
Skini sur Node.JS

Mode d'emploi et implémentation

B. Petit, 05/2022

Table des matières

1	Introduction	5
1.1	Processus de composition.....	5
1.2	Patterns	5
1.3	Orchestration	5
1.4	Des pièces avec une DAW	6
1.5	Des pièces avec des musiciens	6
2	La configuration.....	7
2.1	Installation.....	7
2.2	Installation de Processing pour la passerelle osc/MIDI	7
2.3	Configuration réseau.....	7
2.3.1	Configuration IP.....	7
2.4	Configuration MIDI pour Node.js	8
2.5	Configuration des pièces.....	9
2.5.1	Accès aux pièces et descripteurs des patterns	9
2.5.2	Façon de recevoir les commandes MIDI	9
2.5.3	Mode de réaction	9
2.5.4	Les fichiers sons pour les clients et les clients	10
2.5.5	Synchronisation.....	10
2.5.6	Groupes de patterns	11
2.5.7	Musiciens.....	11
2.6	Configuration des patterns	12
2.7	Configuration de l’affichage de l’orchestration sur grand écran	13
3	Lancer Skini.....	14
4	Utilisation du contrôleur	16
5	Utilisation du configurateur MIDI	16
6	Utilisation du client SKINI	17
7	Utilisation du simulateur.....	18
8	Programmation des orchestrations	20
8.1	L’interface graphique de Skini.....	20
8.2	Les modules.....	21
8.3	La programmation du temps dans les automates	21
8.4	Tempo	23
8.5	Réservoirs et groupes de patterns	23
8.5.1	Groupes de patterns	23

8.5.2	Actions liées aux groupes	24
8.5.3	Création des réservoirs	24
8.5.4	Actions sur les réservoirs.....	25
8.6	Les files d'attentes des instruments.....	25
8.6.1	Vidage des files d'attentes	25
8.6.2	Mise en pause et test des files d'attente	26
8.6.3	Mettre un pattern spécifique en files d'attente	26
8.7	Patterns	27
8.8	Affichage de l'orchestration	27
8.9	Jeu Skini	28
8.10	Commande et Control Change MIDI	28
8.11	Spécifique Ableton Live	29
8.12	Commandes OSC	29
8.13	Les blocs avancés	29
8.14	Contrôle Interface Z	30
8.15	Programmation des transitions « stingers »	31
8.15.1	Cas de la réaction à l'exécution.....	31
8.15.2	Cas du retour de jeu de pattern depuis la DAW	32
8.16	Priorité dans les files d'attente	33
8.16.1	Exemple de programmation d'un duel de solos (revoir pour Node)	34
8.16.2	Exemple de transposition en boucle	36
9	Avec des musiciens (Revoir avec Node.js)	37
10	Interface OSC avec la plateforme de jeu Unreal Engine 4 (à revoir avec node)	38
10.1	Principe.....	38
10.2	Depuis l'orchestration	38
10.3	Mise en œuvre avec l'orchestration	39
10.4	Depuis la plateforme hors orchestration	39
11	OSC avec Bitwig Studio.....	40
12	OSC avec Max4Live.....	41
12.1	Piste de conversion	41
12.2	Piste de routage Midi	41
12.3	Piste de feedback	42
12.4	Les patches M4L.....	43
12.5	Conclusion OSC Skini avec Ableton	44

13	Contrôle de patterns sur Raspberry	45
14	Annexes	45
14.1	Skini avec Ableton Live	45
14.2	Skini avec Bitwig Studio.....	46
14.3	Exemples d'orchestrations.....	46
14.4	Organisation du système de fichier.....	46
14.5	Enregistrement de partition dans Finale.....	47
14.5.1	A partir d'un enregistrement d'Ableton.....	47
14.5.2	Directement depuis Skini	48
14.6	Passer de Finale à Microsoft Word ou PowerPoint	48
14.7	Recevoir les infos MIDI de lancement de clip d'Ableton	48
14.8	Enregistrement de patterns en Live dans le séquenceur distribué	50

1 INTRODUCTION

Skini est une plateforme de composition de musique collaborative et générative. Ce document aborde les points techniques relatifs à la mise en œuvre de Skini sur Node.js et la création d'orchestration. Il correspond au fonctionnement de Skini dans sa version de juin 2021. 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 au 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 pas des processus aléatoires. La solution comporte un serveur Web qui intègre des modules d'orchestration écrits au moyen d'un outil de programmation qui est une couche d'abstraction au-dessus du langage HipHop.js. La méthode de composition repose sur deux concepts de base : les patterns 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 groupes¹. 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 pas 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 d'automatismes complexes à des DAW du commerce. L'orchestration peut aussi commander la mise en file d'attente de patterns comme le ferait l'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 textuelle ou graphique pour la version Hop.js et uniquement graphique pour la version Node.js. Une connaissance élémentaire

¹ Si le compositeur souhaite mettre à la disposition de l'audience un pattern particulier, il peut créer un groupe avec un seul élément par exemple.

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 DES PIECES AVEC UNE DAW

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 dans un fichier de configuration des patterns. 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. 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. Pas 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 0 sur les configurations, puis passer à l'orchestration.

1.5 DES PIECES AVEC DES MUSICIENS

Skini peut être utilisé pour dialoguer avec des musiciens 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

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) 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).

2.3 CONFIGURATION RESEAU

2.3.1 Configuration IP

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. }
```

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 une 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.

2.4 CONFIGURATION MIDI POUR NODE.JS

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.

```
[
  {
    "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...).

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.

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*.

2.5 CONFIGURATION DES PIECES

La commande pour lancer Skini est : **node skini**

La pièce est configurée dans un fichier JavaScript, ce fichier est chargé en même temps que l'orchestration 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 principaux paramètres du fichier descripteur de la pièce.

2.5.1 Accès aux pièces et descripteurs des patterns

Définition des chemins d'accès des pièces et des configurations des patterns par exemple :

```
exports.sessionPath = './pieces' ;
exports.piecesPath = './pieces' ;
```

Il n'est pas possible d'accéder au chemin complet depuis un navigateur, c'est pour cela que ces paramètres sont nécessaires.

2.5.2 Façon de recevoir les commandes MIDI

Pour le paramètre `directMidiON` de la ligne :

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

true signifie que la communication entre Skini et la DAW se fait via MIDI. *false* signifie que 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.

2.5.3 Mode de réaction

Le paramètre `reactOnPlay` définit la façon de faire réagir l'automate d'orchestration. Par défaut c'est à la sélection. Avec `reactOnPlay=true` c'est au moment où se joue le pattern. Ceci a un impact important sur la façon de penser l'automate. Les stingers ne sont possibles qu'avec `reactOnPlay=true`. Ex :

```
exports.reactOnPlay = false;
```

2.5.4 Les fichiers sons pour les clients et les clients

Le paramètre `soundFilesPath1` de la ligne

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

définit le chemin des fichiers sons, associés aux patterns, qui sont téléchargés par les clients depuis le répertoire `.\sounds`.

Le paramètre :

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

Fixe le nombre de groupes de personnes dans l'audience que l'orchestration peut gérer. Avec le paramètre suivant `simulatorInAseperateGroup` positionné à `true`.

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

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 `simulatorInAseperateGroup` est absent, c'est qu'il n'y a pas de groupe dédié au simulateur.

Le paramètre `algoGestionFifo` permet d'activer des algorithmes de traitement sur les files d'attente (cf. chapitre **Erreur ! Source du renvoi introuvable.** p. **Erreur ! Signet non défini.**).

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

2.5.5 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. Pas OSC ceci est possible en direct 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 *synchroLink* et *synchoOnMidiClock*.

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

Si toutes ces synchro 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.

2.5.6 Groupes de patterns

Le nommage des groupes de patterns se fait via un tableau. Les noms des groupes de patterns sont utilisés pour la création des signaux HipHop de l'orchestration. Voici un exemple :

```
1. exports.groupeDesSons= [
2.   // Pour group: nom du groupe (0), index du groupe (1), type (2), x(3), y(4),
   nbe d'éléments(5), color(6), prédécesseurs(7), n° de scène graphique
3.   ["groupe1", 0, "group", 170, 100, 20, rouge, [], 1 ], //0 index d'objet
   graphique
4.   ["groupe2", 1, "group", 20, 240, 20, bleu, [], 1 ], //1
5.   ["groupe3", 2, "group", 170, 580, 20, vert, [], 1 ], //2
6.   ["groupe4", 3, "group", 350, 100, 20, gris, [], 1 ], //3
7.   ["groupe5", 4, "group", 20, 380, 20, violet, [], 1 ], //4
8.   ["groupe6", 5, "group", 350,580, 20, terre, [], 1 ], //5
9.   ["groupe7", 6, "group", 540,100, 20, rose, [], 1 ], //6
10.  ["derwish", 7, "group", 740,480, 20, rose, [], 1 ],
11.  ["gaszi", 8, "group", 540,580, 20, rose, [], 1 ],
12.  ["djembe", 9, "group", 740,200, 20, rose, [], 1 ],
13.  ["piano", 10,"group", 740,340, 20, rose, [], 1 ]
14. ];
```

2.5.7 Musiciens

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 « Configuration ». 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	Free	Group	Duration	IP address	Buffer num	Level
1	11	510	0	FM8-1	Piano1	1	0	4	0	0	4			
2	12	510	0	FM8-2	Piano2	1	0	4	0	0	4			
3	14	510	0	Biopoly1	Piano9	2	0	4	0	2	4	192.168.1.24		60
4	15	510	0	Biopoly2	Piano10	2	0	4	0	2	4	192.168.1.24		60
5	16	510	0	Capa1	Capa1	3	0	4	0	3	4			
6	17	511	0	Capa2	Capa2	3	0	4	0	3	4			
7	18	511	0	MassiveX1	MassiveX1	4	0	4	0	4	4	192.168.1.28		70
8	19	511	0	MassiveX2	MassiveX2	4	0	4	0	4	4	192.168.1.28		70
9	30	511	0	Gam1	Gam1	5	0	4	0	5	4			
10	31	511	0	Gam2	Gam2	5	0	4	0	5	4			

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 fonctionne car Ableton Live (et sans doute les autres DAW aussi) permettent 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 à Ableton Live, ce n'est pas utile en général.

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

La colonne **nom** 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 colonnes **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. Il en est de même pour les deux dernières colonnes, « attente » et « pseudo ».

La colonne **type** permet de qualifier un pattern pour pouvoir réorganiser les files d'attente, la description de cette fonctionnalité est dans le chapitre 8.16, « Priorité dans les files d'attente ».

La colonne **groupe** est en correspondance avec les index des groupes décrits dans le fichier de configuration JavaScript dans le tableau *groupesDesSons*. 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** 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. **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 dans le fichier de configuration 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), prédécesseurs(7), n° de scène graphique(8) .

Pour définir le graphique il est conseillé de dessiner la partition avec un outil *draw* (Libreoffice par exemple) et de reporter les coordonnées des rectangles dans ce fichier.

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, par exemple ligne 18 à 21.

Il faut faire attention à la numérotation des prédécesseurs. Un groupe ou un réservoir peuvent être utilisés comme prédécesseur. Dans le fichier de configuration en exemple nous avons ajouté en fin de ligne le numéro du groupe en tant que prédécesseur. Jusqu'à la ligne 18 tout est simple. A la ligne 18 nous commençons un réservoir qui se termine en ligne 21. Ces 4 lignes sont dans

le même « prédécesseur » de valeur 11. Donc en ligne 22 commence le « prédécesseur » de valeur 12.

```
1. exports.groupeDesSons = [  
2.  
3.   // Pour group: nom du groupe (0), index (1), type (2), x(3), y(4), nbe d'élé  
   ments(5), color(6), prédécesseurs(7), n° de scène graphique(8)  
4.   // Pour tank: nom du groupe(0), index(1), type(2), x(3), y(4), numéro du ta  
   nk(5), color(6), prédécesseurs(7), n° de scène graphique(8)  
5.   ["violonsEchelle", 0, "group", 323, 176, 12, ocre, [3,8], 1], //0  
6.   ["violonsChrom", 1, "group", 800, 180, 16, ocre, [9], 2], //1  
7.   ["violonsTonal", 2, "group", 450, 25, 24, ocre, [30], 3], //2  
8.   ["altosEchelle", 3, "group", 200, 114, 12, violet, [5], 1], //3  
9.   ["altosChrom", 4, "group", 800, 270, 16, violet, [9], 2], //4  
10.  ["cellosEchelle", 5, "group", 52, 173, 10, vert, [], 1], //5  
11.  ["cellosChrom", 6, "group", 800, 360, 16, vert, [9], 2], //6  
12.  ["cellosTonal", 7, "group", 650, 100, 8, vert, [2], 3], //7  
13.  ["ctrebassesEchelle", 8, "group", 200, 244, 12, bleu, [5], 1], //8  
14.  ["ctrebassesChrom", 9, "group", 590, 430, 16, bleu, [25, 29], 2], //9  
15.  ["ctrebassesTonal", 10, "group", 650, 160, 6, bleu, [2], 3], //10  
16.  ["trompettesEchelle1", 11, "tank", 273, 374, 1, orange, [5,0], 1], //11  
17.  ["trompettesEchelle2", 12, "tank", 200, 10, 1, orange, [], ],  
18.  ["trompettesEchelle3", 13, "tank", 200, 10, 1, orange, [], ],  
19.  ["trompettesEchelle4", 14, "tank", 200, 10, 1, orange, [], ],  
20.  ["trompettesTonal1", 15, "tank", 650, 280, 2, orange, [2], 3], //12  
21.  ["trompettesTonal2", 16, "tank", 200, 100, 2, orange, [], ],  
22.  ["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.

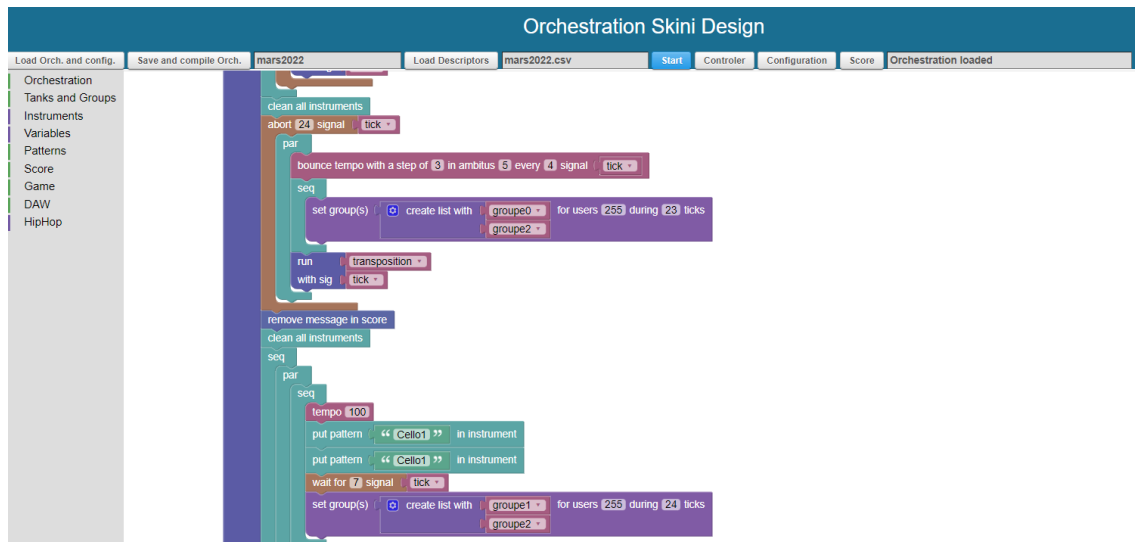
Pour lancer Skini il faut :

- Lancer : node skini

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. 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).

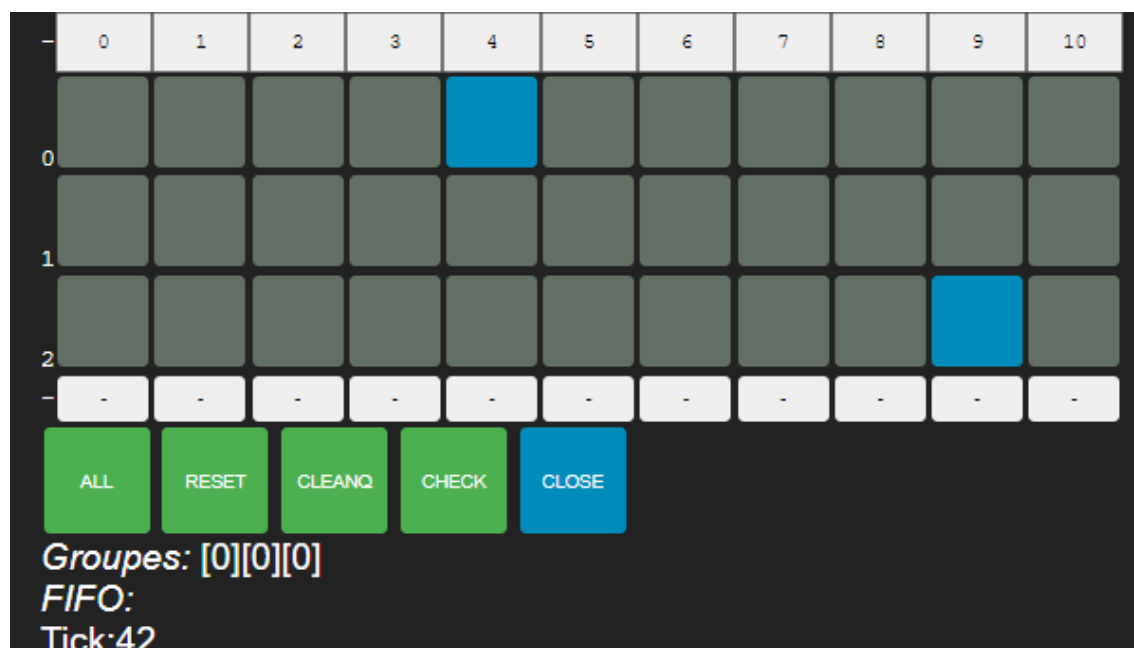
4 UTILISATION DU CONTROLEUR

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 l'adresse [http://adresse du serveur hop/conf](http://adresse_du_serveur_hop/conf)). La configuration des patterns est présentée au chapitre « Configuration des patterns ».

Skini note	Send Note			CC command	CC value	Send CC		CLOSE			
	Note	Note stop	Flag	Text	Sound file	Instrument	Slot	Type	Free	Group	Duration
1	10	510	0	Alto1	Alto1	1	0	4	0	0	4
2	11	510	0	Alto2	Alto2	1	0	4	0	0	4
3	12	510	0	Alto3	Alto3	1	0	4	0	0	8
4	13	510	0	Alto4	Alto4	1	0	4	0	0	8
5	14	510	0	Alto5	Alto5	1	0	4	0	0	8
6	15	511	0	Alto6	Alto6	1	0	4	0	0	8
7	16	511	0	Alto7	Alto7	1	0	4	0	0	4
8	20	511	0	Cello1	Cello1	2	0	4	0	1	8
9	21	511	0	Cello2	Cello2	2	0	4	0	1	8
10	22	511	0	Cello3	Cello3	2	0	4	0	1	4
11	23	511	0	Cello4	Cello4	2	0	4	0	1	8

6 UTILISATION DU CLIENT SKINI

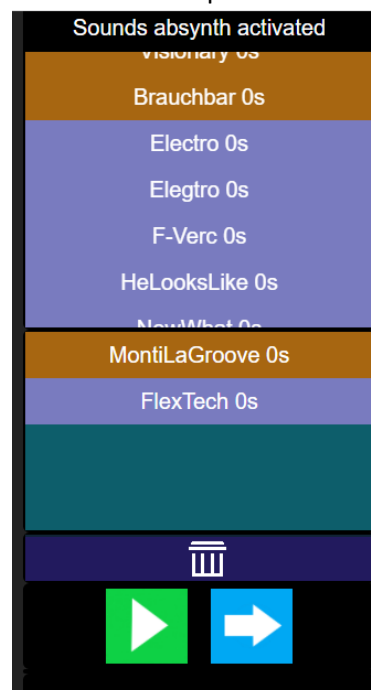
Le client qui s'appelle par l'URL [http://<adresse_serveur :port>/skini](http://<adresse_serveur_port>/skini) permet aux membres de l'audience de créer une liste de patterns et de l'envoyer au serveur. Le client peut alimenter une liste (partie inférieure de l'écran) à partir des listes de patterns disponibles (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 sont vidées 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 est un outil qui permet 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 le fichier de configuration de la pièce avec les commandes :

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

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

Le paramètre `simulatorInAseperateGroup` quand il est positionné à `true`, signifie que le dernier groupe client est réservé au simulateur. L'audience n'y aura pas accès. Positionné à `false` il signifie que le simulateur pourra se comporter comme d'importe quel groupe de l'audience.

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 `simulatorInAseperateGroup` est `true` se lance 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.

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

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

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.

limiteDureeAttente est le paramètre qui définit la durée d'attente d'un pattern au-delà de laquelle le simulateur ne fera pas appel au serveur.

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.

8 PROGRAMMATION 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.

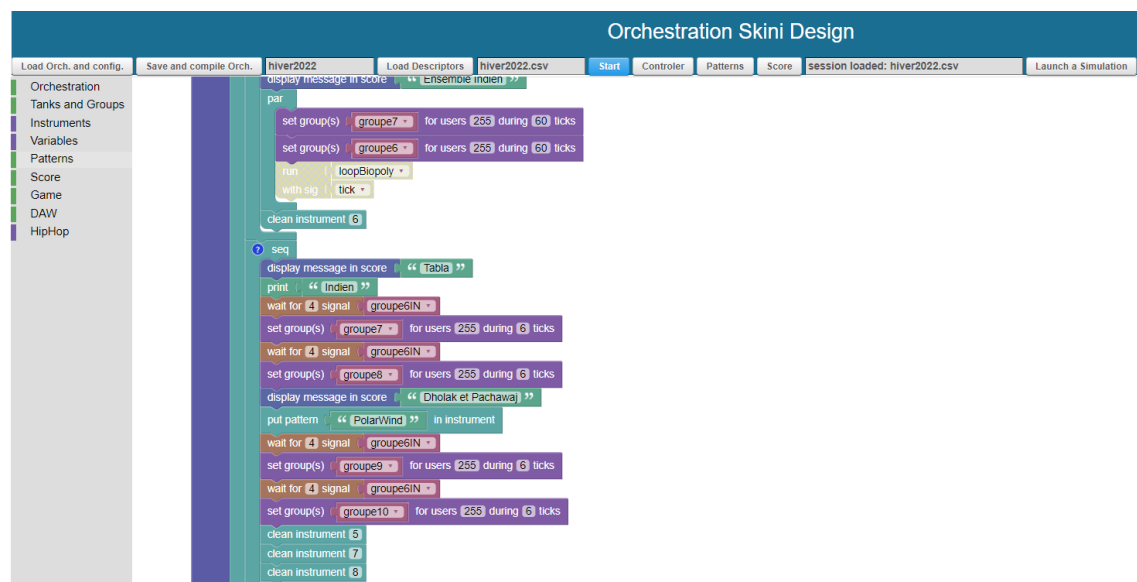
Nous n'abordons pas ici le fonctionnement de Blockly. Un passage sur le site

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

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

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.

« Pattern » 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.

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 MODULES

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

Le premier module est indispensable. « 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.



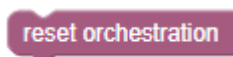
Le code blockly est organisé dans des blocs qui peuvent être mis en parallèle avec le bloc :



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.

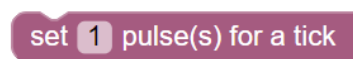


Réinitialise la matrice de l'orchestration, tous les groupes actifs sont désactivés.

8.3 LA PROGRAMMATION DU TEMPS DANS LES AUTOMATES

Le déclenchement des patterns se fait selon un mécanisme de file d'attente. Ce mécanisme a deux rôles. Il permet d'éviter le chevauchement de patterns pour un même instrument et il permet de définir une stratégie de synchronisation des tous les patterns.

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



Pour des pièces avec des durée de patterns homogènes, c'est-à-dire avec toutes les mêmes durées, il faut donner au *tick* cette durée si l'on souhaite que les départs de patterns se fassent tous aux mêmes instants.

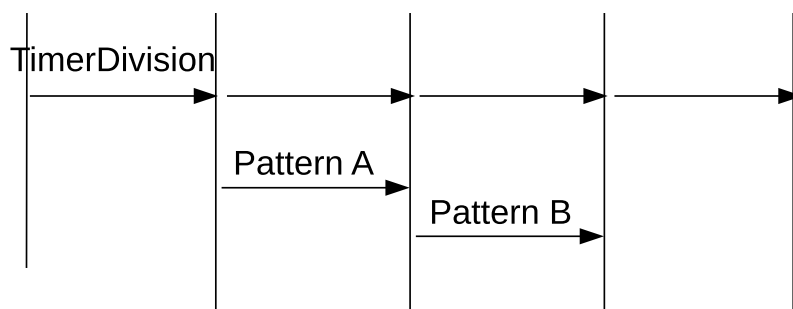
Quand on souhaite introduire des patterns de durées différentes, il est possible de donner au *tick* une valeur correspondant à la durée la plus courte d'un pattern. Ceci signifie que les patterns en s'enchainant vont tous être synchronisés entre eux sur la valeur du tick.

Pour bien comprendre le mécanisme, il faut comprendre que les files d'attente sont lues à chaque cycle du tick.

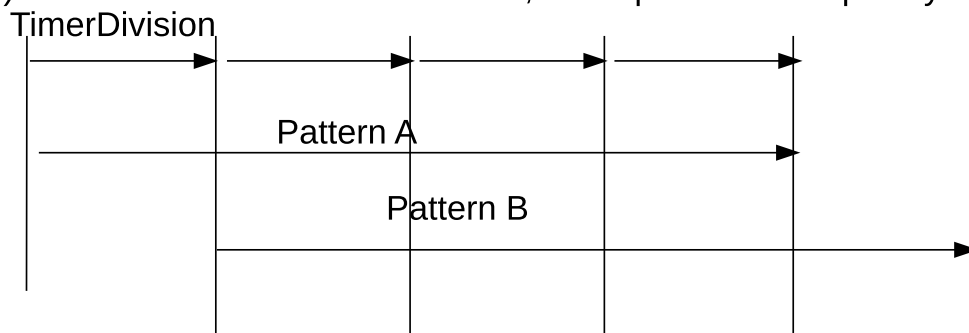
Dans le cas où des patterns ont des durées multiples du tick. Il faudra faire attention à la façon dont les patterns peuvent s'enchaîner. Si tous les patterns ont la même durée, par exemple 16 pulsations et que le *tick* est égal à 4 pulsations. Si un pattern A est sollicité à l'instant T et un pattern B à l'instant T + 5 pulsations par exemple. Le pattern B démarrera un cycle de *tick* après le pattern A car les démarrages de patterns sont synchronisés par le *tick*. Donc les patterns de 16 pulsations bien que tous de mêmes durées pourront être décalés les uns par rapport aux autres de 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 *tick*.

Remarque : Il est possible de modifier la durée du tick avec *set pulse* durant une pièce.

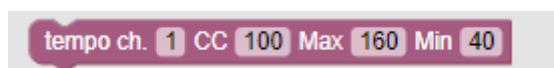
Il est aussi possible d'utiliser la *pulsation* fournie par la DAW dans une orchestration et non les *ticks* pour mesurer le temps, il faut déclarer :

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

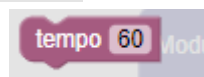
dans le fichier de configuration de la pièce et inclure un signal in *pulsation* dans l'orchestration.

8.4 TEMPO

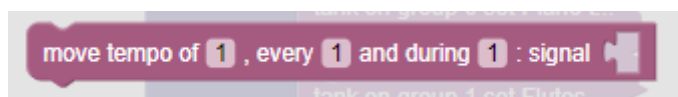
La modification des tempi depuis l'orchestration nécessite un paramétrage de la DAW qui lui permette de recevoir les Control Change sur le tempo. Nous avons vu qu'un bus MIDI de contrôle permettait d'envoyer les informations MIDI vers la DAW. C'est ce bus qui est utilisé pour les contrôles de tempo. 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. Le bloc

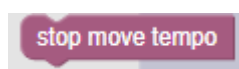


Permet d'intervenir sur le tempo à n'importe quel moment de l'orchestration. Le bloc



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 :



8.5 RESERVOIRS ET GROUPES DE PATTERNS

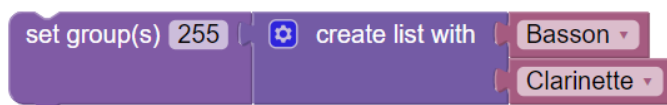
Bien qu'il s'agisse dans les deux cas de traitements d'ensemble de patterns il s'agit de deux classes de processus assez différents. Les groupes sont contrôlés à l'aide de signaux, les actions d'activation et de désactivation ne sont pas bloquantes.

8.5.1 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 :



8.5.2 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.

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

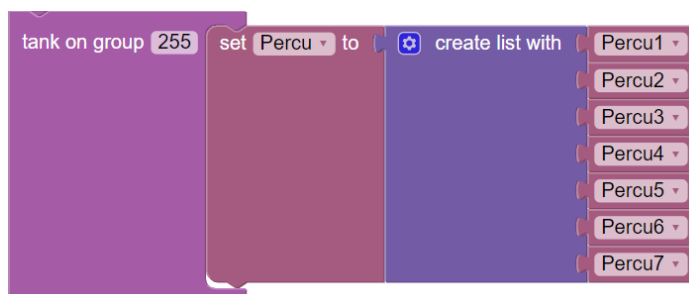
permet d'activer de façon aléatoire « max » groupes parmi un liste durant une période.

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.

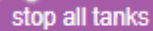
8.5.3 Création des réservoirs

Un réservoir est un bloc à trois niveaux. 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.



8.5.4 Actions sur les réservoirs

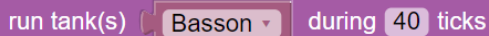
Les réservoirs (tanks en anglais) sont en fait des sous-modules Skini ayant pour paramètres des patterns. Ils sont bloquants dans un déroulement. Un réservoir s'arrêtera une fois qu'il sera vide ou qu'on l'aura tué avec

A purple block with the text "stop all tanks" in white.

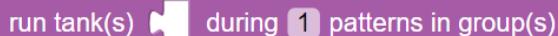
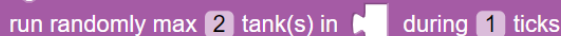
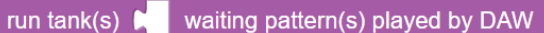
Il existe des fonctions simplifiant la gestion des réservoirs :

A purple block with the text "run tank(s)" followed by a white puzzle piece icon, then "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 :

A purple block with the text "run tank(s)" followed by a dropdown menu showing "Basson", then "during 40 ticks".

On comprend de façon intuitive le fonctionnement des blocs suivants :

A purple block with the text "run tank(s)" followed by a white puzzle piece icon, then "during 1 patterns in group(s)" followed by another white puzzle piece icon.A purple block with the text "run randomly max 2 tank(s) in" followed by a white puzzle piece icon, then "during 1 ticks".A purple block with the text "run tank(s)" followed by a white puzzle piece icon, then "waiting pattern(s) played by DAW" followed by another white puzzle piece icon.

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 **chaînes de caractère** (strings).

8.6 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.

8.6.1 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 :

A teal block with the text "clean all instruments" in white.

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 controleAbleton.js. (Le block ne réalise pas de cleanChoiceList).

Attention : les files d'attentes, 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).

8.6.2 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.

8.6.3 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.

8.7 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 :

wait for 1 pattern(s) in group NappeAlto

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.

wait for pattern (string)

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 :

wait for pattern (string) " Percu7 "

8.8 AFFICHAGE DE L'ORCHESTRATION

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

add scene score 1

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.

remove scene 1 in score

Fait disparaître un niveau

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

refresh scene score

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

display message in score

remove message in score

8.9 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 leurs longueurs 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.

8.10 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.

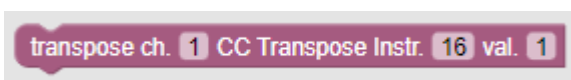
Remarques sur les commandes MIDI : De façon générale, les opérations réalisées par les blocs Skini n'interviennent pas sur les notes MIDI (au sens Skini) mais sur les groupes de patterns, les réservoirs et même les patterns mais à l'aide de leurs noms. Ceci est vrai sauf pour le blocs sendMIDI et sendCC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

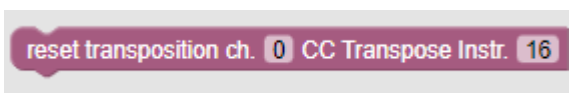
Le compositeur devra donc faire attention aux canaux MIDI dans le cas de blocs agissant sur des Control Changes (CC) et les commandes MIDI.

8.11 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.

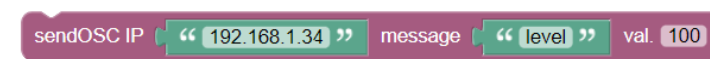
 Définit les ratio et offset pour le calcul de la transposition.

 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.

8.12 COMMANDES OSC

Pour envoyer une commande OSC depuis l'orchestration 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. Dans la version actuelle on ne passe qu'une valeur associée à la commande.

8.13 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

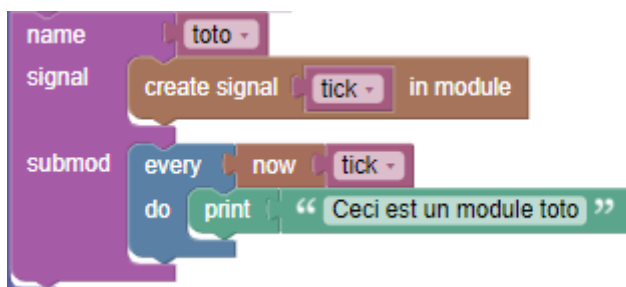
de programmer en HipHop avec Blockly. Ceci suppose de posséder les bases de la programmation réactive synchrone.

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 submod. Voici un exemple de sous-module :



Un sous-module est exécuté avec la commande run :



8.14 CONTROLE INTERFACE Z

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.



8.15 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* (cf. **Erreur ! Signet non défini.**, **Erreur ! Source du renvoi introuvable.**), 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.33.

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 ».

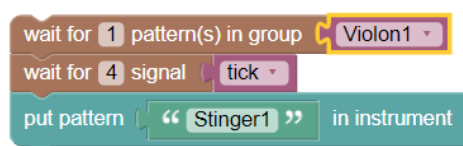
8.15.1 Cas de la réaction à l'exécution

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 les automates, p.21).

Une *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 4eme *tick*. Ici le *tick* vaudrait une pulsation.



8.15.2 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.

8.16 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.

Il est possible d'activer un traitement des files d'attente (FIFO). Il faut ajouter la commande suivant dans le fichier de configuration de la pièce :

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

(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 in 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 ». Le type P est utilisé dans des contextes de jeux interactifs, ou un joueur sélectionne un pattern qui ne sonne pas dans la pièce ou la liste des patterns du client skini).

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.

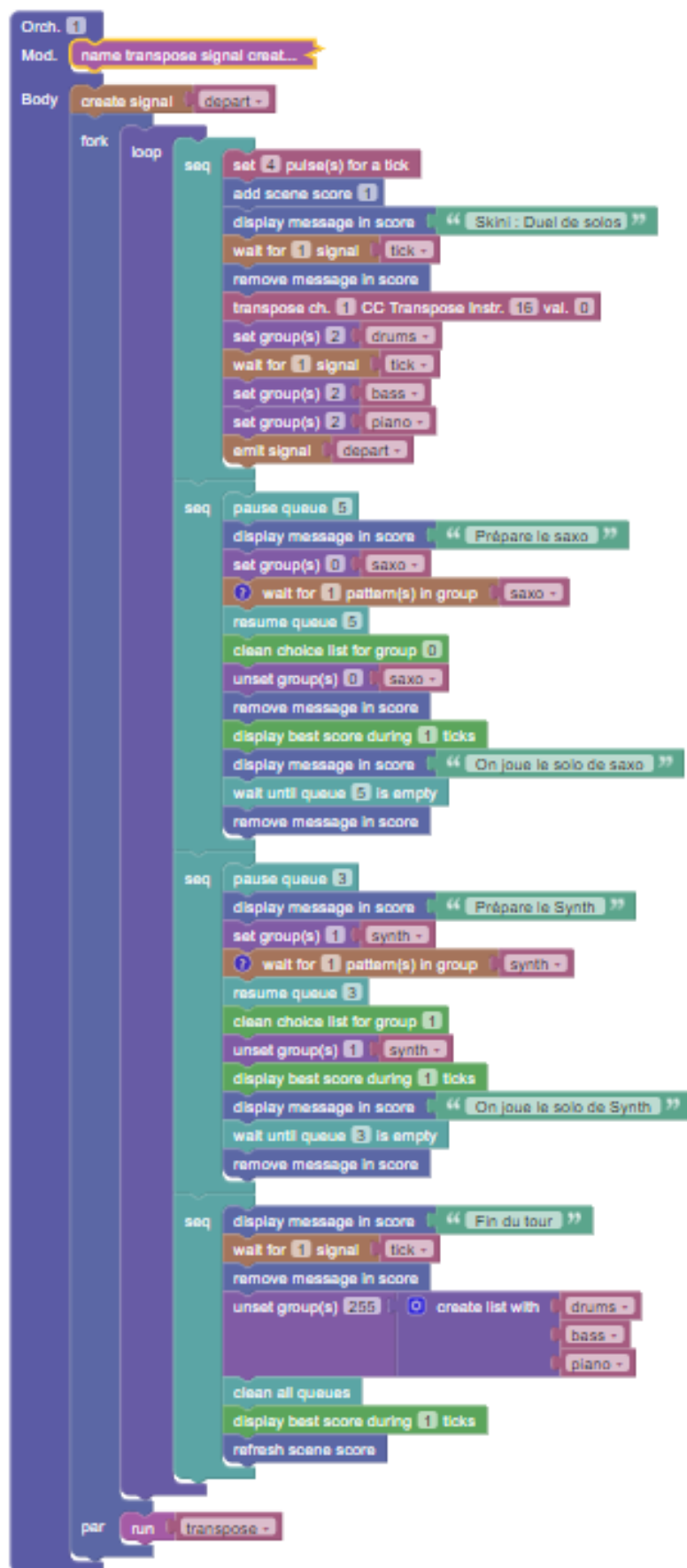
8.16.1 Exemple de programmation d'un duel de solos (revoir pour Node)

Les patterns sont utilisés uniquement sous forme de groupes. Le simulateur est utilisé sur un groupe dédié. Ceci est décrit dans le fichier de configuration de la pièce (funkBitwig) en précisant que le simulateur est associé au dernier groupe. Ici nous avons les groupes 0 et 1 pour les joueurs et le groupe 2 pour le simulateur.

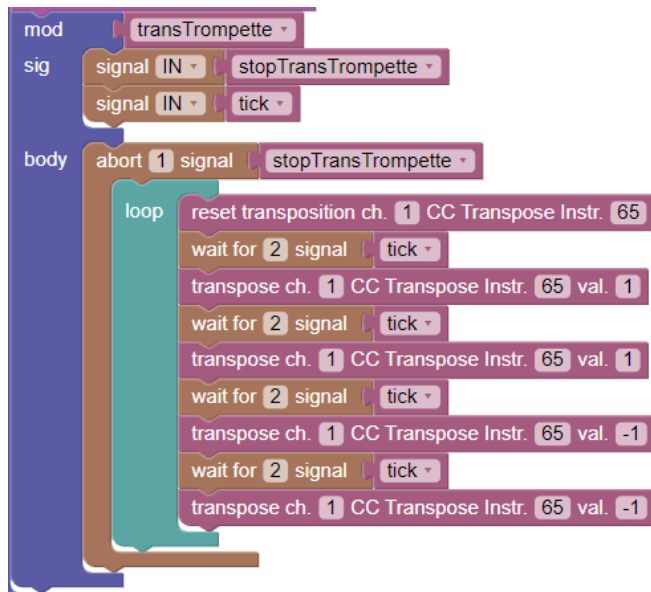
```
exports.nbeDeGroupesClients = 3;  
exports.simulatorInAseperateGroup = true;
```

Le simulateur se lance avec l'option « -sim », il joue la partie rythmique indépendamment des solistes saxo et synthé.

Tous les patterns ont la même longueur de 4 pulsations, il est donc plus économique en ressource de prendre un tick de 4 pulsations. Ceci permet de synchroniser facilement les transpositions. La limite serait sur des changements de tempo qui ne sont pas à l'ordre du jour pour cette pièce funky.



8.16.2 Exemple de transposition en boucle



9 AVEC DES MUSICIENS (REVOIR 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 ».

10 INTERFACE OSC AVEC LA PLATEFORME DE JEU UNREAL ENGINE 4 (A REVOIR AVEC NODE)

10.1 PRINCIPE

Si une plateforme de développement de jeu peut envoyer des commandes OSC, il est possible de générer des signaux dans l'orchestration à partir d'événements du jeu et inversement de générer des événement OSC dans le jeu à partir de l'orchestration.

10.2 DEPUIS L'ORCHESTRATION

Pour la mise en œuvre des signaux destinés aux échanges entre Skini et la plateforme de jeu nous utilisons un mécanisme similaire à celui utilisé pour la création des signaux à partir du fichier de configuration de la pièce, ici au moyen de deux tableaux de chaînes de caractères. Un tableau pour les messages OSC entrants et un pour les messages OSC sortants. Il y a une correspondance directe entre les messages OSC et les signaux HipHop.js. Les signaux émis et reçus peuvent avoir des valeurs. Pour les signaux reçus par l'orchestration ce sont les `react()` qui assignent directement les valeurs depuis les commandes OSC reçues.

Les déclarations des signaux sont créées dans le fichier :

```
.\serveur\autocontrôleur\automateInt.js.
```

Les *listeners* des signaux sont créés au moment de la création de l'automate d'orchestration dans :

```
.\serveur\autocontrôleur\automateInt\groupeClientsSons.js.
```

L'émission des message OSC vers la plateforme de jeu se fait avec la fonction `sendOSCGame(message, value)` dans `./serveur/logosOSCCandMidiLocal.js` appelée dans `./serveur/autocontrôleur/groupeClientsSons.js` à la création des *listeners*.

Pour les signaux entrant dans l'orchestration, les fonctions traitant la réception des commandes OSC doivent avoir accès à l'automate HipHop.js pour appeler la fonction `automatePossibleMachine.react()` comme le fait `websocketServerSkini.js`. La gestion des échanges OSC se fait dans un module spécifique `.\serveur\gameOSC.js` qui est initialisé dans `websocketServerSkini.js` car c'est dans ce module qu'est créé l'automate d'orchestration.

La mise en œuvre d'orchestration pour le jeu n'a pas besoin de laisser la possibilité de lancer plusieurs automates dans une même session Skini. Il n'y a donc qu'un ensemble de signaux « OSC » par session Skini.

Remarque sur les signaux OSC : Les signaux créés dans le fichier de configuration pour OSC ne doivent pas avoir les mêmes noms que des signaux utilisés dans l'orchestration. Il est prudent de définir un format commun à ces signaux en leur adjoignant un suffixe ou un préfixe *OSC* ou *Game* ou autre.

10.3 MISE EN ŒUVRE AVEC L'ORCHESTRATION

La mise en œuvre est très simple, le mécanisme de création des signaux pour l'orchestration se met en place si les tableaux de signaux `exports.gameOSCIn = []` et `exports.gameOSCOut = []` sont présents dans le fichier de configuration de la pièce.

Ex :

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

Il faut aussi que dans ce fichier il y ait la ligne :

```
exports.remoteIPAdressGame = ipConfig.remoteIPAdressGame;
```

Et que dans `ipConfig.js` « `remoteIPAdressGame` » soit mis à l'adresse de la plateforme de jeu.

Ex : `"remoteIPAdressGame": "192.168.1.6"`

`autoOpus4Jeu.js` est un exemple de mise en œuvre d'une orchestration avec Unreal Engine 4.

10.4 DEPUIS LA PLATEFORME HORS ORCHESTRATION

11 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 :

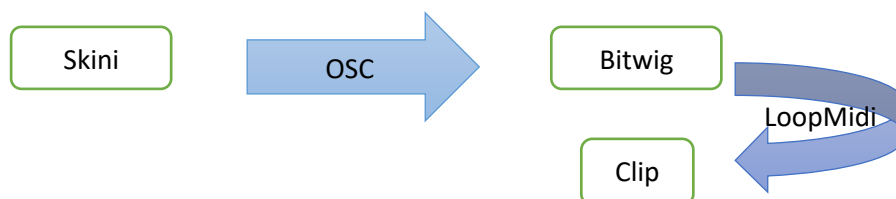
Remote OSC Ip address: 192.168.1.6

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.

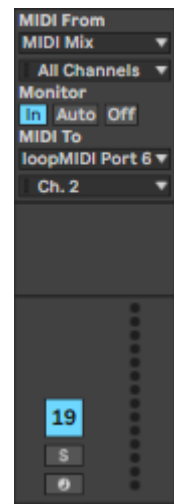
12 OSC AVEC MAX4LIVE

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.



12.1 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.

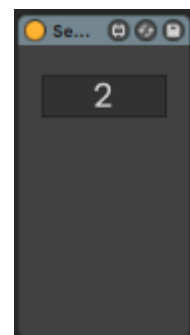


12.2 PISTE DE ROUTAGE MIDI

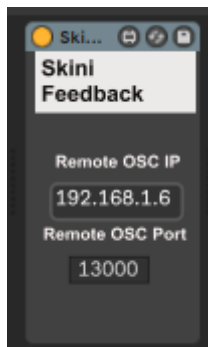


Un bug d'Ableton Live complique les opérations dès que l'on a plus de 127 notes associées aux patterns dans Skini. **M4L dans une piste Ableton ne sait pas adresser autre chose que le canal Midi numéro 1.** Pour contourner ce problème il faut créer des pistes supplémentaires, autant que de multiples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



12.3 PISTE DE FEEDBACK



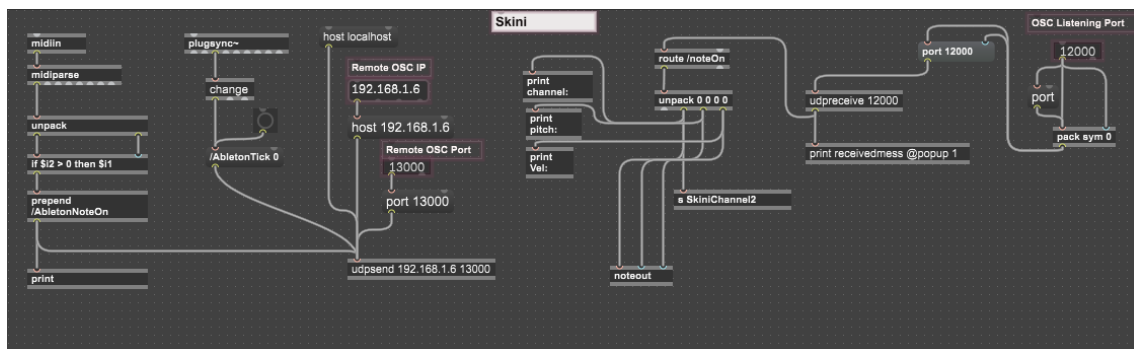
Pour traiter les informations Midi envoyées par Ableton lorsqu'un pattern est joué (cette fonction n'est utile que si on utilise le signaux *patternSignal* dans l'orchestration), il faut encore ajouter une piste Midi avec un patch M4L *SkiniAbletonFeedback.amxd*. Cette piste a pour port d'entrée (MIDI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

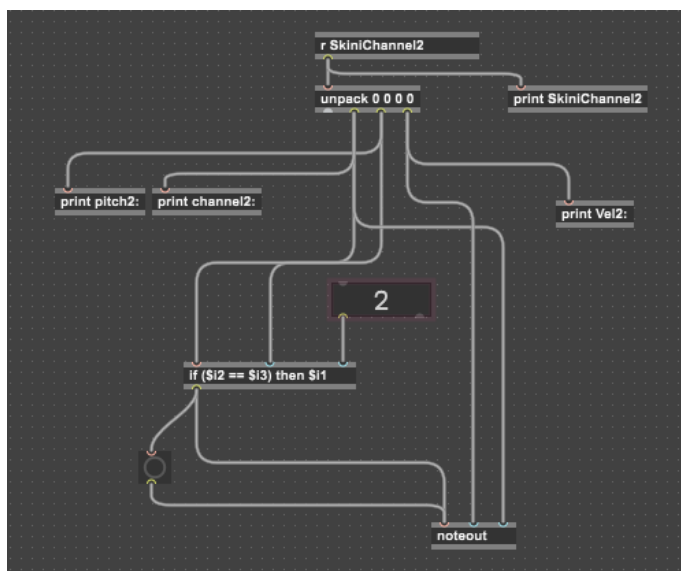


12.4 LES PATCHS M4L

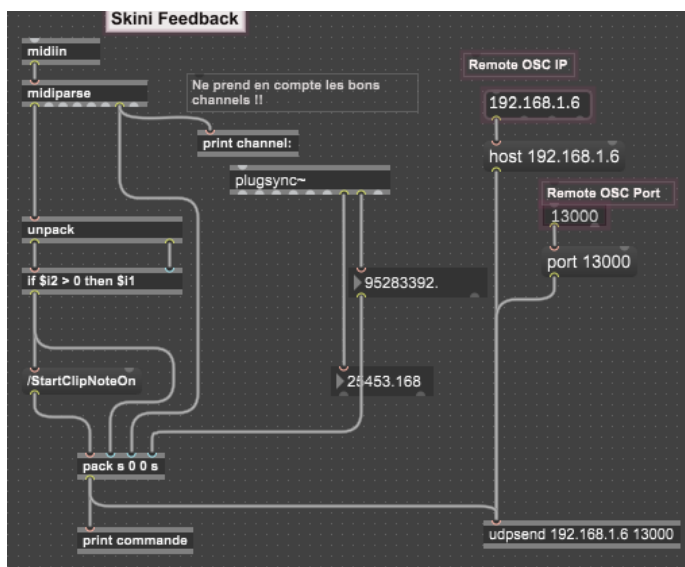
Voici le patch *SkiniOSC2.amxd* :



Le patch *SendToMIDIChannel.amxd* :



Le patch *SkiniAbletonFeedback.amxd* :



12.5 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.

13 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 ajouter dans le fichier de configuration de la pièce :

```
exports.useRaspberries = true;  
exports.playBufferMessage = 'test';  
exports.raspOSCPort = 4000;
```

Le paramètre *useRaspberries* 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 : 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 le *fichier de configuration* l'adresse IP du Raspberry est en index 11, le buffer en index 12 et le niveau en index 13.

14 ANNEXES

14.1 SKINI AVEC ABLETON LIVE

Nous avons essentiellement 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.

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.

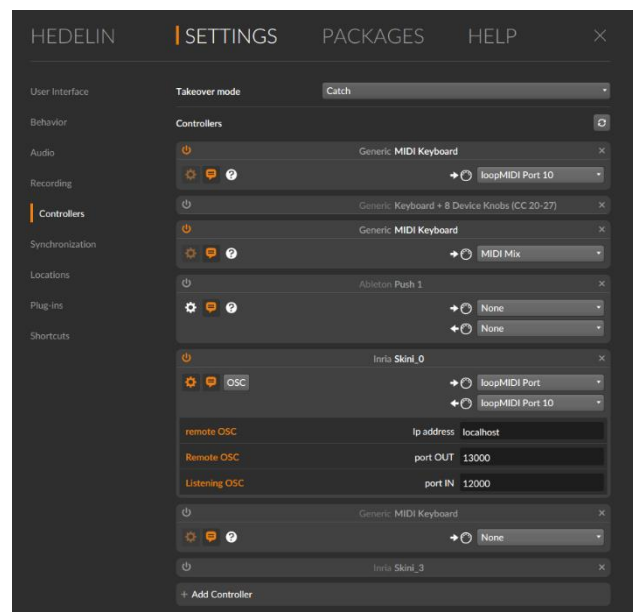
Il faut activer la synchronisation MIDI d'Ableton sur le port OUT correspondant au port IN de la passerelle Processing pour que le tempo d'Ableton contrôle Skini.

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.

14.2 SKINI AVEC BITWIG STUDIO

Avec Bitwig studio il est 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.



14.3 EXEMPLES D'ORCHESTRATIONS

trouveLaPercuNode.xml : est un exemple de jeu où l'audience doit trouver la correspondance entre des patterns et une ambiance sonore.

opus5Node.xml : est exemple de configuration qui fait appel essentiellement à des réservoirs avec des commandes MIDI de transposition, de changement de mode.

14.4 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

14.5 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.

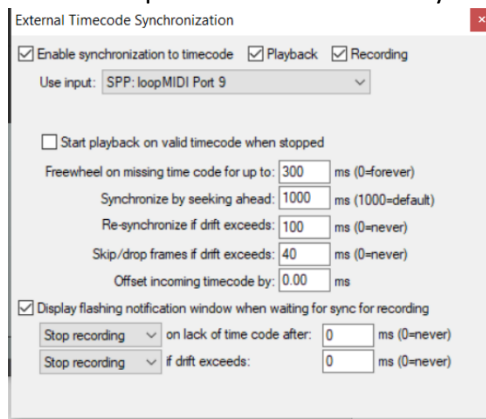
Attention aussi à la durée des patterns. Ici il faut nécessairement activer la quantification globale d'Ableton Live pour éviter les micro-décalages MIDI impossibles à retraiter. Fixer la quantification globale revient à fixer la longueur des patterns dans Skini.

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

14.5.1 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.

14.5.2 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.

14.6 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

14.7 RECEVOIR LES INFOS MIDI DE LANCEMENT DE CLIP D'ABLETON

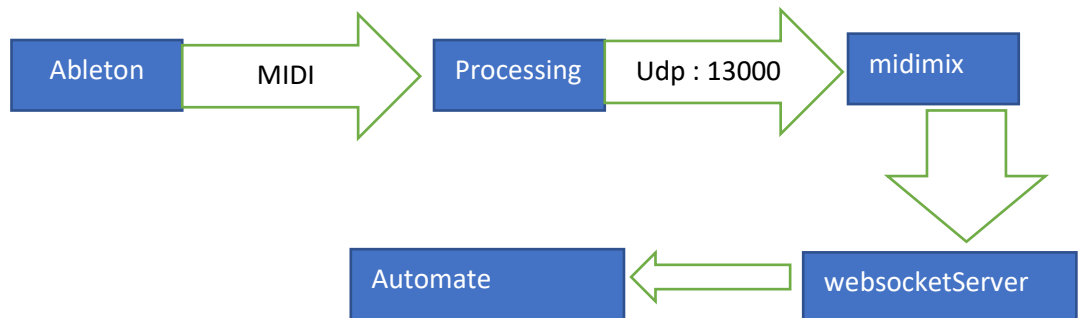
- Il faut des adresses IP dans IPconfig, pas de localhost.
- Mettre un canal MIDI out inutilisé en « Telec » (remote). Ceci revient à considérer qu'il y a une surface de contrôle sur le port (dans opus1 et 2 c'est le loopMIDI Port out 12).
- La correspondance doit être faite dans Processing (sequenceurSkini.pde)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



14.8 ENREGISTREMENT DE PATTERNS EN LIVE DANS LE SEQUENCEUR DISTRIBUE

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