COMPOSITION DES FIGURES EN TANT QUE SEQUENCES D'EVENEMENTS FINIES DECLAREES COMME VARIABLES D'ENVIRONNEMENT (DEMO)

Mario Lorenzo
CICM/Musidanse EA1572
Université Paris 8 - MSH Paris Nord.

RÉSUMÉ

Dans un but de clarification, nous montrons dans cet article un aspect de notre approche à la composition musicale basée sur l'écriture des figures en tant que séquences d'événements finies. Composées avec *SuperCollider*, nous les déclarons comme variables d'environnement sous le nom de « figure_*n* ». Elles deviennent alors des références à partir desquelles, par abstraction et héritage, emboîtement et juxtaposition, variations et transformations, nous obtenons un réseau et une collection de figures indépendantes et interdépendantes à niveaux spatio-temporels multiples.

1. INTRODUCTION

Nous présentons ici un aspect de notre approche à la création musicale en milieu informatique[1]. Plus spécifiquement, nous montrons une manière d'aborder la composition des figures à travers quelques exemples composés avec l'environnement *SuperCollider* (SC) et son langage orienté objet[2]. En faisant appel principalement aux classes *Pattern* et *Environment*, nous écrivons des séquences d'événements dont la particularité essentielle est d'avoir un début et une fin. En les déclarant par la suite comme variables d'environnement sous le nom de « figure_*n* », nous les intégrons partiellement ou intégralement, variées ou à l'identique, dans d'autres séquences, nouvelles ou déjà composées. Mises en parallèle, juxtaposées ou emboîtées, toutes les séquences composent un réseau de figures indépendantes et interdépendantes à échelles spatio-temporelles multiples et, en même temps, une collection permettant de les réintroduire dans la composition.

Bien que notre manière de faire se réfère ici à la composition électroacoustique, il est possible de l'inscrire dans le contexte de la musique instrumentale, en tenant compte, bien entendu, des spécificités de chaque domaine. D'ailleurs, un bon nombre de mes pièces pour instruments a été mené à bien en travaillant avec un grand nombre de séquences finies de tailles diverses en interaction. L'illustration 1 montre un exemple d'une composition pour ensemble :

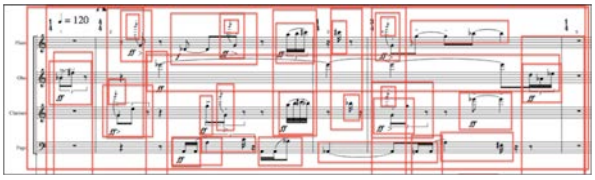

Illustration 1 Extrait de la partie des bois de ma composition *ESC* composée d'un grand nombre de figures indépendantes et interdépendantes.

1.1. Clarifier le processus de composition

Il est important de bien préciser le contexte dans lequel nous nous situons qui est celui du domaine du codage musical « sur la marche », indépendant de l'ingénierie logicielle[3]. En effet, en multipliant « manuellement » les lignes de code selon nos besoins, notre perspective n'est pas celle de la programmation mais celle de l'éclaircissement du processus de la création musicale[4].

Cela dit, SC étant un outil orienté temps réel, il est nécessaire de faire une autre remarque importante vis-à-vis de l'utilisation que nous en faisons. En effet, l'écriture « à la volée » du code demeure assujetti aux contraintes d'un travail de composition. Autrement dit, nous avons besoin, contrairement aux pratiques du codage en temps réel (*live coding*) d'arrêter régulièrement la machine pour envisager une stratégie visant une pluralité d'événements ayant leurs particularités de temps et d'espace[5].

[1] Cet article s'inscrit dans la continuation d'une recherche de clarification conceptuelle autour de nos déterminations dans le domaine de la composition musicale [1]. Il est complémentaire d'un autre texte dans lequel nous exposons le processus de composition sous l'angle de la « perception de l'aspect » [2].

[2] http://supercollider.github.io/

[3] Comme le notent Blackwell et Collins, avec les techniques du code en temps réel le codage musical devient indépendant de l'ingénierie logicielle [3].

[4] C'est dans cette perspective que nous faisons appel à SC car sa syntaxe apparaît à nos yeux particulièrement claire et bien adaptée à nos propos [4].

[5] Il ne s'agit nullement de nier l'importance pour le processus de composition d'avoir une réponse immédiate, mais d'en prendre une certaine distance. D'une part, on n'est pas toujours dépendant des sons physiques. En effet, avec un entraînement, on développe une certaine écoute interne nous permettant de nous en passer. D'autre part, la pratique temps réel, comme toute improvisation, a une portée limitée de par son attachement à la flèche du temps.

2. COMPOSITION DE FIGURES

2.1.1. Cycle écriture-réécriture: voir "comme figure" et "comme objet"

Comme nous l'avons montré en détail ailleurs [2], nous percevons le cycle écriture-réécriture propre à la composition tel que nous l'abordons, sous un double aspect. D'une manière générale, nous voyons ou entendons ce que nous sommes en train composer, tantôt « comme objet », en ce sens qu'il encapsule des données et des méthodes nous permettant d'agir sur lui, tantôt « comme figure », c'est-à-dire comme une propriété « rare » de l'objet, à savoir sa possibilité d' « atteindre un état musical satisfaisant ». C'est grâce à la perception de ce double aspect que nous pouvons construire, par étapes successives, des formes à multiples niveaux que nous donnons à entendre.

2.1.2. Voir l'objet "comme figure"

Dans ce contexte, lorsque nous disons « voir l'objet comme figure », nous définissons « figure » ainsi : « un objet multiple que je perçois comme une unité musicale qui, par un acte d'écriture effectif, devient indépendante et référente pour de nouvelles opérations, à l'intérieur de l'environnement en cours » [2]. Cela dit, cette unité étant « transparente », dans ce sens qu'elle montre ses données et ses méthodes [5], elle peut à tout moment être perçue à nouveau comme objet et de ce fait, traité comme tel[6].

2.1.3. "Comme objet"

Dans ce qui suit nous nous concentrons davantage justement sur l'aspect « objet » de ce que nous composons, sous l'angle du langage orienté objet. Toujours dans un but de clarification, cela signifie que nous percevons des « instances d'une classe », ayant des « attributs » qu'on manipule à travers des « méthodes » en leur envoyant des « messages » avec des « arguments ». Le but étant de leur donner un « état », ou plus exactement, un état satisfaisant qui pourra nous permettre de définir une figure et avec elle un réseau et une collection de figures.

3. SEQUENCE D'EVENEMENTS FINIE

Pour y arriver, il faut commencer par dire que, pour nous, toute figure, vue comme objet est : une séquence d'événements finies, déclarée comme variable d'environnement et, de ce fait, référente pour de nouvelles figures.

[6] C'est la raison pour laquelle nous utilisons, presque exclusivement, le langage alphanumérique car, contrairement au langage graphique (gui), il se montre « complètement », en ce sens qu'on voit les données et les méthodes.

3.1. Séquences d'événements finies

3.1.1. Stream

Pour créer une séquence (*stream*) en SC il suffit d'envoyer à n'importe quel objet le message *.next* autant de fois que l'on veut quand on veut, indépendamment de toute notion temporelle.

```
1 (
2 7.do {
3     arg a; a = "événement";
4     a.next;
5 };
6 )
```

Illustration 2 Une séquence où l'on répète 7 fois l'objet "événement"

3.1.2. Event

Il faut noter que si l'on veut faire de la musique, un événement (*event*) est plus qu'une valeur ou qu'un simple objet texte comme on vient de le voir. En effet, un événement est une collection des couples clé/valeur (*Event* est une sous-classe d'*IdentityDictionary*). La clé (ou le nom) représente un paramètre, qu'il soit un paramètre par défaut (fréquence, l'amplitude, etc.), soit un argument spécifique d'un ou plusieurs objets qui génèrent la synthèse[7].

3.1.3. Un objet dans le serveur

Cela signifie qu'il faut faire appel aux abstractions qui représentent le serveur où la synthèse est générée[8]. Dans l'illustration suivante nous nous servons principalement de la classe *Buffer*, *SynthDef* et *Dictionary* pour définir un objet.

```
1 //==================BUFFERS
2 (
3 d = Dictionary.new;
4 d.add(\quit -> PathName ("/Users/.../5_01/").entries.collect ({
5     arg sf;
6     Buffer.read(s, sf.fullPath);
7     });
8 );
9 )
10 x = Buffer.sendCollection(s, Signal.hammingWindow(1024));
11 y = Buffer.sendCollection(s, Env.perc.discretize, 1);
12
13 //================SYNTHDEF
14
15 (
16 SynthDef(\grainbuf_01, {
17     arg out = 0, pos = 0, buf = 0, windowbuf = 1, graindur = 0.2,
18     rate = 1, loop = 1, pan = 0, amp = 1;
19     var window, src;
20     src = PlayBuf.ar(
21         1,
22         buf,
23         BufRateScale.kr(buf) * rate,1,
24         round(pos * BufFrames.kr(buf)),
25         loop,
26         2);
27     window = BufRd.ar(
28         1,
29         windowbuf,
30         EnvGen.ar(Env([0, BufFrames.kr(windowbuf)], [graindur]),doneAction: 2),
31         loop,
32         4);
33     OffsetOut.ar(out, Pan2.ar(src, pan, amp) * window);
34 }).add;
35 )
```

Illustration 3 Définition d'un objet avec les classes représentant le serveur.

[7] On peut aussi créer une clé pour mettre en relation les valeurs entre les paramètres.

[8] À propos de l'archtecture de SC, cf. http://doc.sccode.org/Guides/ClientVsServer.html

Une fois les « bords définis » [5], l'objet peut être instancié autant que l'on veut à travers une liste d'événements avec l'objet *Synthé* ou la classe *Pattern*. Comme nous le verrons plus bas, c'est avec cette dernière que nous composons un grand nombre de séquences d'événements (*Event Stream*) venant donner à l'objet autant d'états.

3.1.4. Générer des séquences

S'il est possible d'écrire événement par événement, on peut également, ce qui est très utile, se servir de différents mécanismes pour générer le nombre d'événements que l'on veut. Pour cela on peut écrire une fonction avec la classe *Routine*.

```
1 (
2 var a;
3 a = Routine ({
4     var dictionnaire;
5     dictionnaire=([[freq: rrand (200, 210)], [dur:0.5]]);
6     2.do({dictionnaire.yield;});
7 });
8 a.nextN(2).postln;
9 )
```

Illustration 4 Génération des événements mécaniquement avec *Routine*

3.1.5. La classe Pattern

Cela dit, il y a un moyen dans SC de créer de multiples routines depuis un modèle unique, ce qui nous évite d'écrire les fonctions, en général longues et compliquées, et nous permet de nous concentrer, grâce à une syntaxe réduite, sur les séquences elles-mêmes et leurs interactions. Il s'agit de la classe *Pattern*. Un pattern n'est pas une séquence mais une représentation de haut-niveau pouvant retourner séquentiellement des objets d'une série d'expressions depuis un modèle particulier ou sous-classe[9].

```
1 Pseq ([0.06, 0.17, 0.36, 0.6, 0.11, 0.7], 2);
```

Illustration 5 Une liste de valeurs, répétée deux fois avec un *Pseq*

3.1.6. Pbind : séquence d'événements

Pbind c'est le pattern de base qui permet de lier plusieurs séquences de valeurs en une seule séquence d'événements (*Event Pattern* ou *EventStreamPlayer*). Il est constitué d'une série de paires clé/valeur. Pour chaque message *.next*, on obtient un objet *Event*, c'est-à-dire, l'ensemble des paramètres et des valeurs que nous envoyons au serveur. Dans l'illustration 6, la variable p se réfère à un *Pbind* qui inclut trois autres patterns : *Pseq*, qui envoie une séquence fixe de valeurs à la clé \dur3 (ligne 4); *Prand* envoie les valeurs de la liste à la fréquence, avec une fonction aléatoire (ligne 5) ; du même que *Pwhite* dont les valeurs envoyées à l'amplitude sont entre 0.01 et 0.6. (ligne 6).

```
1 (
2 p = Pbind (
3     \instrument, \un_synth,
4     \dur, Pseq ([0.06, 0.17, 0.36, 0.31,0.7], inf),
5     \freq, Prand ([497, 502, 505, 499], inf),
6     \amp, Pwhite (0.01,0.6, inf)
7 )
8 )
9 p.play;
```

Illustration 6 un *Pbind* qui se réfère à l'instrument « un_syhth » et à ses paramètres avec trois autres patterns. Il génère une séquence d'événements.

3.1.7. Clock

Pour préciser le temps des événements, il faut faire appel à l'une des méthodes de la classe *Clock*. Elle permet de planifier une tâche dans un point arbitraire du temps. N'importe quel objet peut être « réveillé » à n'importe quel moment (*awake method*)[10]. On peut se servir, entre autres, de l'objet *Routine* et le message *.play* et *.wait* comme unité de temps pour la planification des événements. En tant qu'abstraction, les patterns incluent ces objets et ces méthodes.

3.2. Séquence Finie

La particularité des séquences d'événements que nous générons à travers les patterns pour composer les figures, réside dans le fait qu'elles sont du type « finie » (*finite stream*). En effet, il est essentiel pour notre démarche cherchant à construire des formes ayant une multiplicité de niveaux temporels que chaque séquence ait une définition de durée globale. Pour cela, il faut le préciser, car autrement on obtient des séquences qui ne s'arrêtent pas (*infinite stream*). Dans la ligne 9 du code que nous venons de voir dans l'illustration 6, on associe la variable (p) à la méthode *play*. Quand on l'exécute, on obtient une séquence infinie (*infinite streams*) d'événements, car nul part nous n'avons déterminé un arrêt. En effet, tous les patterns de cet exemple prennent comme deuxième argument la valeur *inf* (infini). Autrement dit, on envoie aux objets le message *.next* sans arrêt. Une solution serait d'utiliser la méthode *.stop* (o.*stop*), mais cela ne définirait pas la durée de la séquence puisque son évaluation dépendrait de notre action en temps réel.

3.2.1. Variable de répétition

Pour composer une séquence d'événements finie, il faut le préciser. C'est-à-dire que nous devons assigner

[9] La classe *Pattern* est associée à un vaste ensemble de sous-classes. Sans compter les extensions, il y a dans SC plus de 120 sous-classes de patterns. Elles ont la particularité de commencer par la lettre P. (Pbind, Pseq, Prand, etc.)

[10] L'un des avantages de SC3 est le fait que, par rapport à la version 2, on peut créer séparément autant de *Synths* que l'on veut au moment où on le veut. Pour chaque événement, SC crée un *Synth* indépendant et le ferme lorsque l'événement finit. Cf. http://doc.sccode.org/Guides/Spawning.html

une valeur à la variable de répétition (*repeats variable*), dont la plupart des patterns peuvent recevoir, indépendamment du fait qu'ils soient joués ou non. Dans l'illustration suivante nous avons une séquence qui s'arrête au bout de 6 événements (3 valeurs * 2), car le septième message *next* (ligne 2) retourne la valeur nil.

```
1 z = Pseq ([1, 2, 3], 2).asStream;
2 z.nextN (7);//-> [ 1, 2, 3, 1, 2, 3, nil ]
```

Illustration 7 Une séquence finie dans laquelle on répète deux fois la liste.

Lorsqu'il y a plusieurs séquences dans un *Pbind*, si l'une est finie, les autres s'arrêtent avec elle, même si celles-là sont définies comme infinies. Dans l'illustration 8 *Pseq* détermine le nombre d'événements de toutes les autres séquences, car, une fois les cinq valeurs passées, le pattern s'arrête. La dernière valeur de la séquence liée à la fréquence est 497 (ligne 5).

```
1 (
2 p = Pbind (
3     \instrument, \un_synth,
4     \dur, Pseq ([0.06, 0.17, 0.36, 0.31,0.7], 1),
5     \freq, Prand ([497, 502, 505, 499], inf),
6     \amp, Pwhite (0.01,0.6, inf)
7 ).play;
8 )
```

Illustration 8 Une séquence d'événements finie

3.3. Environnement

Dans SC, un environnement (*Environment*) est un dictionnaire (doté de clés et de valeurs), permettant de créer un contexte particulier. Il existe un environnement (global) par défaut au lancement de l'application: *currentEnvironment*

3.3.1. Variable d'environnement

Une variable d'environnement est une nouvelle définition qu'on peut donner à l'environnement en cours en ajoutant un dictionnaire, avec un nom propre et une valeur, un ensemble de valeurs ou d'événements.

```
1 currentEnvironment.put ("ma_variable", 5);
2 ~ma_variable = 5;
```

Illustration 9 Deux manières d'écrire une variable d'environnement

C'est une variable dans ce sens qu'elle peut être invoquée dans n'importe quel contexte, à condition que l'environnement global ne soit pas redéfini[11]. Dans l'illustration suivante nous déclarons deux variables d'environnement, dont la deuxième inclut la première.

```
1 (
2 ~ma_variable_01 = Pbind (
3     \freq, Pwhite (30, 35),
4     \dur, Pxrand ([0.1, 0.7, 0.2, Rest (1)],
5 );
6 )
7 ~ma_variable_02 = Pn (~ma_variable_01, 5);
```

Illustration 10 Deux variables d'environnement

3.3.2. La figure déclarée comme variable d'environnement

Lorsque nous disons que nous voyons l'objet comme figure, la figure correspond au fait que nous déclarons la séquence finie, par un acte d'écriture effectif, comme variable d'environnement, en lui donnant le nom « ~f » [2]. Des deux séquences de l'illustration 11, à première vue « identiques », nous voyons seulement la deuxième comme figure puisque nous la déclarons, effectivement, comme variable d'environnement en lui donnant le nom.

```
1 Pbind (\dur, Pn (rrand (0.1, 0.13), 5));
2 ~f = Pbind (\dur, Pn (rrand (0.1, 0.13), 5));
```

Illustration 11 La même séquence, dont une est vue comme figure, déclarée comme variable et nommée comme telle.

3.4. Pattern de référence

3.4.1. Pdef

Si nous déclarons une figure comme variable d'environnement c'est, en effet, parce que nous avons l'intention de l'utiliser ailleurs. Cela dit, une fois qu'on a défini une séquence d'événements (*Pbind*) dans SC, on ne peut pas accéder à son état interne dans d'autres contextes. On peut l'appeler telle quelle. Pour la varier ou l'utiliser partiellement, nous l'associons à un pattern de référence (Pdef) de la bibliothèque *JITlib*[12]. Un *Pdef* (ou *PatternProxy*), véhicule un environnement local, c'est-à-dire, un *IdentityDictionary* auquel nous avons accès pour varier ou pour utiliser partiellement.

```
1 (
2 ~f_01 = Pdef (\f_01,
3     Pbind (\dur, Pn (0.11, 3))
4 );
5 )
```

Illustration 12 La variable ~f_01 associée à un pattern de référence (Pdef)

3.4.2. « Figure de référence »

Dans cette perspective, une figure, en tant qu'unité perceptuelle déclarée comme variable d'environnement et associé à un *Pdef* peut être vue également comme « figure de référence » en ce sens que c'est à partir de

[11] C'est la raison pour laquelle on la considère comme « pseudo-variable ».

[12] http://doc.sccode.org/Overviews/JITLib.html

cette dernière que nous allons nous concentrer pour multiplier des variantes, dans le même sens qu'un thème est varié. Cela dit, il faut être prudent avec l'analogie puisque nous ne faisons pas seulement des variations et la situation est toute autre.

3.4.3. Figures (~f_n)

En effet, selon notre approche, et si nous reprenons l'illustration 12, ~f_01 n'est pas la source d'un changement quelconque seulement (qu'on pourrait noter ~f_01b, ~f_01c, etc.), mais le point de départ d'autre chose, d'une autre forme, même si, du point de vue de l'objet, elle hérite de ses attributs. Autrement dit, avec les changements à partir d'une figure de référence nous visons, en dernière instance, une autre figure qui peut être, à son tour, référence. Lorsque c'est le cas, nous additionnons 1 (~f_*n*+1). Dans l'illustration 13 nous introduisons ~f_01 dans un autre pattern (*Pbindf*) nous permettant d'ajouter un paramètre. Une fois vue comme nouvelle figure, nous la déclarons comme variable avec le nom ~f_02.

```
1 ~f_02 = Pdef (\f_02,
2     Pbindf (~f_01,
3         \amp, Pseq ([0.05, 0.1, 0.2],1))
4 );
```

Illustration 13 Une nouvelle figure à partir d'une « figure de référence »

Cela signifie, entre autres, que la figure n'est pas pour nous une forme à suivre (comme serait un thème), mais, en tant qu'objet, un ensemble d'attributs pouvant être utilisé, comme nous allons le voir plus bas, partiellement ou totalement, varié ou complètement transformé dans d'autres contextes, en vue de la composition d'une nouvelle figure.

3.5. Séquence, figure, environnement

3.5.1. La figure modifie l'environnement global

Insistons sur le fait qu'en ajoutant une figure en tant que variable associé à un *Pdef*, nous modifions l'environnement.

```
1 Pbind (\dur, Pn (0.11, 3));
2 //-> Environment[  ]
3 (
4 ~f_01 = Pdef (\figure_01,
5     Pbind (\dur, Pn (0.11, 3))
6 );
7 )
8 // -> Environment[ (f_01 -> Pdef('figure_01')) ]
```

Illustration 14 La figure modifie l'environnement en cours.

Nous obtenons ainsi un nouveau contexte. L'occasion de voir l'environnement que nous sommes en train de changer et de réfléchir, « hors environnement », à de nouvelles séquences, à de nouvelles figures. Autrement dit, il ne s'agit pas de collectionner des figures indépendamment de ce qu'elles modifient.

```
1 (
2 ~f_02 = Pdef (\figure_02,
3     Pseq ([~f01, Rest (0.5)], 2)
4 );
5 )
6 (
7 ~f_03 = Pdef (\figure_03,
8     Pbindf (~f01, \dur, Pseq ([0.2, 0.5, 0.3], 1))
9 );
10 )
11 (
12 ~f_04 = Pdef (\figure_04,
13     Ptpar ([
14         0.0, Pbindf (~f01, \pan, -1),
15         0.03, Pbindf (~f01, \pan, 1)]
16         , 1)
17 );
18 )
19 /* -> Environment [
20 (f_01 -> Pdef('figure_01')),
21 (f_02 -> Pdef ('figure_02')),
22 (f_03 -> Pdef('figure_03')),
23 (f_04 -> Pdef('figure_04'))
24  ]
25 */
```

Illustration 15 Composition de nouvelles figures en fonction des modifications de l'environnement.

4. COMPOSITION DES FIGURES A PARTIR DE SEQUENCES D'EVENEMENTS FINIES DECLAREES COMME VARIABLES D'ENVIRONNEMENT

Nous montrons dans cette partie quelques exemples tirés d'une composition réalisée, toujours dans un souci de clarification, entièrement en SC. D'abord nous regarderons les étapes pour la composition d'une première figure, puis, quelques mécanismes clés à travers différents patterns pour la création de nouvelles figures.

4.1. Composition d'une figure

Pourquoi nous voyons une séquence d'événements finie comme une figure et non, par exemple, une « phrase », un « geste », un « motif », etc ? Les raisons à cela sont sans doutes nombreuses. Dans le cadre de cet article, limitons nous à dire que, derrière la notion de figure, il y a un savoir-faire polyphonique de plusieurs siècles. C'est peut-être l'une des raisons pour lesquelles nombreux sont les compositeurs qui, non sans nuances importantes, continuent à en faire appel[13]. On pourrait ajouter à cela que le mot figure nous dit quelque chose de plus spécifique que ne le fait le mot objet. En effet, du point de vue de l'utilisation ordinaire du langage,

[13] Nous nous inspirons en bonne partie des avancées théoriques de H. Vaggione [5]. Nous rétenons aussi le travail du compositeur Carlos Caires et de son logiciel *Irin* dans lequel la figure occupe une place majeure [6]. Bien que notre approche est assez différent en ce sens que nous ne cherchons pas à définir une classe, ni a lui donner une hierarchie, le principe d'écriture et réécriture, d'encapsulation, d'héritage et de réseau motivent notre recherche dans une direction similaire. Il est a signaler également l'importance pour notre réflexion des recherches de Guilherme Carvalho sur le rapprochement entre la figure géométrique projective et la composition musicale [7].

« figure » renvoie davantage à l'aspect perceptuel de la forme en tant que singularité, tandis qu'« objet » se réfère plutôt à une généralité.

Quoi qu'il en soit de cette première réponse à un sujet aussi vaste dont nous n'avons pas la prétention d'épuiser en quelques lignes, regardons de plus près les étapes de la composition d'une figure en tant qu'état possible de l'objet défini préalablement.

4.1.1. Buffer

1 - Nous commençons par déclarer deux variables globales « b » et « y » correspondant, respectivement, à un fichier son[14] et à une enveloppe de type percussion, qui représentent le buffer dans le serveur.

```
(
b = Buffer.read(s, Platform.resourceDir +/+ "Sounds" +/+ "68-v05.wav");
y = Buffer.sendCollection(s, Env.perc.discretize, 1);
)
```

Illustration 16 Définition des deux variables (b, y) avec la classe *Buffer*

2 - Ensuite nous définissons un synthétiseur (illustration 17). Il s'agit d'un moteur granulaire qui nous permettra de traiter le fichier son stocké dans la mémoire. Il a deux variables « sr » (source) et « window » (fenêtrage). La source est composée principalement de l'Ugen *PlayBuf* et la génération d'une fenêtre avec *BufRd* et *EnvGen*[15]. Il prend huit arguments (out, pos, buf, windowbuf, graindur, rate, pan, amp) et un nom (« bufgrain_01 ») qui recevront les messages des patterns.

```
(
SynthDef(\grainbuf_01, {
    arg out = 0, pos = 0, buf = 0, windowbuf = 1, graindur = 0.2,
    rate = 1, loop = 1, pan = 0, amp = 1;
    var window, src;
    src = PlayBuf.ar(
        1,
        buf,
        BufRateScale.kr(buf) * rate,
        1,
        round(pos * BufFrames.kr(buf)),
        loop,
        2);
    window = BufRd.ar(
        1,
        windowbuf,
        EnvGen.ar(Env([0, BufFrames.kr(windowbuf)], [graindur]),doneAction: 2),
        loop,
        4);
    OffsetOut.ar(out, Pan2.ar(src, pan, amp) * window);
}).add;
)
```

Illustration 17 Définition du *SynthDef* « grainbuf_01 » dans le serveur auquel nous enverrons les valeurs à travers les patterns pour générer les séquences d'événements.

3 - Et finalement, la composition de la figure proprement dite (~f_01)

```
(
~f_01 = Pdef (\f_01,
    Pbind (
        \instrument, \grainbuf_01,
        \graindur, 0.05,
        \dur, 0.01,
        \windowbuf, y,
        \buf, b,
        \amp, 0.9,
        \pos, Pseq(
            Pseq ([0.0, 0.8, 0.7], inf),
            Pseq ([0.15, 5,0.4],1),
            \lin),
        \rate, Pwhite (3.98, 4.02, inf)
    )
).play;
)
```

Illustration 18 Une première figure

4.2. Nouvelles figures

4.2.1. Abstraction et héritage

En déclarant la figure de référence comme variables d'environnement, nous nous servons, des deux concepts majeurs du langage orienté objet : l'abstraction et l'héritage. Puisque, comme nous l'avons noté, notre démarche est celle de la composition plutôt que de la programmation, nous ne cherchons pas ici à utiliser ces concepts pour créer de nouvelles classes mais des objets singuliers, de nouvelles figures.

4.2.2. Modification ou ajout des arguments (Pbindf)

Le pattern *Pbindf* nous permet de varier une figure en rappelant son nom, puis en ajoutant ou en modifiant des arguments.

```
~f_19 = Pdef (\f_19,
    Ptpar ([
        {0.031.rand}, Pbindf (~f_18, \pan, -1),
        {0.031.rand}, Pbindf (~f_18, \pan, 1)], 1));

~f_20 = Pdef (\f_20,Pbindf (~f_19, \graindur, Pgeom (0.03, 1.1, 32)));
```

Illustration 19 deux figures héritant des attributs des figures précédentes

Il y a un grand nombre de sous-classes permettant de varier une séquence, parmi celles que nous avons utilisées, *Pfin*, *Pstrech*, *Pwrand*, *Prand*, etc.

4.2.3. Juxtaposition (Pseq)

Avec *Pseq*, nous mettons une figure après l'autre

```
~f_38 = Pdef (\f_38,
    Pseq ([~f_35,~f_29,~f_37,~f_29,~f_41,
        Pseq ([~f_16,Rest (4.5)], 2),Rest (5.2),~f_42], 1)
);
```

Illustration 20 Juxtaposition des figures

4.2.4. Décalages des figures (Ptpar)

La sous-classe *Ptpar* prend comme premier argument une liste de temps et de patterns en parallèle (*time*,

[14] Il s'agit d'un son de guitare classique, plus précisément, la note Sol# jouée sur la première corde, mf, et d'une durée de 1.56 seconde. J'ai volontairement choisi un son très simple par rapport à sa morphologie, pour mettre en valeur le travail avec les patterns. Il va sans dire que composer avec des sons plus complexes enrichit davantage notre palette.

[15] Héritée des programmes *Music n* de Max Mathews, l'expression *unit generator*, désignant une unité de génération, de lecture, d'enregistrement ou de traitement de signal, est reprise dans plusieurs environnements d'informatique musicale, dont SC avec la classe *UGen*.

pattern, *time*, *pattern*, etc.). Elle nous permet de mettre en parallèle des figures avec un décalage temporel. Nous l'utilisons, entre autres, pour donner des attributs d'espace aux figures. En suivant les avancés théoriques d'Horacio Vaggione [8], nous considérons une figure spatiale, comme une séquence de deux figures ou plus, étant décalées par des intervalles microtemporels et ayant des sorties différentes.

```
~f_24 = Pdef (\f_24,
    Ptpar ([
        0.0, ~f_18,
        0.02, ~f_19
], 1)
);
```

Illustration 21 Décalage de deux figures

4.2.5. Mise en parallèle (Ppar)

Nous mettons plusieurs figures strictement en parallèle, avec *Ppar*, lorsqu'elles sont déjà décalées. Ce qui justifie que nous n'utilisons pas *Ptpar*.

```
(
~f_04 = Pdef (\fig_04,
    Ppar ([~f_01,~f_02,~f_03
    ], 1)
).play;
)
```

Illustration 22 Trois figures, déjà décalées, en parallèle

4.2.6. Mécanisation (Pspawn)

Pspawn fait partie, sans doute, des abstractions très puissantes dans SC, mais son utilisation, comme tous les automatismes, demande un certain dosage. Les « séquences enfants » (*child events*) engendrées ne sont pas indépendantes. Ce sont des copies de la « séquence mère » (*parent events*) qu'on peut reproduire autant que l'on veut et placer, à des distances voulues, mais on ne peut pas les modifier individuellement et les utiliser dans d'autres contextes.

```
(
~f_05 = Pdef (\fig_05,
    Pspawn (
        Pbind(
            \method,\par,
            \pattern, Pfunc {
                Pbindf (~f_03,
                    \amp, 0.15
                )

            },
            \delta, Pn (0, 8)),
)
).play;
)
```

Illustration 23 Génération automatique de figures avec *Pspawn*

5. UN RESEAU ET UNE COLLECTION DE FIGURES

5.1. Réseau des figures

5.1.1. La figure est contenue par l'état de l'environnement

Il faut remarquer que nous ne considérons pas la figure comme occupant un niveau inférieur ou intermédiaire. De quelques millisecondes à la totalité d'une pièce, la figure n'occupe pas une échelle temporelle particulière[16]. Elle ne fait pas partie, si nous utilisons le langage plus traditionnel, d'une « phrase » ou une « section ». Pour les mêmes raisons, elle n'occupe pas la place dans une « *timeline* ». Autrement dit, la figure, en tant que séquence d'événements finie déclarée comme variable d'environnement, n'a pas de « conteneur » intermédiaire ou final. S'il y a quelque chose qui contient à une figure, c'est une autre figure, ou plus exactement, l'état en cours de l'environnement, avec le réseau de figures. Il est, hiérarchiquement, autonome.

5.1.2. Composition d'un réseau de figures

Comme nous l'avons noté plus haut, le cycle écriture-réécriture, tel que nous l'abordons, est ponctué par une situation dans laquelle nous percevons l'état de l'objet comme « celui que je voulais », autrement nous continuons à modifier les séquences. C'est grâce à la perception de ce double aspect que nous pouvons construire, par étapes successives, un réseau de figures indépendantes et interdépendantes.

```
(
q = Buffer.read(s,Platform.resourceDir +/+ "Sounds" +/+ "45-v04.wav");
w = Buffer.sendCollection(s, Signal.hanningWindow(1024));
)

(
~f_11 = Pdef (\f_11, Ptpar([ 0.0, ~f_14, 0.13, ~f_15], 1));

~f_12 = Pdef (\f_12, Ptpar ([(0.06.rand), ~f_13 = Pdef (\f_13, Pbind (\instrument,\grainbuf_01,\windowbuf, y,\buf,
b,\graindur, 1.5,\dur, Pwhite (0.1,0.16,2),\amp, Pseq ([0.04,0.1]) , \pan, -1)),(0.06.rand), Pbindf (~f_13,
\amp,Pseq ([0.1,0.05]), \pan, 1)), 1));

~f_14 = Pdef (\f_14, Pseq ([Pfin (3, Pbindf (~figure_02,\amp, Pexprand (0.002, 0.2))), 15));

~f_15 = Pdef (\f_15, Pbindf (~figure_09, \rate, 10, \amp, Prand ([0.04,0.35], [9, 1].normalizeSum, inf)));

~f_16 = Pdef (\f_16, Pbindf (~f_12, \rate, Pwhite (10, 10.2)));

~f_17 = Pdef (\f_17, Pfin (6, Pbindf (~figure_03)));

~f_18 = Pdef (\f_18,Pbind (\instrument, \grainbuf_01,\windowbuf, y,\buf, b,\graindur, 0.03, \dur, Pgeom (0.1, 0.96,
35) ,\amp, Pgeom (0.85, 0.96, 35),\pos, Pseq(Pseq ([0.0, 0.8], inf), 4.5, \lin),\rate, Pwhite (3.98, 4.02, inf)));

~f_19 = Pdef (\f_19, Ptpar ([(0.031.rand), Pbindf (~f_18, \pan, -1),(0.031.rand), Pbindf (~f_18, \pan, 1)], 1));

~f_20 = Pdef (\f_20,Pbindf (~f_19, \graindur, Pgeom (0.03, 1.1, 32)));

~f_21 = Pdef (\f_21, Ptpar ([(0.021.rand), ~f_21b = Pdef (\f_21b, Pbindf (~f_18, \dur, Pgeom (0.04, 0.92, 35),
\graindur, Pgeom (0.055, 0.85, 35),\pan, -1)), (0.021.rand), Pbindf (~f_21b, \pan, 1)], 1));

~f_22 = Pdef (\f_22, Ptpar ([0.0, ~f_23 = Pdef (\f_23, Pbind (\instrument,\grainbuf_01,\windowbuf, y,\buf, b,
\graindur, Pwhite (0.5,1.5),\dur, Pn (Pwhite (0.15, 0.18, 1), 1),\amp,Pwhite (0.07,0.3),\rate, Pwhite (0.99, 1.01,
inf),\pan, -1)),0.025, Pbindf (~f_23, \pan, 1)], 1));

~f_24 = Pdef (\f_24, Ptpar ([0.0, ~f_18, 0.02, ~f_19], 1));

~f_25 = Pdef (\f_25,Pbind (\instrument, \grainbuf_01,\windowbuf, y,\buf, b,\graindur, Pexprand (0.01,0.1), \dur,
Pexprand (0.04, 0.06),\amp, 0.7,\pos, Pseq(Pseq ([0.0, 0.8, 0.03, 0.2], inf), Prand ([0.1, 0.2, 0.13, 0.22], 8),
\lin),\rate, Pwhite (3.98, 4.02, inf)));
```

Illustration 24 Un extrait d'un réseau de figures que nous avons composé.

5.2. Collection des figures

Nous obtenons en même temps une collection de figures. Avec l'objet *PdefAllGui* de la extension

[16] Nous suivons ici la même notion de figure que Horacio Vaggione. [9]

JITLib[17] on peut y accéder. C'est une interface qui nous montre l'ensemble des *Pdef* et qui constitue le réseau que nous sommes en train de composer. Dans le sens inverse, nous pouvons jouer séparément une figure pour décider de la réintroduire (telle quelle ou avec variations) dans une autre partie de la composition.

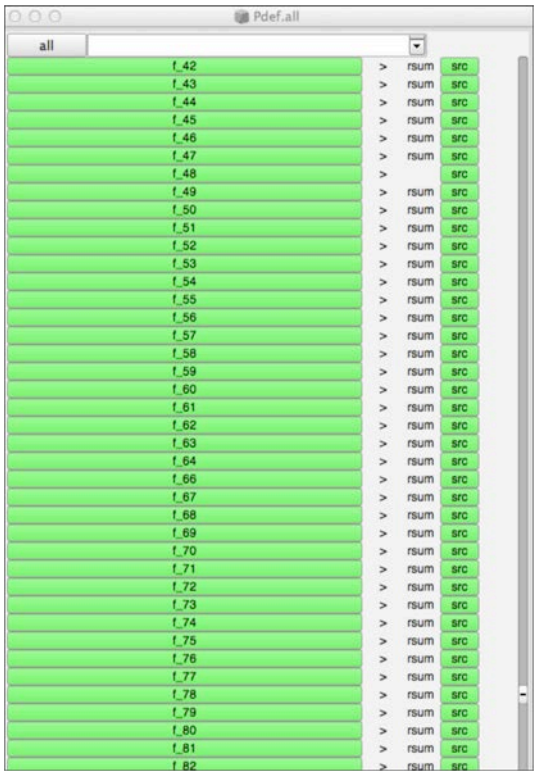

Illustration 25 Extrait d'une collection de figures

6. CONCLUSION

Dans ce texte nous avons visé la clarification d'un aspect du processus de la composition musicale tel que nous l'abordons en milieu informatique. Il apparaît nettement qu'une séquence d'évènements finie, plutôt qu'un flux continu est la condition essentielle avec laquelle nous pouvons commencer à penser la composition d'une forme singulière.

Comme le thème permettant des variations, ce que nous avons nommé « figure », est le point d'accroche pour toute opération postérieure. Cela dit, il faut être prudent avec l'analogie car la situation est toute autre. En faisant de « variations », nous arrivons à un autre « état satisfaisant » de l'objet, ce qui fait que la séquence soit à son tour une nouvelle figure, une nouvelle référence. Dans ce sens l'idée de « figure de référence », en tant que séquence d'événements finie, déclarée comme variable d'environnement, semble plus adapté à la composition visant une pluralité de niveaux spatio-temporels indépendants et interdépendants.

[17] http://doc.sccode.org/Overviews/JITLib.html

7. REFERENCES

[1] Lorenzo M. *Choix et composition musicale : dans l'espace des raisons*. Saint Denis : Paris VIII, 2014.

[2] Lorenzo M. « "Comme objet" et "comme figure" ». Non publié, mars 2016.

[3] Blackwell A., Collins N. « The Programming Language as a Musical Instrument ». In : Romero P (éd.), 2005.

[4] Wilson S., Collins N. *The SuperCollider book*. Cambridge, Mass. : MIT Press, 2011.

[5] Vaggione H. « Son, temps, objet, syntaxe. Vers une approche multi-échelle dans la composition assistée par ordinateur ». In : Soulez A (éd.). *Musique Ration. Lang. Harmon. Monde Au Matér.* Paris : L'Harmattan, 1998.

[6] Caires C. « IRIN: Micromontage in Graphical Sound Editing and Mixing Tool ». *Int. Comput. Music Conf. Proc.*, 2004.

[7] Carvalho G. « Formaliser la forme ». In : *Manières Faire Sons*. Paris : L'Harmattan, 2010.

[8] Vaggione H. « Décorrélation microtemporelle, morphologies et figurations spatiales ». In : *Actes Journ. Inform. Music. JIM 2002*, Marseille, 2002.

[9] Solomos M. (éd.). *Espaces composables : essais sur la musique et la pensée musicale d'Horacio Vaggione*. Paris : L'Harmattan, 2007.