été désactivé (retiré des patterns disponibles pour l'audience ou le simulateur) immédiatement. Donc un fonctionnement sûr des réservoirs implique de désactiver *react on play*.

## 8.7 LES FILES D'ATTENTES DES INSTRUMENTS

Les files d'attente sont associées aux instruments. Elles sont créées en fonction de la description des patterns dans le fichier csv des patterns. Elles sont remplies par des actions externes à la plateforme Skini soit par une interaction avec un public, soit par des simulateurs qui génèrent des interactions plus ou moins aléatoires.

### 8.7.1 Vidage des files d'attentes

Ces blocs sont utiles lorsque le compositeur souhaite que l'on entende plus aucun pattern en attente pour un instrument. Il peut vider toutes les files avec :

clean all instruments

Le client skini vide sa liste de choix dès qu'un clean all queues est réalisé. Il peut vider la file d'un instrument particulier en donnant son index avec :

clean instrument 1

Le client skini vide sa liste de choix dès qu'un clean queue est réalisé. Il ne tient pas compte du n° de la FIFO, comme le fait `controlAbleton.js`. (Le block ne réalise pas de `cleanChoiceList`).

**Attention** : les files d’attente, comme les groupes dans l’audience sont paramétrées avec des **index** pouvant commencer à partir de **0**. Les affichages de score se font avec des paramètres qui commence à 1. (Le meilleur score est en 1).

### 8.7.2 Mise en pause et test des files d’attente

pause all instruments

Arrête le jeu des patterns pour tous les instruments.

resume all instruments

Reprend le jeu des patterns pour tous les instruments.

pause instrument 1

Arrête le jeu des patterns pour un instrument.

resume instrument 1

Reprend le jeu des patterns pour un instrument.

wait until instrument 1 is empty

Bloque l’orchestration en attendant que les patterns d’un instrument aient été joués.

### 8.7.3 Mettre un pattern spécifique en files d’attente

Le compositeur peut faire un jouer un pattern spécifique de façon impérative avec le bloc.

put pattern in instrument

Le paramètre du bloc est une chaîne de caractère avec le nom du pattern.

Ex :

put pattern BassonDebut1 in instrument


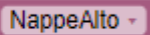
Ce bloc permet d’introduire des éléments de type « séquenceur » dans une orchestration de musique collaborative ou générative.

**Attention** : « put pattern » est sensible à la synchronisation. Bien que Skini considère que les actions système se font en temps nul, ce n’est en réalité pas le cas. Or si dans un instant il faut recevoir la synchronisation, mettre en pattern dans la FIFO, lire la FIFO et envoyer une commande sur la DAW, skini peut se trouver dans une situation où la commande de la DAW ne passe pas au bon tick de synchronisation. La solution est d’introduire un délai au lancement du pattern par la DAW (launch synchronisation). Dans ce cas la DAW lance la commande après un court délai qui permet la réception de la commande en toute sécurité. La DAW envoie donc un tick de synchronisation et attend un peu avant de déclencher les patterns.



## 8.8 PATTERNS

Skini étant principalement destiné au traitement de groupes de patterns et de réservoirs, il y a peu de fonctions dédiées aux patterns, en voici deux :

Ce bloc va attendre l'exécution (ou la demande selon le paramétrage de la pièce) d'un certain nombre de pattern appartenant à un groupe.

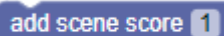


Est un bloc qui attend l'exécution (ou la demande selon le paramétrage de la pièce) d'un certain pattern passé en paramètre sous forme de chaîne de caractère. Par exemple :

## 8.9 AFFICHAGE DE L'ORCHESTRATION

L'affichage sur un grand écran du déroulement de l'orchestration est contrôlé par l'orchestration avec des blocs spécifiques :

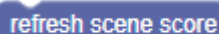


Un affichage peut se faire sur plusieurs niveaux. Ce bloc affiche un des niveaux. Les niveaux sont associés aux groupes de patterns dans le fichier de configuration de la pièce.

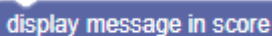


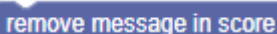
Fait disparaître un niveau

Le bloc suivant permet de rafraîchir l'affichage pour donner suite à des opérations sur les files d'attentes par exemple :



Les blocs suivants permettent d'afficher et de faire disparaître un message dans une fenêtre « popup » sur le grand écran :





## 8.10 JEU SKINI

Skini propose une interface standard pour l'audience qui permet aux participants de créer des listes de patterns et de les envoyer au serveur pour être jouées. Le compositeur peut à partir de cette interface inventer des jeux, comme « trouver la bonne séquence de patterns au sein d'un groupe ». Les scénarios de jeu ne sont pas programmés par l'orchestration. Ce sont des fonctions JavaScript sur le serveur.

Un modèle de base est proposé qui consiste à définir des types de patterns et associer des notes à la façon dont le participant va organiser ses listes. Le gagnant sera celui qui aura accumulé les meilleures listes durant la pièce.

Ce qui est possible depuis l'orchestration est d'agir sur les listes des participants en définissant les longueurs des listes et en les vidant impérativement les listes d'un groupe.

set pattern list length to 3 for group 255

clean choice list for group 255

Le bloc suivant permet d'afficher le meilleur score en cours :

display best score during 2 ticks

Pour afficher les autres scores (et le meilleur) on utilise le bloc

display score of rank 1 during 2 ticks

Le « ranking » se compte à partir de 1.

display group score of rank 1 during 2 ticks

set scoring policy 1 Définit le type d'algorithme qui va calculer les scores. Ces algorithmes se trouve dans le fichier ./serveur/computeScore.js

set scoring class 1 Définit la class prise en compte dans le calcul du score. La *classe* est un des paramètre (type en index 7) des patterns dans le fichier csv de description des patterns. Voir le fichier ./serveur/computeScore.js sur le traitement des classes.

## 8.11 COMMANDE ET CONTROL CHANGE MIDI

Il est possible d'émettre des « Control Changes MIDI » (CC) depuis l'orchestration. C'est au compositeur de définir et de paramétrer les fonctions des CC dans la DAW. Il faut définir le canal sur lequel les CC seront envoyés directement dans la commande.

sendCC ch. 1 CC 0 val. 0

sendMidi ch. 1 note 13 vel. 123

Ce bloc permet d'envoyer une commandes MIDI

Le choix du canal MIDI dépend de la façon dont ont été programmé les affectations MIDI aux clips dans la DAW. **Il y a une différence en le canal vu de Skini qui commence à 1 et sur les DAW où il peut commencer à 0.** Cette différence ne demande pas d'attention en dehors de ces commandes. Ne pas hésiter à faire un test et comparer l'activation « sendMIDI » avec l'outil de configuration par exemple.

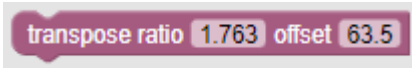
**Remarques sur le contrôle MIDI :** La définition du bus MIDI (port MIDI) associé au Block est fixée dans le fichier de configuration. Les blocks ne font donc jamais appel à ce paramètre.

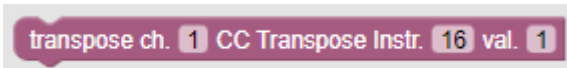
**Remarques sur les commandes MIDI :** De façon générale, les opérations réalisées par les blocs Skini n'interviennent pas sur les notes MIDI (au sens Skini) mais sur les groupes de patterns, les réservoirs et même les patterns mais à l'aide de leurs noms. Ceci est vrai sauf pour le blocs sendMIDI et sendCC qui s'adressent directement à un clip dans la DAW.

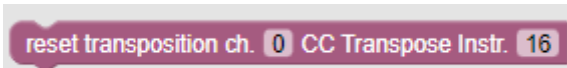
Le compositeur devra donc faire attention aux canaux MIDI dans le cas de blocs agissant sur des Control Changes (CC) et les commandes MIDI.

## 8.12 SPECIFIQUE ABLETON LIVE

Ces blocs sont utilisés pour commander l'outil de transposition d'Ableton Live. Les ratio et offset sont à calculer en fonction de cet outil.

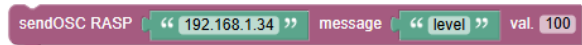
 Définit les ratio et offset pour le calcul de la transposition.

 Bloc de la transposition, les valeurs sont en demi-tons. Il faut donner le numéro du canal MIDI. La commande CC est celle qui contrôle l'outil de transposition d'Ableton.

 Bloc pour remettre à zéro la transposition. La commande précédente réalise un incrément.

## 8.13 ENVOI DE COMMANDES OSC VERS RASPBERRY

Pour envoyer une commande OSC depuis l'orchestration vers un Raspberry il faut utiliser le bloc sendOSC du menu DAW. Voici un exemple :



Qui envoie le message OSC `/level` avec la valeur 100 à l'adresse 192.168.1.34 sur le port « Raspberry OSC Port » des paramètres. Dans la version actuelle on ne passe qu'une valeur associée à la commande.

NB : Il y a la possibilité d'envoyer des commandes OSC à d'autre équipement que des Raspberries via le mécanisme décrit dans le chapitre *Interface OSC*. (Il serait judicieux de faire converger les deux processus.)

## 8.14 LES BLOCS AVANCES

Il est possible de créer des orchestrations complexes sans connaître les détails de HipHop, mais pour des fonctions basées sur des blocs non standards il existe une série de menus permettant

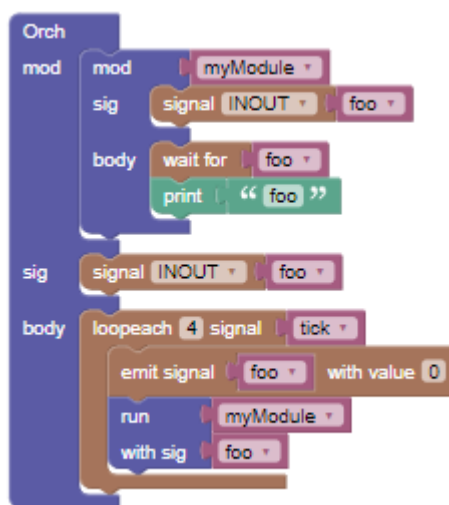
de programmer en HipHop avec Blockly. Ceci suppose de posséder les bases de la programmation réactive synchrone.

Ces menus sont : Signals, Signals Advanced et Module Advanced.

Au-delà des réservoirs, il est possible de créer des sous module Skini.



Ces sous modules comportent des signaux en entrée/sortie qu'il faudra créer dans le champ signal. Le corps du sous-module est dans body. Voici un exemple de sous-module :



Un sous-module est exécuté avec la commande run.

## 8.15 COMPOSITION DE PIECES POLYPHONIQUES

Le mode de fonctionnement standard de Skini permet de contrôler l'enchaînement séquentiels des patterns joués de façon aléatoire dans des groupes et pour un instrument. Cependant il n'y a pas de prise en compte de ce que contiennent les files d'attente des autres instruments. Il n'est donc pas possible de contrôler une polyphonie autrement qu'en activant des groupes de patterns « globalement compatibles » entre eux selon le projet musical. Ceci rend la composition de pièce tonale difficile (sans jouer sur les transpositions et modes) et contrapuntique quasiment impossible.

Pour aller plus loin dans la compatibilité des patterns et permettre la composition de pièces polyphoniques ou tonales, il existe un algorithme permettant de contrôler la compatibilité « verticale » des patterns, au sens de la verticalité d'une partition. Dans les paramètres, il suffira de mettre « Algo Fifo management » avec la valeur 2. La gestion des files d'attente actuelles est alors complétée par un mécanisme qui prend en compte la compatibilité des clips selon les règles suivantes :

- Dans les instruments, les patterns sont en correspondance selon leur index. C'est à dire que si un pattern de type « n » est présent à l'index « i » d'un instrument A et qu'il s'agit

d'ajouter un pattern de type « n » à un autre instrument B, il se trouvera à l'index « i » de l'instrument B. Nous appelons ce processus « association ». Ceci revient visuellement à avoir les patterns en correspondance verticale.

- Nous définissons le type « 0 » comme pouvant être associé à n'importe quel type.
- On définit un pattern vide à un index « 0 » et un type « 0 » pour pouvoir compléter les files d'attente si besoin.
- Les instruments étant vides au départ, l'ajout d'un pattern en index « i » d'un instrument nécessite d'introduire des patterns « vides » pour les éléments indéfinis qui le précèdent dans la file d'attente de l'instrument.
- Si un pattern « y » doit être ajouté à un instrument A et qu'à l'index « j » d'un autre instrument B se trouve un pattern du même type, on n'ajoutera le pattern « y » en index « j » que s'il n'y a rien à cet index ou qu'il s'y trouve un pattern de type « 0 » dont le pattern à insérer prendra la place.
- Si pour un pattern, aucune position ne permet une association, le pattern sera placé après le dernier pattern de tous les instruments.
- Il suffira de trouver une seule association pour insérer un pattern.

Ceci pose quelques contraintes :

- Tous les clips doivent avoir la même longueur. Il sera peut-être possible d'utiliser à terme des clips de longueurs différentes, mais ça compliquera fortement le mécanisme de réorganisation dynamique des files d'attente. (Des longueurs différentes reviendraient à avoir des mesures différentes et variables sur chaque voix) ; en contrepartie il n'y a pas de contrainte sur la longueur. Des longueurs courtes combinées avec des types séquentiels permettent des combinaisons complexes.
- La description de l'algorithme est donnée ci-dessous. Mais il faut garder en tête que la base de la production musicale de Skini repose sur un processus aléatoire. La compatibilité n'est pas ici déterministe. Si l'on pense en contrechant comme pour de la musique écrite, on aura plusieurs écueils possibles : le contrechant ne sera pas forcément un contrechant. La compatibilité ne définit pas de hiérarchie entre les patterns ; ce que l'on imagine comme un contrechant peut être perçu comme un élément mélodique à part entière s'il n'y a pas de pattern compatible en piste.
- La façon d'organiser les patterns en types séquentiels et types verticaux sera déterminante sur le rendu. Par exemple, le nombre de patterns dans chaque catégorie va jouer sur le déroulement rendu. Peu de patterns compatibles devrait donner une polyphonie sobre. Il n'y a pas de bon ou mauvais choix à ce niveau, tout dépendra du but recherché.
- L'algorithme fonctionne sur des files contenant des patterns à un instant précis. Il y a donc une correspondance entre la vitesse où seront remplies les files et la pertinence des combinaisons. Autrement dit, l'algorithme fera sens s'il y a suffisamment de patterns en attente. Pour de la musique générative, la fréquence d'émission d'un simulateur/générateur est un paramètre important sur le rendu.

## 8.16 CONTROLE INTERFACE Z

Skini intègre une carte **16 entrées (8ana – 8num) réseau OSC** et une carte **MiniWi** d'Interface Z. Ces cartes émettent des messages OSC en fonction des capteurs connectés. Elles fonctionnent sur un réseau internet filaire pour la 8ana-8num et en Wifi pour la MiniWi. L'interface Skini prend en compte les 8 entrées analogiques de la 8ana-8num et les 4 entrées de la MiniWi soit 12 capteurs possibles.

La configuration de la carte se fait dans *ipConfig.json* avec :

- *interfaceZIPaddress*, adresse IP de la carte InterfaceZ 8ana-8num.
- *portOSCFromInterfaceZData*, pour le port qui envoie les commandes issues des capteurs analogique et numérique de la 8ana-8num
- *portOSCFromInterfaceZMidi*, pour le port qui envoie les commandes issues de l'interface MIDI de la carte 8ana-8num (fonction pas utilisée pour le moment).
- *portOSCToInterfaceZ*, pour le port sur la carte 8ana-8num qui reçoit des commandes MIDI de Skini.
- *portOSCFromInterfaceZMiniWi*, pour la réception des messages OSC de la MiniWi.

Exemple de configuration : L'interface est programmée en 192.168.1.250 avec port data en 3005 et port MIDI en 3006. L'installation fonctionne en filaire sur un switch avec l'adresse IP 192.168.1.251 pour le PC.

Le paramétrage des capteurs se fait dans la fenêtre paramètre :

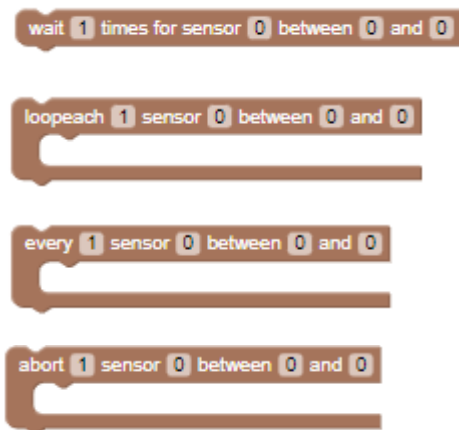
☒ Sensor OSC

Sensors Temporisation	5	0	12	0	0	0	0	0
Sensors Sensibilities	5	5	102	5	5	5	5	5

La temporisation définit la fréquence à laquelle on souhaite prendre en compte les informations envoyées par les capteurs. En effet, la carte Interface Z 8ana-8num envoie des données de façon systématique l'état des capteurs à une fréquence fixée par un potentiomètre qui permet de faire varier les émissions de 100 à 3000 messages par seconde. Pour ne pas ralentir les temps de réponse de Skini, un *thread* est dédié au traitement de ces messages. Les *temporisations* fixent pour chaque capteur la fréquence de prise en compte des messages. 15 signifie que l'on ne considère qu'un message sur 15. Par exemple pour 100 messages par seconde de la carte, on aura un signal toutes les 150 ms vers Skini au lieu de toutes les 10ms.

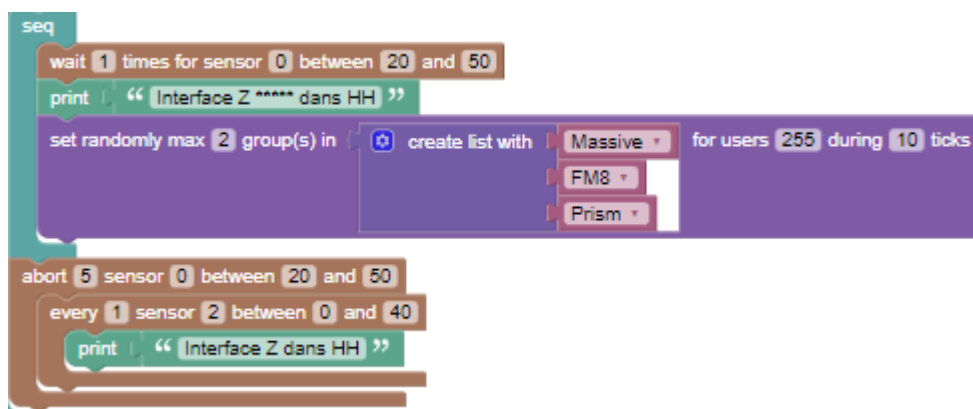
La sensibilité permet de prendre en compte les variations permanentes des capteurs analogiques. Si la variation (en plus ou en moins) avec la valeur prise en compte à la temporisation précédente est inférieure à la sensibilité on n'émet pas de signal. Ici encore il s'agit de ne pas surcharger Skini avec des informations de signaux non pertinentes.

Skini propose une série de blocs pour gérer spécifiquement des informations des capteurs Interface Z. Ce sont des blocs dédiés aux signaux :



Ils ont la même structure. *sensor* est le numéro du capteur sur les cartes Interface Z. Les sensors de 0 à 7 sont ceux de la carte 8ana-8num et de 8 à 11 ceux de la carte MiniWi. Comme les valeurs fournies par les capteurs analogiques sont fluctuantes les signaux sont pris en compte selon une plage entre un minimum et un maximum.

Exemple de programmation :



### Commande à revoir, plus opérationnelle (1/9/2022)

Cette commande est utilisée pour envoyer une commande MIDI via l'interface OSC d'interface Z. Il s'agit d'un cas d'usage très spécifique.



## 8.17 PROGRAMMATION DES TRANSITIONS « STINGERS »

Le principe est d'associer à un groupe de patterns (ou un pattern qui est un singleton) un pattern de transition. Dans le monde de l'audio pour jeu vidéo, on parle de *stinger*. Idéalement un *stinger* devrait être associé au passage spécifique d'un groupe de patterns A à un groupe de patterns B. La difficulté dans Skini est que l'automate n'a pas de vision sur la façon dont les FIFO sont remplies. L'automate ne sait pas repérer la séquence « temporelle » de deux patterns dans deux FIFO différentes.

Dans le scénario de *réaction à la sélection*, l'automate voit comment les FIFO se remplissent, mais il n'y a rien de prévu pour en gérer l'historique. (Il faudrait créer une sorte de doublon des FIFO en HH ou imaginer des signaux venant des FIFO vers HH). Cette gestion pourrait même être assez compliquée si les patterns ont des durées variables.

Dans le scénario de *réaction à l'exécution*, on n'aurait besoin d'une vision des patterns en attente dans les FIFO pour repérer le bon successeur à A. C'est-à-dire un B en position d'attente d'au moins de la durée de A dans sa FIFO.

En fait Skini est fait pour gérer les processus avec lesquels les FIFO se remplissent, mais pas sur leur organisation, excepté dans le cas de la gestion des priorités dans les files d'attente expliquée au chapitre *Priorité dans les files d'attente*, p.41.

Dans un premier temps nous allons simplifier le problème en traitant le cas où à un groupe de pattern A nous voulons associer un ou des *stingers* à chaque pattern. Dans ce cas, il suffit d'attendre le signal d'exécution d'un pattern de A et de lancer en décalage un pattern S, ou de lancer un *stinger* qui intègre le décalage. On sait donc facilement faire des *stingers* « en sortie ».

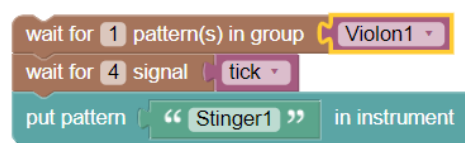
#### 8.17.1 Cas de la réaction à l'exécution

Ce scénario est assez simple à réaliser dans le cas des *réactions à l'exécution*, c'est-à-dire au moment où le pattern est lancé dans la DAW. Pour gérer le décalage de S, il faut fixer un *tick* qui permette de prendre en compte un décalage pour le lancement du stinger. Notons que *tick* devra donc être au minimum de la durée de ce décalage (cf. La programmation du temps dans , p.24).

Un *stinger* pourra être programmé sur le schéma :

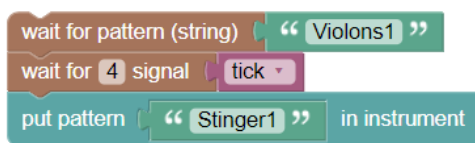
1. Wait for *patternAIN*
2. Wait for *X tick*
3. Put *pattern Stinger* in instrument!

Un exemple pour des patterns de violon de 8 ticks avec un stinger se lançant au 4ème *tick*. Ici le *tick* vaudrait une pulsation.



#### 8.17.2 Cas du retour de jeu de pattern depuis la DAW

La logique est la même, mais le principe du stinger est dépendant d'un pattern en particulier. Ce cas fonctionne même quand la réaction se fait à *la sélection* car le signal du pattern est émis par la DAW au moment où le pattern est lancé.





**Remarque :** Plutôt que de compter des *ticks*, on peut dans la conception du *stinger* prévoir un silence initial pour décaler le son. On n'a pas de contrainte sur la durée des patterns.

## 8.18 PRIORITE DANS LES FILES D'ATTENTE

**Attention : Ne pas utiliser d'algorithme de modification des FIFOs en musique interactive si l'algorithme peut supprimer des patterns des files d'attente FIFO, ceci crée des situations de blocage sur les clients qui attendent un retour sur le jeu des patterns demandés.**

Cette fonction est utilisable pour de la musique générative. Elle applique un traitement sur les files d'attente (FIFO) selon les types (horizontaux) définis dans les paramètres.

La gestion des types horizontaux peut se faire d'une façon plus complète à partir du simulateur. Les deux méthodes sont différentes dans leur logique.

La gestion des priorités (des types horizontaux) ici n'est pas compatible avec la gestion des types verticaux, contrairement à la gestion des types horizontaux depuis le simulateur. Elle a une existence historique, **nous conseillons d'utiliser plutôt la gestion des types horizontaux sur le simulateur** qui est compatible avec la gestion des types verticaux.

Donc ce qui fonctionne :

- Gestion des types horizontaux sur le simulateur (fonction « set type list for simulator »)
- Avec Algo Fifo management à 2

Sinon :

- Algo Fifo management à 1 seul.

(la valeur 1 assigné permet d'envisager que l'on peut développer plusieurs types d'algorithmes et leur donner des identifiants différents).

Il faut que pour chaque pattern dans le fichier csv on définisse un type au pattern à l'index 7 de chaque ligne. On définit cinq types de pattern. D : Début, M : Milieu, F : Fin. N : neutre (sans traitement) et P : Pain (un pain = mauvais pattern). Le type est déclaré par une valeur numérique dans le fichier de configuration csv des patterns : 1 pour D, 2 pour M, 3 pour fin et 4 pour neutre, 5 pour « pain ».

Pour améliorer la structure des phrases musicales on regarde l'état d'une file d'attente avant d'y ajouter un pattern. Comme on écrit dans une FIFO en ajoutant un dernier élément et lie en retirant le premier, on balaye une FIFO en commençant par la fin (dernier ajout) pour intervenir sur les derniers pattern mis dans la FIFO, les plus récents donc. Voici l'algorithme en place :

A) Pour ajouter un pattern F dans une file :

1. Si dans la file il y a 2 D qui se suivent, on insère F entre les deux.
2. Si dans la file il y a 2 M qui se suivent, on insère F entre les deux.
3. Si le dernier élément de la queue est déjà un F, on cherche un D qui soit suivi d'un M, si on en trouve un on met le F à empiler juste après ce D.
4. Sinon on empile F (ce qui donne deux F d'affilée)

On peut donc avoir des suites de F

B) Pour ajouter un pattern D dans une file :

1. Si dans la file il y a un F sans D avant on insère D avant ce F.
2. Si dans la file il y a un M sans D avant on insère D avant ce M.
3. Sinon on empile D (ce qui donne deux D d'affilée)

On peut donc avoir des suites de D

C) Pour ajouter un pattern M dans une file :

1. Si dans la file il y a un D immédiatement suivi d'un F on met M entre les deux.
2. Si dans la file il y a 2 D qui se suivent on met M entre les deux.
3. Si dans la file il y a 2 F qui se suivent on met M entre les deux.
4. Sinon on empile M (ce qui donne deux M d'affilée, mais n'est pas un problème).

Si l'on souhaite ne pas avoir de suite de D ou de F, il ne faut rien faire dans les cas 1.4 et B.3. C'est possible en musique générative pas en musique interactive.

D) Pour un pattern N on ne fait pas de traitement.

Voir dans le fichier ./serveur/controleDAW.js la fonction `ordonneFifo()` pour avoir le détail de l'algorithme effectivement en place.

#### 8.18.1 Exemple de programmation d'un jeu : trouve la percu

Le but du jeu est que les joueurs alternativement choisissent des patterns correspondant à une ambiance sonore. Les bons choix incrémentent un score, les mauvais le décrémentent.

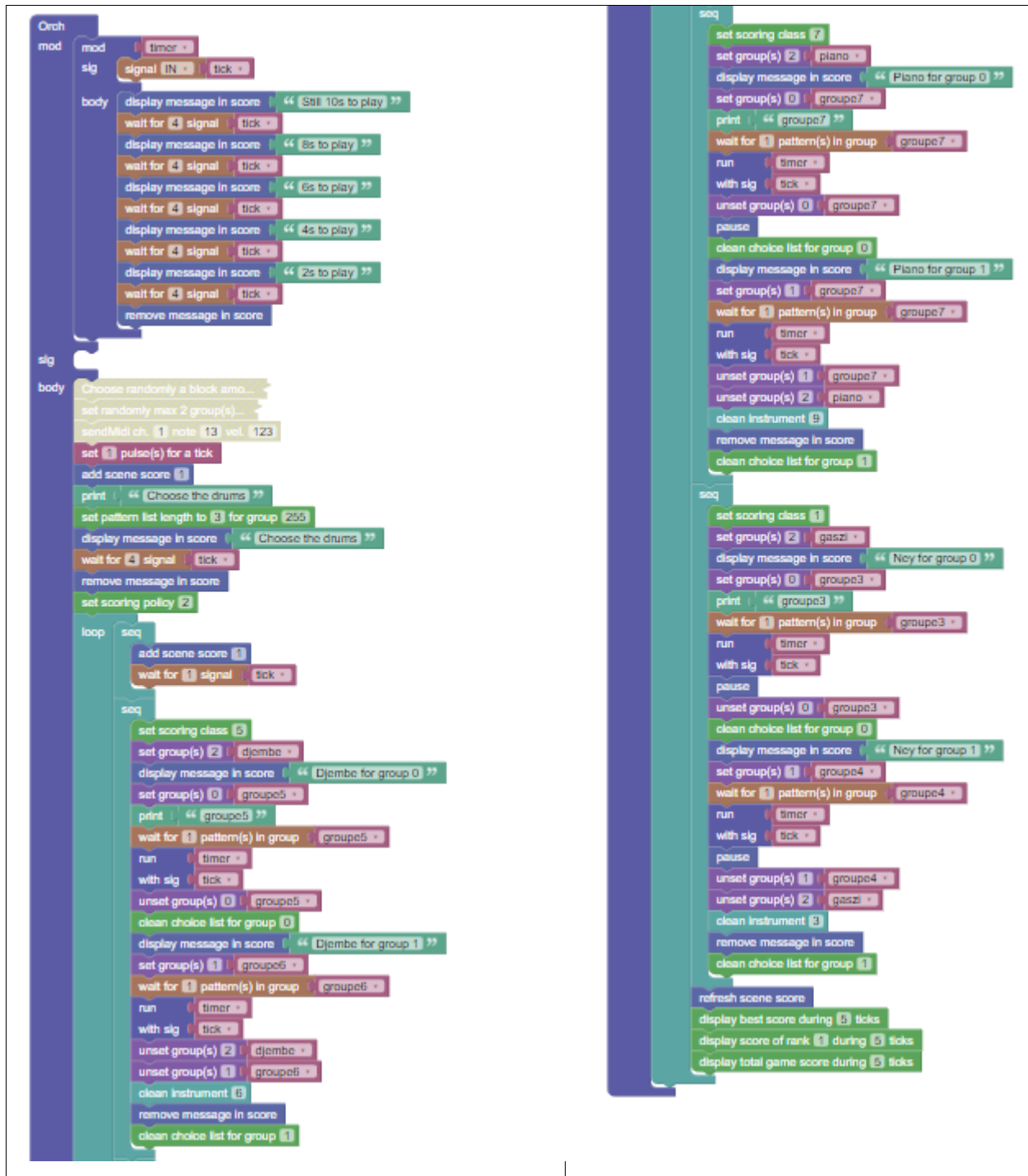
Les patterns sont utilisés uniquement sous forme de groupes. Le simulateur est utilisé sur un groupe dédié qui va jouer l'ambiance musicale pour laquelle le joueur doit trouver les patterns qui correspondent. Ici nous avons les groupes de joueurs 0 et 1. Le groupe d'utilisateur 2 est celui qui va jouer les ambiances sonores. Pour ceci « Simulator in a separate group » est activé dans les paramètres.

Pour les patterns nous avons les ambiances sonores dans les groupes de patterns 7 à 10. Les autres groupes sont pour les joueurs. Ces groupes associent des patterns de types différents parmi lesquels le joueur devra choisir les bons. Il y a autant de patterns correspondant à l'ambiance que de patterns ne correspondant pas à l'ambiance. Un joueur au hasard à 50% de chance de réussite.

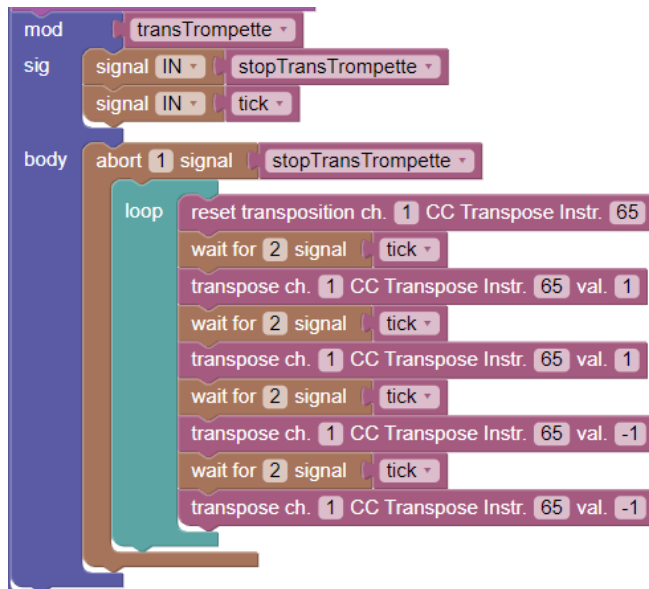
	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupe1	0	group	170	100	20	#CF1919		1
2	groupe2	1	group	20	240	20	#008CBA		1
3	groupe3	2	group	170	580	20	#4CAF50		1
4	groupe4	3	group	350	100	20	#5F6262		1
5	groupe5	4	group	20	380	20	#797bbf		1
6	groupe6	5	group	350	580	20	#008CBA		1
7	groupe7	6	group	540	100	20	#E0095F		1
8	derwish	7	group	740	480	20	#E0095F		1
9	gaszi	8	group	540	580	20	#E0095F		1
10	djembe	9	group	740	200	20	#E0095F		1
11	piano	10	group	740	340	20	#E0095F		1

La longueur maximale de la liste pour les joueurs est de 3 patterns.

Quand un joueur envoie une première liste d'au maximum trois patterns, il ne lui sera possible de jouer que pendant 10 secondes supplémentaires. C'est le timer en module qui affiche le temps restant dans la fenêtre « score ». A la fin du jeu on donne le vainqueur et les scores.



### 8.18.2 Exemple de transposition en boucle

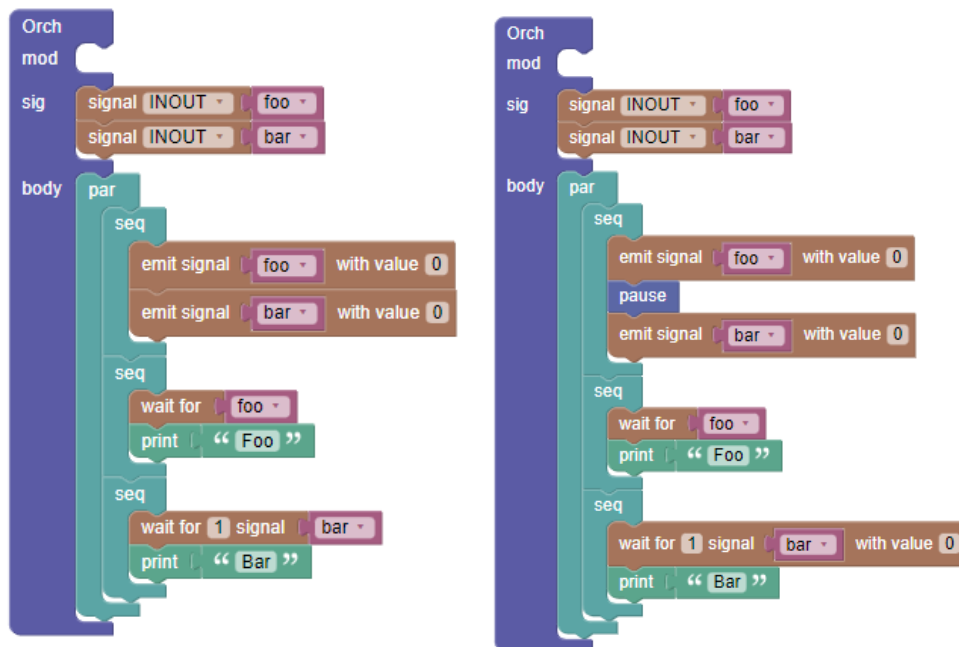


## 8.19 QUELQUES SUBTILITES DE LA PROGRAMMATION REACTIVE SYNCHRONE

### 8.19.1 Réaction immédiate

La programmation réactive synchrone est intuitive, car son fonctionnement est assez proche de la façon dont nous exprimons. « Je fais ceci, en attendant cela. Si quelque chose arrive alors j'arrêterai de faire quelque chose... ». Néanmoins sa sémantique très précise peut faire apparaître des incohérences peu visibles dans nos habitudes d'expression. Cette sémantique à l'initialisation d'un programme fait la différence entre un signal pris en compte immédiatement ou après une première réaction.

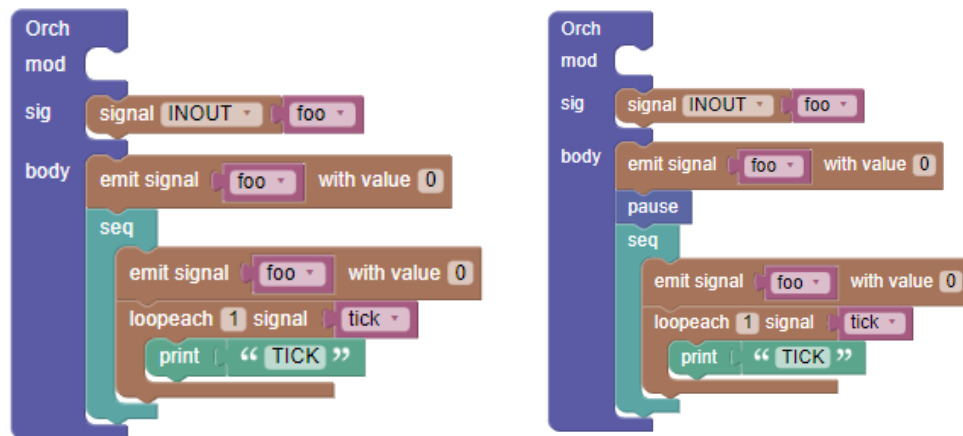
Si vous lancez le programme de gauche, vous verrez que seul « Foo » apparaît sur la console. En effet il y a une différence subtile entre « wait for » et « wait for 1 signal ». Le « wait for » réagit immédiatement, « wait for 1 signal » a besoin d'une première émission et d'une réaction pour démarrer. Il est assez rare que dans une orchestration cette subtilité pose un problème.



Dans Skini il n'y a que « wait for », « wait for pattern » qui réagisse à la première émission.

### 8.19.2 Double émission d'un même signal

Le programme de gauche va poser un problème à l'exécution. « foo » sera émis deux fois à la même réaction. Ce message va s'afficher dans la console : `TypeError: Can't set single signal 'foo' value more than once.`



Le block « pause » dans le programme de droite permet de décaler la deuxième émission d'une réaction.

## 8.20 POUR LA MUSIQUE GENERATIVE

La plupart des fonctions de base de Skini couplées avec le simulateur permettent de produire de la musique de façon automatique.

Nous avons la possibilité de réorganiser les files d'attentes sur le serveur. Ceci est paramétré par « Algo Fifo management » dans la fenêtre « Parameters ».

- 0 signifie pas de traitement.
- 1 active un algorithme de classement horizontal début (type 1), milieu (type 2), fin (type 3) et neutre (type 4).
- Active un algorithme de classement vertical

Il existe une autre façon de contrôler l'organisation des séquences de patterns (horizontal) depuis le simulateur. Cette fonction est plus riche car elle permet de réorganiser à la volée la politique de traitement des séquences.

Certaines fonctions sont dédiées au contrôle du simulateur qui permettent de gérer des comportements sur la construction des listes de patterns envoyés par le simulateur. Ceci permet de structurer des phrases musicales en fonction des types (horizontaux) assignés aux patterns.

Pour activer l'organisation des listes envoyées par le simulateur il faut insérer le bloc :

activate Type list in simulator

Pour désactiver l'organisation dans le cours d'une pièce :

deactivate Type list in simulator

Le simulateur construira des phrases musicales en prenant pour modèle la liste des types définie par le bloc :

set type list for simulator “ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ”

Ici, parmi les patterns disponibles, le simulateur les rangera dans l'ordre défini par la liste des types en choisissant de façon aléatoire un pattern par type. S'il n'y en a aucun pour un type, celui-ci est ignoré. Pour que le processus fonctionne sur tous les types déclarés il faut que la liste des patterns pour les clients (et donc le simulateur) soit de la taille de la liste des types. Par exemple selon les types définis ci-dessus :

set pattern list length to 11 for group 255

*Remarque 1* : Il est possible d'ordonner plusieurs instruments en même temps en gérant des types différents pour chacun des instruments. Voir opus4 comme exemple.

*Remarque 2* : La génération de musique avec manipulation des types doit aussi prendre en compte la façon dont Skini réagit aux activations des patterns. « React on play » activé laisse le champ ouvert tant que le pattern n'est pas joué puisqu'il peut y avoir un délai entre demander un pattern et l'entendre.

*Remarque 3* : Le rythme auquel le simulateur envoie des séquences a un impact sur la répétition de patterns dans le cas de l'utilisation de réservoirs (tank) si « React on play » est actif.

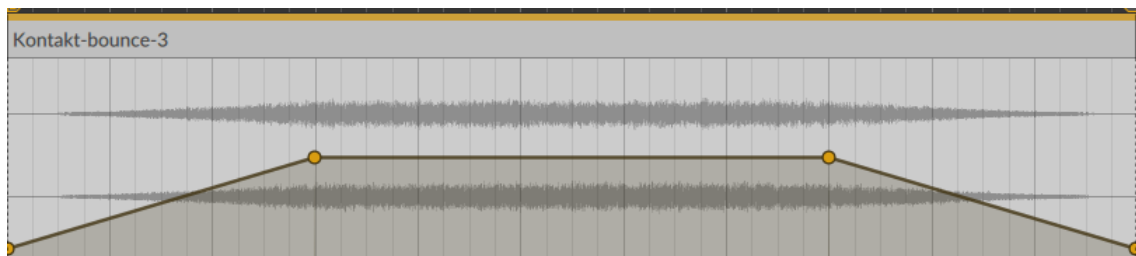
Il y a donc une alchimie à réaliser entre réservoir, rythme du simulateur et modèle de réaction. Ceci permet de créer des séquences de patterns conformes à des modèles de types et en polyphonie puisqu'il est possible de « typer » plusieurs instruments en même temps. Globalement, ces fonctions permettent de contrôler et modifier dynamiquement la probabilité d'apparition des patterns et donc d'organiser la musique en conséquence.

## 8.21 SPATIALISATION PAR TUILAGE AVEC LE DISPOSITIF PRE

Voici une façon de mettre en place des effets de spatialisation par tuilage c'est-à-dire de patterns qui se recouvrent partiellement dans le temps.

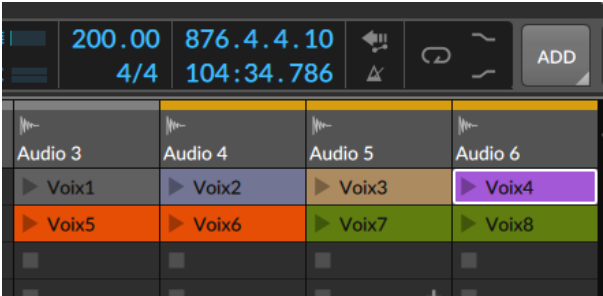
(Exemple espace1.xml, espace1.csv, espace1.js).

Les patterns ont ici tous la même structure avec un Fade IN et un Fade OUT pour permettre le recouvrement des patterns. Nous prendrons un tempo rapide pour s'autoriser des recouvrements rapides que nous ne ferons pas ici, mais qui seront ainsi possibles.



Les patterns ont une durée de 44 pulsations avec des Fade IN et OU de 12 pulsations chacun.

Voici les patterns dans bitwig Studio. La répartition des patterns doit uniquement prendre en



compte la fait qu’il ne faudra pas interrompre un pattern en cours si on utilise un processus aléatoire. C’est un point à prendre en compte dans l’affectation des instruments et des groupes.

Voici un exemple de descripteur. L’exemple est donné avec une DAW, mais il suffirait d’affecter des Raspberry Pi aux Patterns.

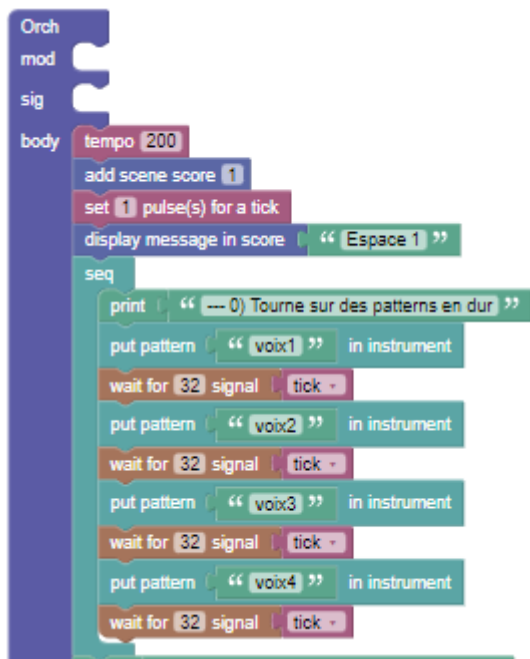
	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration
1	11	10	0	voix1	voix1	0	0	0	0	0	44
2	12	10	0	voix2	voix2	1	0	0	0	1	44
3	13	10	0	voix3	voix3	2	0	0	0	2	44
4	14	10	0	voix4	voix4	3	0	0	0	3	44
5	15	10	0	voix5	voix5	4	0	0	0	4	44
6	16	10	0	voix6	voix6	4	0	0	0	4	44
7	17	10	0	voix7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

Voici un exemple de groupes :

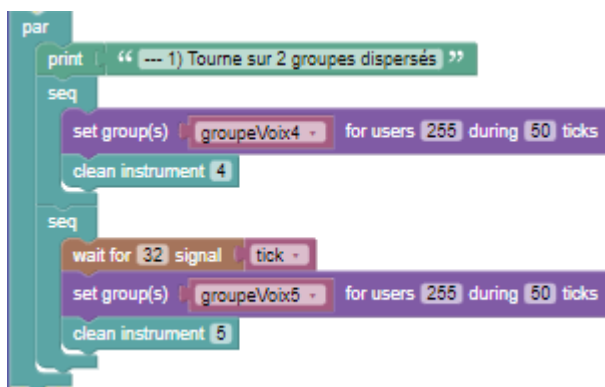
	Groupe	Index	Type	X	Y	Nb of El. or Tank nb	Color	Previous	Scene
1	groupeVoix0	0	group	200	100	20	#CF1919		1
2	groupeVoix1	1	group	400	100	20	#CF1919		1
3	groupeVoix2	2	group	600	100	20	#CF1919		1
4	groupeVoix3	3	group	800	100	20	#CF1919		1
5	groupeVoix4	4	group	200	300	20	#CF1919		1
6	groupeVoix5	5	group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

Pour un contrôle en « dur », c’est-à-dire figé, sans phénomène aléatoire. Il suffit de lancer les patterns les uns à la suite des autres en attendant le début du Fade OUT à la 32ème pulsation.





Pour faire « tourner » sur deux groupes avec une sélection aléatoire dans chaque groupe, il suffit de décaler l’activation des groupes en parallèle.



Pour faire tourner sur plusieurs groupes il suffira de les décaler en parallèle. L’exemple ci-dessous donne la même chose que le premier exemple « figé » car nous n’avons qu’un seul pattern dans chaque groupe.



**Remarque :** C'est la notion d'instrument Skini qui rend facile la réalisation de spatialisation par tuilage. Dans un instruments les patterns étant mis en file d'attente il suffit de jouer sur des décalages d'instruments pour enchaîner des patterns qui sont dans des instruments différents.

## 9 PROGRAMMATION TEXTUELLE DES ORCHESTRATIONS

Il est possible de programmer les orchestrations avec le langage HipHop.js sans utiliser l'interface de programmation graphique. Les fonctions sont similaires mais l'intégration avec JavaScript est plus riche puisque JavaScript est utilisable de la même façon que HipHop.js. Cette façon de travailler nécessite une bonne compétence en programmation JavaScript et HipHop.js.

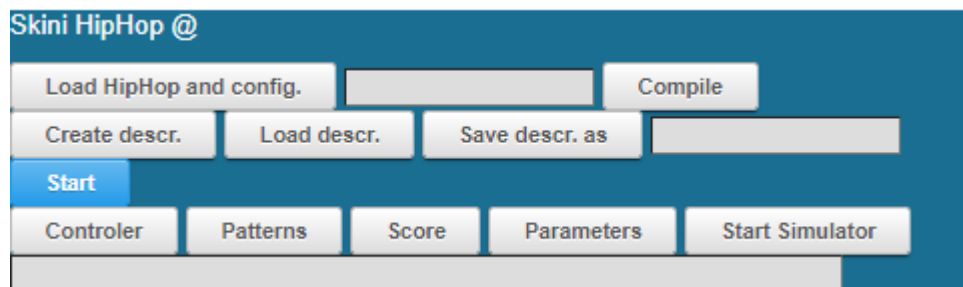
Pour la programmation HipHop.js une documentation est disponible à l'url :

<http://hop.inria.fr/home/hiphop/index.html>

Les fichiers d'orchestration HipHop.js doivent être nommés sous la forme *<fichier d'orchestration>.hh.js*, le fichier de paramètres est le même que celui de l'interface graphique *<fichier d'orchestration>.js*. Les paramétrages et patterns suivent les mêmes processus que ceux de l'interface graphique.

### 9.1 INTERFACE DE CONTROLE POUR LA PROGRAMMATION TEXTUELLE

Le chargement, la compilation et l'exécution des orchestrations s'effectuent à l'aide d'un accès par navigateur à Skini : <http://IP du serveur :8080/contrHH>.



La plupart des fonctions sont similaires à ceux de la fenêtre Blockly. Le fichier HipHop.js est saisi avec le bouton *Load HipHop and Config*. On peut à tout moment recompiler le fichier HipHop.js sans le recharger avec le bouton *Compile*.

L'orchestration est un programme HipHop.js qui est chargé et exécuté par la plateforme Skini. Il a un format particulier dans le sens où certaines fonctions sont définies et indispensable pour que Skini puisse exécuter le programme.

Quelques exemples sont disponibles :

- `helloSkini.hh.js`
- `opus4.hh.js`

Il y a deux fonctions indispensables :

```
export function setServ(ser, daw, groupeCS, oscMidi, mix) {  
  if (debug) console.log("hh_ORCHESTRATION: setServ");  
  DAW = daw;  
  serveur = ser;  
  gcs = groupeCS;
```

```
oscMidiLocal = oscMidi;
midimix = mix;
}
```

Appelé par Skini pour mettre en place tous les accès aux fonctions de contrôle.

```
export function setSignals(param){...}
```

qui comprend l'orchestration dans le module

```
const Program = hiphop module() {...}
```

elle-même générée par l'instruction :

```
const prg = new ReactiveMachine(Program, "orchestration");
```

## 9.2 LES FONCTIONS DE SKINI VIA HIPHOP.JS (A FAIRE)

# 10 AVEC DES MUSICIENS (PAS EN PLACE AVEC NODE.JS)

Avec des musiciens, Skini met en place un décompte avant le jeu du premier pattern dans une file d'attente.

Dans le fichier de configuration de la pièce, pour mettre en place les spécificités liées au jeu avec des musiciens :

```
exports.avecMusicien = true;
```

Ceci introduit un pattern vide avant le premier pattern dans une FIFO.

ATTENTION : Ce décalage se fait en fonction du *tick* (et non de la pulsation) et doit être un multiple du *tick*, sinon le player se bloque.

```
exports.decalageFIFOavecMusicien = 8;
```

Pour spécifier la position des fichiers partitions des patterns en format jpg, dans le fichier de configuration de la pièce nous aurons par exemple :

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

Ces chemins sont relatifs au répertoire « ./images ».

## 11 INTERFACE OSC

---

### 11.1 PRINCIPE

Si une plateforme de développement de jeu ou depuis un contrôleur peut envoyer des commandes OSC, il est possible de générer des signaux dans l'orchestration à partir d'événements externes et inversement de générer des messages OSC à partir de l'orchestration.

La configuration des adresses IP des émetteurs récepteurs se fait dans *ipConfig.json*. Les adresses et port IP de l'équipement distant avec lequel se fait le dialogue OSC, sont définis par *remoteIPAddressGame* et *portOSCToGame*, les messages OSC sont reçus sur le port *portOSCFromGame*. (La terminologie *game* vient de la première utilisation de ce mécanisme pour communiquer avec Unreal Engine, mais il est utilisable pour tout équipement qui parle OSC).

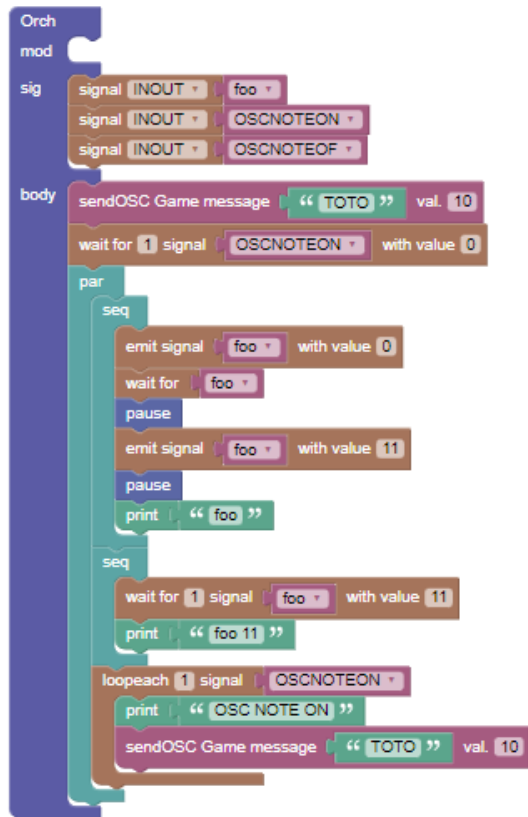
**NB :** La limite du mécanisme actuel est que Skini ne donne des ordres qu'à une seule adresse IP/portIP. En revanche, Skini peut recevoir des messages OSC de plusieurs équipements.

### 11.2 DEPUIS L'ORCHESTRATION

Il faut créer des signaux correspondant aux messages OSC reçus. Les signaux correspondent aux messages sans les slashes « / ». Le message /FOO/BAR donnera le signal FOO\_BAR dans Skini. Le message émis ne comporte pas de slash en entête, il est ajouté par Skini.

**Remarque sur les signaux OSC :** Les signaux créés pour OSC ne doivent pas avoir les mêmes noms que des signaux par défaut. Il est prudent de définir un format commun à ces signaux en leur adjoignant un suffixe ou un préfixe *OSC* ou *Game* ou autre.

### 11.3 MISE EN ŒUVRE AVEC L'ORCHESTRATION



Exemple de programme pour la réception et l'émission de commande OSC :

- depuis un émetteur OSC vers *serverIPAddress* sur le port *portOSCFromGame*.
- vers un récepteur OSC en *remoteIPAddressGame* sur le port *portOSCToGame*.

## 12 OSC AVEC BITWIG STUDIO

Il est possible de faire communiquer Skini en OSC avec Bitwig Studio, donc sans utiliser Processing comme passerelle OSC/Midi. Le contrôleur Bitwig, Skini\_0.control.js, se comporte comme la passerelle Processing pour Ableton.

Sur Bitwig le contrôleur Skini\_0 a les mêmes paramètres que Processing, par exemple :

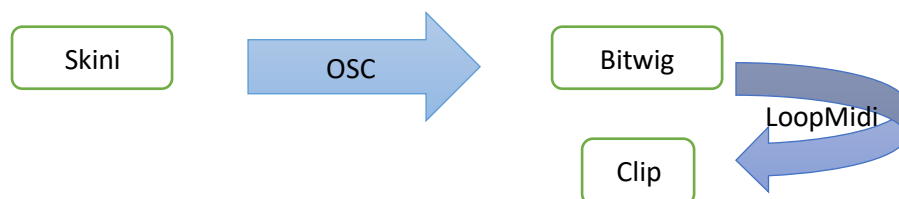
Remote OSC Ip address: 192.168.1.6

Remote OSC port out : 13000

Listening OSC : 12000

Les informations Midi sont envoyées vers Bitwig comme elles le sont vers Processing. Il n'y a donc pas de modification dans Skini au niveau de la génération des commandes Midi pour les patterns ou autres commandes. Pour garder la compatibilité avec la passerelle Processing, Bitwig se renvoie vers lui-même les commandes OSC pour le Midi sur un canal Midi dans LoopMidi. (Je ne sais pas envoyer une commande Midi directement vers les déclenchements de clips). Il est donc nécessaire de déclarer un contrôleur du type clavier générique (Generic Keyboard) qui reçoit les commandes midi issues du contrôleur Skini\_0.

Il y a donc un port Midi OUT à donner dans le contrôleur Skini\_0 vers LoopMidi, qui sera la port Midi en entrée dans le clavier générique.



Le contrôleur Bitwig peut envoyer des commandes Midi issues de l'un de ses contrôleurs vers Skini. C'est le port donné en Midi OUT du contrôleur Skini\_0. L'interprétation des commandes OSC portant du Midi vers Skini se trouve dans serveur/midimix.js.

La correspondance entre les commandes OSC de Bitwig vers Skini est définie dans les fichiers Skini\_0.control.js et midimix.js.

De plus, Bitwig émet un tick en OSC "/BitwigTick". Le calcul de ce tick dans le contrôleur n'est pas vraiment canonique. Il se base sur la barre de transport de Bitwig.

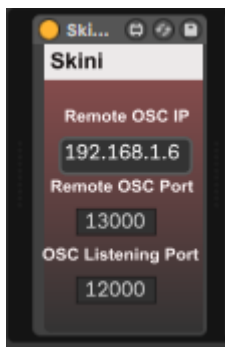
Exemple : pièce technoBitwig sur ABL3.

**Conclusion** : Les seules modifications apportées à Skini pour une première mise en œuvre de Bitwig se trouvent dans midimixi.js et dans websocketServerSkini.js.

**Remarque** : Dans Bitwig studio il n'y a pas de notion de bus (port midi). Bitwig reçoit les messages en OSC et les reroute pour le contrôle via loopMidi. Les paramètres de bus présents dans les commandes OSC envoyées vers bitwig ne sont donc pas pris en compte.

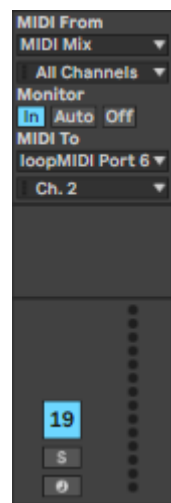
## 13 OSC AVEC MAX4LIVE

Il est possible d'utiliser Skini et Ableton sans passer par la passerelle Processing. Pour ceci on utilise Max for Live (M4L). La mise en œuvre consiste à créer une piste Midi dans laquelle il faut charger le patch Max « SkiniOSC2.amxd ». La piste peut recevoir en entrée un équipement MIDI dont les commandes seront émises vers Skini en OSC.

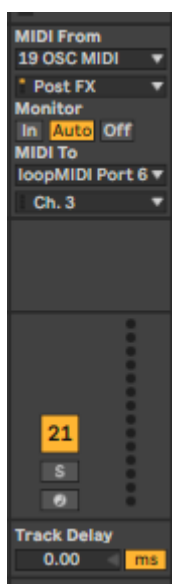


### 13.1 PISTE DE CONVERSION

La piste Midi d'Ableton doit être routée en sortie Midi vers un canal de contrôle qui va recevoir les commandes OSC de Skini converties en Midi. En effet, il n'est pas possible de parler directement Midi à un contrôle Ableton Live depuis M4L, il faut passer par un câble virtuel (LoopMIDI par exemple). Les paramètres du patch concernent OSC. Le paramétrage Midi se fait dans la piste Ableton Live.



### 13.2 PISTE DE ROUTAGE MIDI



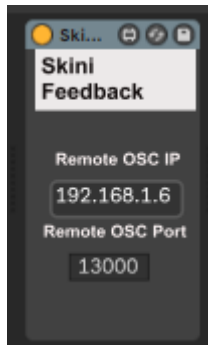
Un bug d'Ableton Live complique les opérations dès que l'on a plus de 127 notes associées aux patterns dans Skini. **M4L dans une piste Ableton ne sait pas adresser autre chose que le canal Midi numéro 1.** Pour contourner ce problème il faut créer des pistes supplémentaires, autant que de multiples de 127 parmi les notes associées aux patterns dans le fichier de configuration des patterns (.csv).

Il faut charger le patch M4L « SendToMIDIChannel.amxd » dans ces pistes Midi supplémentaires et donner le bon canal sur la piste en « MIDI to » Ci-dessous un exemple de configuration d'un canal Midi. Le canal 2 de Skini correspond au canal 3 d'Ableton. Il y a un décalage d'une unité entre Skini et Ableton.





### 13.3 PISTE DE FEEDBACK



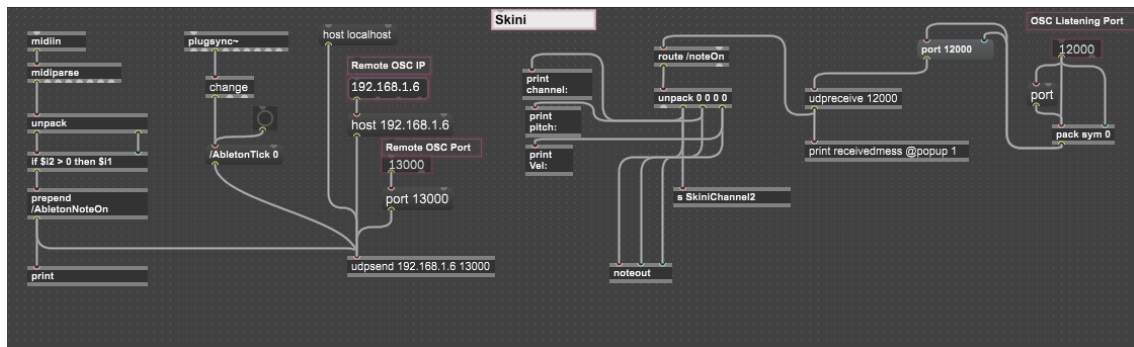
Pour traiter les informations Midi envoyées par Ableton lorsqu'un pattern est joué (cette fonction n'est utile que si on utilise le signaux *patternSignal* dans l'orchestration), il faut encore ajouter une piste Midi avec un patch M4L *SkiniAbletonFeedback.amxd*. Cette piste a pour port d'entrée (MIDI From) un port Midi qui reçoit les « Télécommandes ». Par exemple :

Les « télécommandes » sont envoyées à Skini en OSC. Ici encore nous sommes limités à 127 notes Skini. Pour pallier ce problème il est plus simple d'utiliser Processing, vu le niveau de complexité de paramétrage atteint dans Ableton.

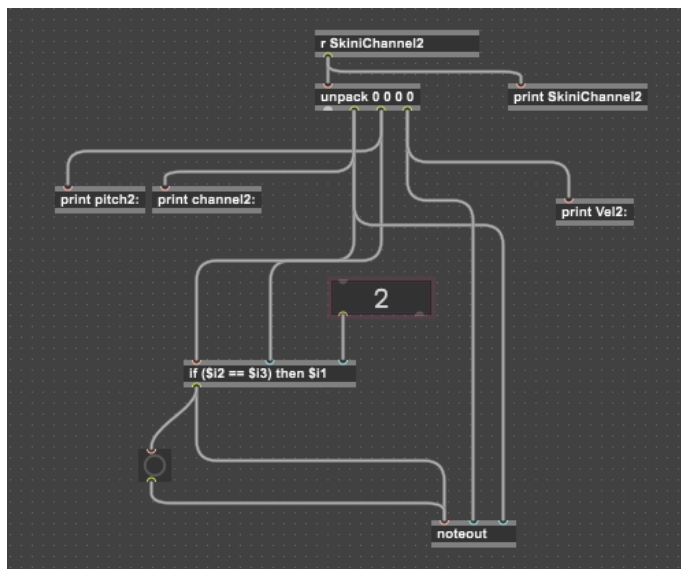


## 13.4 LES PATCHS M4L

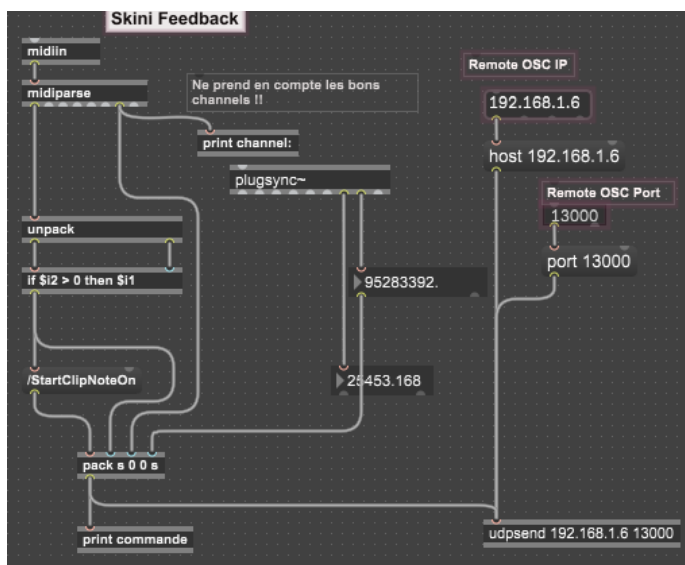
Voici le patch *SkiniOSC2.amxd* :



Le patch *SendToMIDIChannel.amxd* :



Le patch *SkiniAbletonFeedback.amxd* :



### 13.5 CONCLUSION OSC SKINI AVEC ABLETON

L'utilisation d'Ableton Live avec Skini est possible sans Processing, elle devient complexe dès que l'on a plus de 127 notes de Patterns dans Skini à cause d'un problème de conception dans M4L. Il s'agit d'une démonstration supplémentaire de l'absence de maturité de M4L.

Comme exemple Ableton voire la pièce *technoDemoOSC.als*.

#### Remarques :

1) L'assignation des commandes MIDI à des clips dans Ableton, traite les canaux en commençant depuis 0. Skini lui convertit les canaux Midi depuis 1. On voit donc que les commandes de Skini commencent en fait sur le canal 2 du port Midi de contrôle d'Ableton. Si l'on traite moins de 127 notes Skini, on peut utiliser uniquement une piste de *routage Midi* en affectant en sortie le *canal* 2 sur le port Midi de contrôle.

2) Le *noteout* de *SkiniOSC2.amxd*, ne tient pas compte du canal Midi qu'on lui donne mais uniquement du canal Midi en sortie de la piste. La manipulation pour s'en sortir consiste à utiliser *send/recieve* de Max/Msp. Le patch *SkiniOSC2.amxd* envoie la commande Midi au patch *SendToMIDIchannel.amxd* qui compare le canal reçu avec son paramètre. Ce patch perd des notes Midi sans raison, le *send/receive* ne fonctionne pas de temps en temps ou *noteout* ne fait rien et il n'y a pas de mécanisme de *try and catch* pour savoir pourquoi.

## 14 CONTROLE DE PATTERNS SUR RASPBERRY

---

Pour utiliser Skini en contrôle de Raspberry du projet « Pré » (de Jean-Luc Hervé) avec OSC, il y a 3 paramètres à modifier dans la configuration de la pièce :

- « Use Raspberries », coché;
- « Play Buffer Message » avec 'test';
- « Raspberry OSC Port » à 4000;

Le paramètre *Use Raspberries* permet de désactiver le jeu sur Raspberry de façon global et de jouer les patterns sur la DAW. *playBufferMessage* est le message OSC (sans /) que comprend le Raspberry pour jouer un fichier son. *raspOSCPort* est le port UDP utilisé par OSC pour les Raspberries.

On étend le descripteur de pattern avec trois informations :

- L'adresse IP du Raspberry qui doit jouer le pattern
- Le numéro du son dans le Raspberry correspondant au pattern (champ « Buffer num »).
- Le niveau sonore du pattern (0 à 128)

Si le champ « buffer num » n'est pas renseigné, le pattern est considéré comme devant être joué par la DAW.

Il est possible ainsi de jouer dans une même pièce des patterns avec la DAW et les Raspberries. On peut aussi ainsi tester un pattern sur la DAW avant de la faire jouer par un Raspberry.

**NB pour développeur :** La *file d'attente* stocke l'adresse IP du Raspberry à l'index 10, le numéro du buffer à l'index 11 et le niveau à l'index 12. Dans la *configuration* des patterns l'adresse IP du Raspberry est en index 11, le buffer en index 12 et le niveau en index 13.

## 15 ANNEXES

---

### 15.1 SKINI AVEC ABLETON LIVE

Nous avons souvent utilisé Ableton Live comme DAW pour nos développements. Toute DAW pouvant associer une commande MIDI à un clip sans contrainte de synchronisation fait globalement l'affaire. Ableton Live offre la possibilité d'associer des commandes MIDI à un grand nombre de paramètres, tempo, MAX/MSP, commandes d'enregistrement etc. Son usage avec Skini est donc très riche et simple à mettre en œuvre avec le configurateur Skini.

Pour nos développement les patterns ont été conçus dans Ableton, la plupart du temps en format MIDI. Ableton permet la conversion des clip MIDI en sons si besoin, mais on perd alors les traitements MIDI comme les transpositions, les conversions de mode etc.

Pour nos pièces nous n'utilisons pas de quantification globale dans Ableton. La synchronisation des clips est assurée par Skini.

Comme présenté dans cette documentation, la configuration des ports MIDI ne présente pas de difficultés, on fera attention au port IN utilisé pour les commandes Skini vers Ableton qui doit autoriser les Télécommandes (« Téléc. ») et le port OUT utilisé par Ableton pour émettre des messages MIDI vers Skini (passerelle Processing) qui doit aussi autoriser les Télécommandes. La configuration des ports se fait entre les lignes 214 à 256 du « sequenceurSkini » de Processing et les Préférences/MIDI d'Ableton.

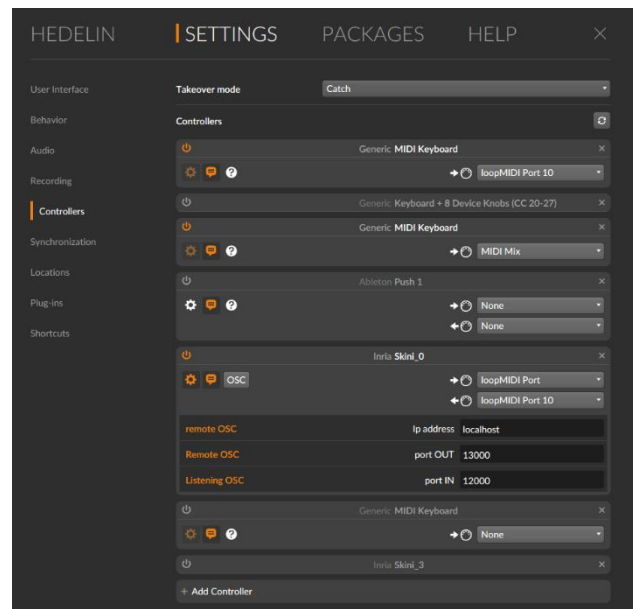
Il faut activer la synchronisation MIDI d'Ableton sur le port OUT correspondant au port IN de la passerelle Processing pour que le tempo d'Ableton contrôle Skini.

Dans la version Windows avec LoopMIDI, le port 12 est utilisé par Ableton pour informer Skini du clip lancé. (Le port13 est utilisé pour passer les contrôles MIDI issus de la vidéo dans Reaper). Ces ports sont « câblés en dur » dans l'onglet *OSCMidi* de la passerelle Processing.

## 15.2 SKINI AVEC BITWIG STUDIO

Avec Bitwig studio il est aussi possible de communiquer en OSC et donc de se passer de la passerelle Processing. Un contrôleur Bitwig a été développé dans ce but (`./bitwig/Skini_0/Skini_0.control.js`). Il reçoit les commandes Skini sous forme OSC et pour ne pas changer le mode de déclaration des patterns, Bitwig Studio reroute en sortie ces commandes OSC vers une interface MIDI virtuelle (sur windows LoopMidi par exemple) qui renvoie en MIDI les commandes reçues.

Le figure de droite donne un exemple de configuration Bitwig Studio.



## 15.3 EXEMPLES D'ORCHESTRATIONS

**trouveLaPercuNode.xml** : est un exemple de jeu où l'audience doit trouver la correspondance entre des patterns et une ambiance sonore. La session Bitwig studio correspondant est worldMusicGame. L'exemple dans Bitwig studio utilise des VST de Native Instruments

**opus5.xml** : est exemple qui fait appel essentiellement à des réservoirs avec des commandes MIDI de transposition. L'exemple dans Ableton utilise des VST de Native Instruments. Le set Live est opus5.

## 15.4 ORGANISATION DU SYSTEME DE FICHIER

A partir du répertoire où est installé Skini nous avons les répertoires :

*client* : Les différents clients de skini se trouvent dans des sous-répertoires.

*docs* : suivi de la thèse et du projet Skini

*images* : dans des sous-répertoires par pièces nous avons ici les partitions des patterns sous forme de fichier jpg.

*pieces* : Dans ce répertoire nous avons les fichiers de configuration des pièces et les fichiers csv de définition des patterns

*Processing* : contient les programmes Processing, *sequenceurSkini* est celui utilisé couramment.

*sequencesSkini* : contient les patterns sauvegardés par le séquenceur distribué.

*serveur* : contient les fichiers Hop.js et HipHop.js de Skini et la configuration générale.

*sounds* : les fichiers son mp3 des patterns sont organisés en sous répertoires.

*blockly\_hop* : qui contient les programmes blockly et les orchestrations au format xml

*bitwig* : contient les extensions pour Bitwig Studio

## 15.5 ENREGISTREMENT DE PARTITION DANS FINALE

On ne peut pas extraire le MIDI directement d'Ableton si l'on craint de perdre les commandes MIDI de transposition envoyées directement par Processing. S'il n'y a pas de modification des patterns en Live on peut directement compléter les pistes MIDI dans Live et les exporter vers Finale. Si on veut utiliser des modifications de patterns il faut appliquer les procédures ci-dessous.

Le prérequis dans ce processus est de ne pas avoir de modification de tempo dans la session Skini. Ces modifications sont à apporter dans la partition finale. Les changements de tempo n'ont pas d'importance car il ne s'agit pas d'un enregistrement sonore mais d'une transcription écrite.

Penser à prévoir une option dans les automates pour désactiver les changements de tempos.

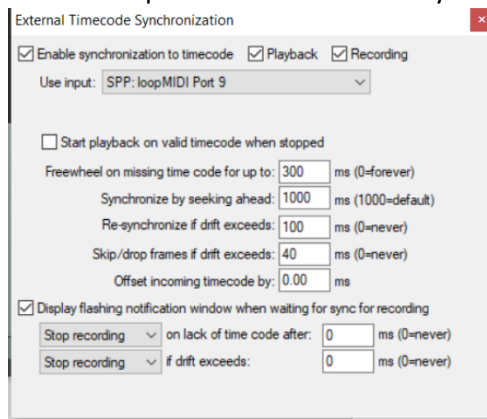
Attention aussi à la durée des patterns. Ici il faut nécessairement activer la quantification globale d'Ableton Live pour éviter les micro-décalages MIDI impossibles à retraiter. Fixer la quantification globale revient à fixer la longueur des patterns dans Skini.

Moyennant ces précautions la transcription de Skini vers Finale prend quelques minutes.

### 15.5.1 A partir d'un enregistrement d'Ableton

- 1) Sauver le fichier Ableton Live d'origine
- 2) Le copier dans un autre fichier pour préparer la conversion MIDI
- 3) Remplacer les VST de chaque piste par un canal MIDI OUT.  
Il suffit de supprimer le VST pour autoriser une sortie MIDI sur une Piste.  
Attention au port de sortie utilisé pour la synchro MIDI.
- 4) Enlever les variations de tempo (s'il y en a)
- 5) Configurer les canaux MIDI dans Reaper (le port 9 est en général utilisé pour la synchro MIDI).

## 6) Mettre Reaper en Slave avec MIDI sync



- 7) Mettre en Ableton en émission MIDI Sync (vérifier les ports de sync Ableton et Reaper)
- 8) Enregistrer dans Reaper en fixant le même tempo qu'Ableton Live. Attention d'avoir Ableton Live avec une quantification globale.
- 9) Dans Reaper mettre les pistes MIDI en forme, supprimer le vide du départ de l'enregistrement, quantifier le MIDI...
- 10) Exporter chaque piste MIDI de Reaper indépendamment
- 11) Charger les pistes MIDI enregistrées par Reaper dans Finale
- 12) Mettre en forme dans Finale.

### 15.5.2 Directement depuis Skini

Les paramétrages et les contraintes sur les variations de tempo sont les mêmes qu'au-dessus. On enregistre directement le déroulement d'une session Skini dans Reaper sans l'enregistrer dans Live.

Il faut faire attention d'avoir une quantification globale dans Ableton Live pour éviter les petits décalages ou dérives MIDI issues de Skini. Il doit être possible de régler le retard sur la synchro MIDI dans Live pour être parfaitement en phase entre Live et Reaper.

## 15.6 PASSER DE FINALE A MICROSOFT WORD OU POWERPOINT

- Exporter la partition au format PDF dans Finale
- Charger le PDF dans INKSCAPE via import en prenant les options qui intègrent les polices (pas l'option par défaut)
- Dans INKSCAPE enregistrer en format SVG simple.
- Glisser le SVG vers Microsoft

## 15.7 RECEVOIR LES INFOS MIDI DE LANCEMENT DE CLIP D'ABLETON

- Il faut des adresses IP dans IPconfig, pas de localhost.
- Mettre un canal MIDI out inutilisé en « Telec » (remote). Ceci revient à considérer qu'il y a une surface de contrôle sur le port (dans opus1 et 2 c'est le loopMIDI Port out 12).
- La correspondance doit être fait dans Processing (sequenceurSkini.pde) dans le tableau « myBusIn » et dans les tests de provenance des commandes MIDI (OSCMidi).

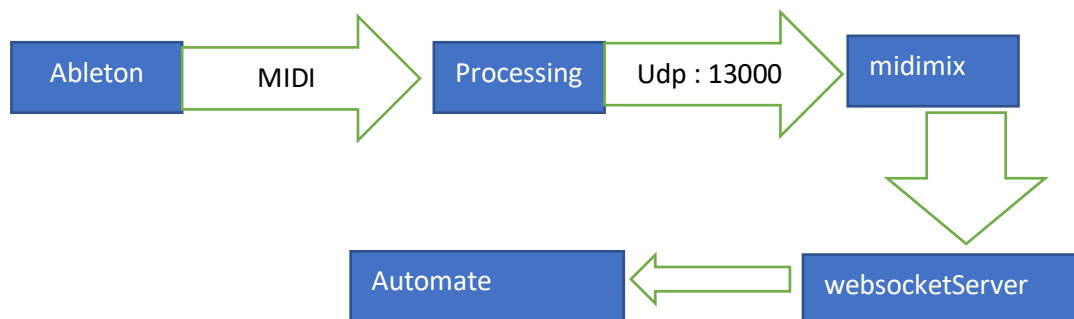
Processing traduit les commandes qui viennent du port MIDI en commande OSC « StartClip ».

- Ableton envoie sur ce canal les commandes MIDI de lancement des clips.

Rem : En mode « piste » sur un canal MIDI out Ableton émet les notes MIDI qu'ils traitent mais pas les CC. Si l'on souhaite récupérer des CC il faudrait qu'ils partent des clips

Les commandes MIDI envoyées par Ableton suivent un protocole bizarre qui est traité par midimix.js qui est le programme de traitement des données OSC provenant de Processing (le nom n'est pas terrible, il remonte au Golem).

Midimix est créé dans golem.js. Il passe les infos à websocketServer, c'est websocketServer qui informe l'automate. On passe donc par websocketServer uniquement pour accéder à l'automate.



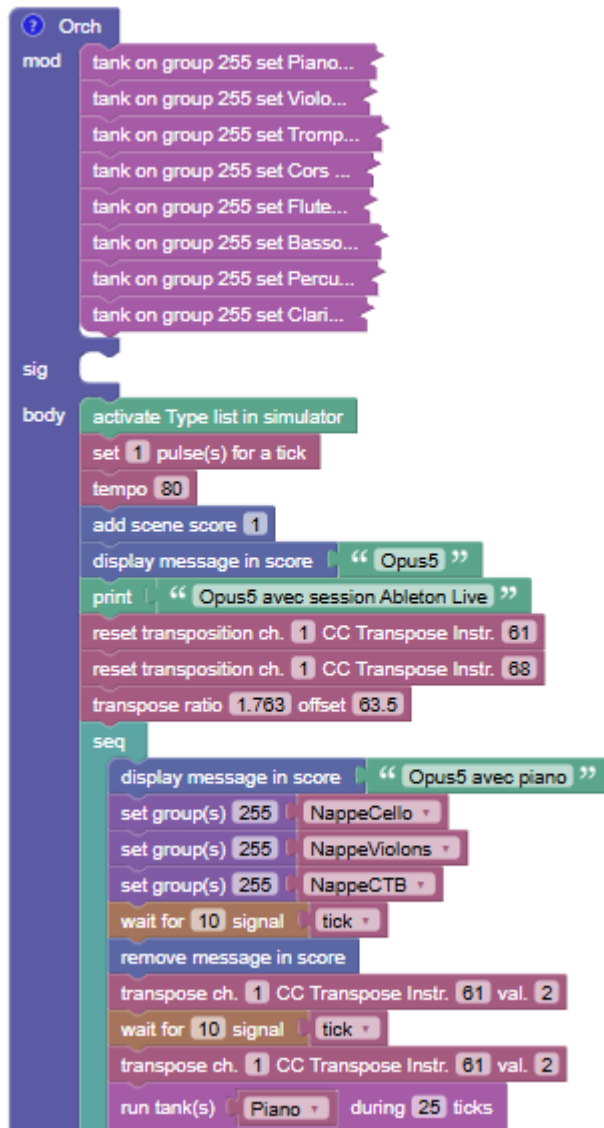


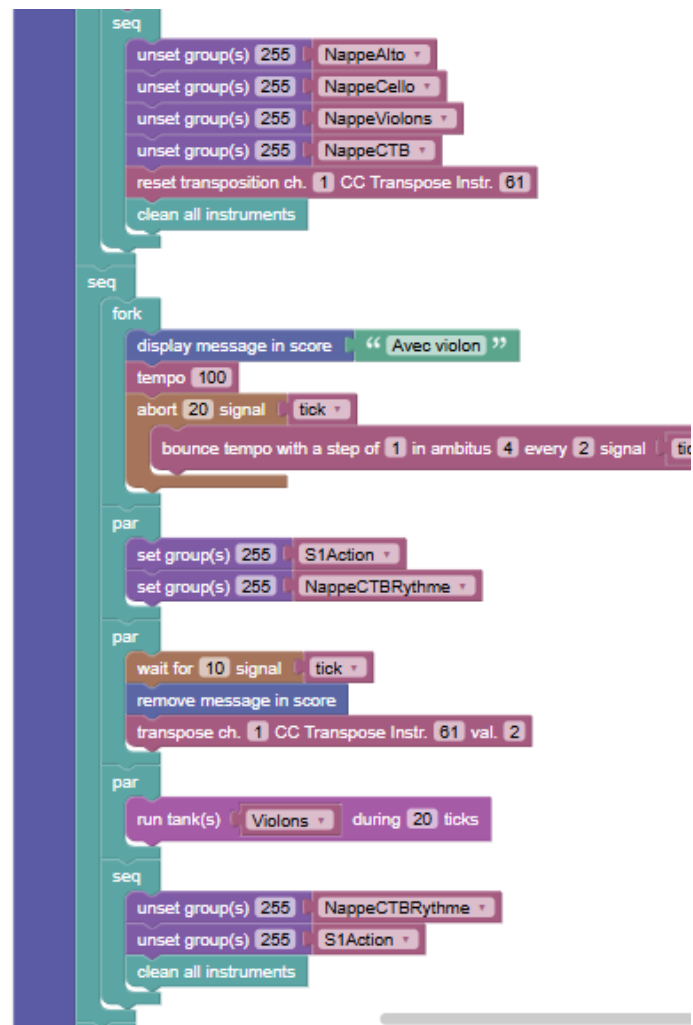
## 15.8 ENREGISTREMENT DE PATTERNS EN LIVE DANS LE SEQUENCEUR DISTRIBUE

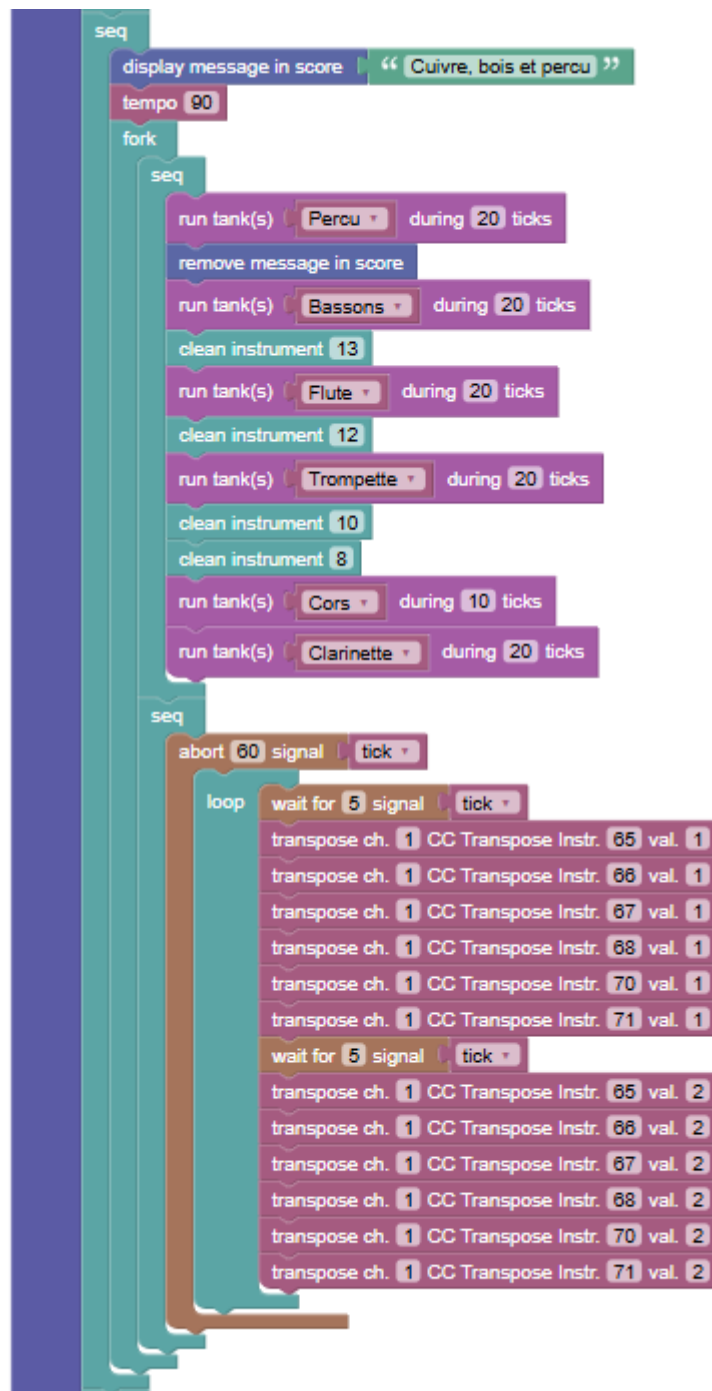
(à faire, donner la structure des programmes.).

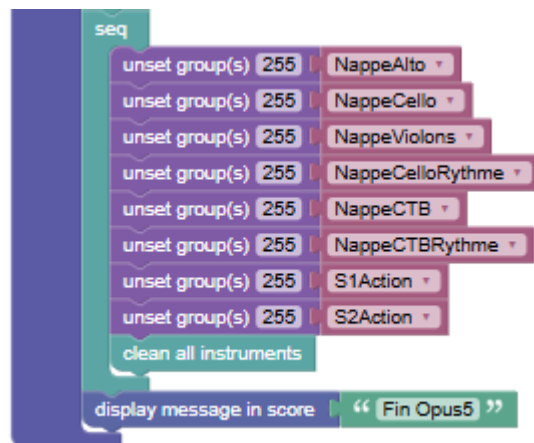
## 16 UN EXEMPLE DE PIECE : OPUS5

Cette pièce utilise la plupart des principes de base de Skini.









## 17 UNE PIECE AVEC CONTROLE DES TRANSPOSITION DIRECTEMENT AVEC CC MIDI

La transposition dans cette pièce Ableton Live (hope3v10) est assurée par un patch MIDI. En effet, les changements d'articulations pour les instruments se font avec des notes MIDI. Il faut donc filtrer ces commandes pour ne pas les transposer. Une façon plus maline est d'utiliser des CC pour les articulations et de transposer avec l'outil Ableton, mais les instruments de SessionHorns ne permettent pas le contrôle des articulations par CC.

