Une introduction en douceur à SuperCollider

par Bruno Ruviaro

Publié pour la première fois en 2014. Cette révision date du 20 novembre 2015.

Contenu

Une introduction en douceur à SuperCollider

Bruno Ruviaro 20

novembre 2015

Première partie

BASE

1 Bonjour le monde

Prêt à créer votre premier programme SuperCollider ? En supposant que vous ayez SC sous les yeux, ouvrez un nouveau document (menu Fichier→Nouveau, ou raccourci [ctrl+N]) et tapez la ligne suivante :

```
1 "Hello World".postln ;
```

Laissez votre curseur n'importe où sur cette ligne (peu importe que ce soit au début, au milieu ou à la fin). Appuyez sur [ctrl+Entrée] pour évaluer le code. "Hello world" apparaît dans la fenêtre Post. Félicitations ! Vous avez créé votre premier programme SuperCollider.

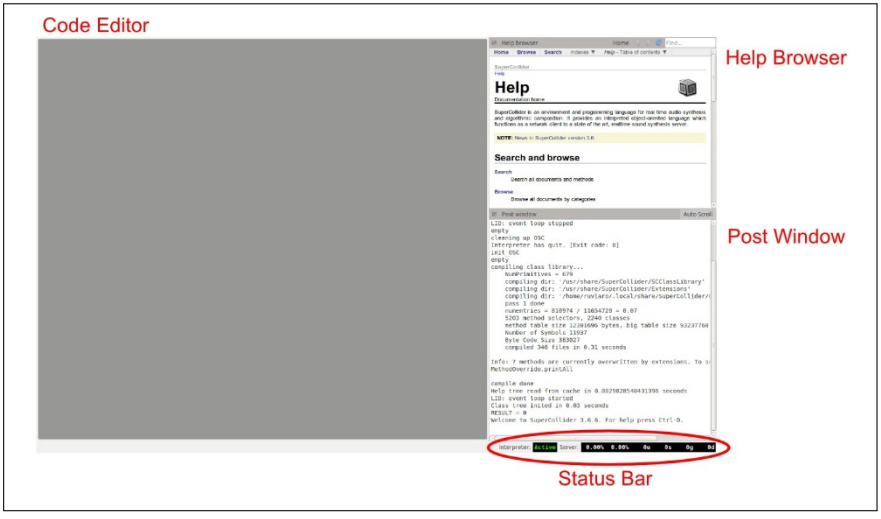

Figure 1 : Interface IDE du SuperCollider.

ASTUCE : Dans ce document, ctrl (control) indique la touche de modification des raccourcis clavier utilisée sur les plates-formes Linux et Windows. Sous Mac OSX, utilisez plutôt cmd (commande).

La figure 1 montre une capture d'écran de l'IDE (environnement de développement intégré) de SuperCollider lorsque vous l'ouvrez pour la première fois. Prenons le temps de le découvrir un peu.

Qu'est-ce que l'IDE SuperCollider ? C'est "un environnement de codage multiplateforme développé spécifiquement pour SuperCollider (. . .), facile à utiliser, pratique à manipuler, et doté de puissantes fonctionnalités pour les codeurs expérimentés. Il est également très personnalisable.

Il fonctionne aussi bien et a l'air presque

les mêmes sur Mac OSX, Linux et Windows."*

Les principales parties de la fenêtre SC sont l'éditeur de code, le navigateur d'aide et la fenêtre d'affichage. Si vous ne voyez aucune de ces parties lorsque vous ouvrez SuperCollider, allez simplement dans le menu

View→Docklets (c'est là que vous pouvez afficher ou masquer chacun d'entre eux). Il y a également la barre d'état, toujours située dans le coin inférieur droit de la fenêtre.

Gardez toujours la fenêtre Post visible, même si vous ne comprenez pas encore tout ce qui y est imprimé. La fenêtre Post affiche les réponses du programme à nos commandes : résultats de l'évaluation du code, notifications diverses, avertissements, erreurs, etc.

ASTUCE : Vous pouvez agrandir ou réduire temporairement la taille de la police de l'éditeur à l'aide des raccourcis [Ctrl++] et [Ctrl+-] (c'est-à-dire la touche Ctrl et les touches plus ou moins, respectivement). Si vous utilisez un ordinateur portable sans véritable touche plus, utilisez [Ctrl+shift+=].

2 Serveur et langue

Dans la barre d'état, vous pouvez voir les mots "Interprète" et "Serveur". L'interprète est activé par défaut ("Actif"), tandis que le "Serveur" est désactivé (c'est ce que signifient tous les zéros). Qu'est-ce que l'interprète et qu'est-ce que le serveur ?

SuperCollider est en fait composé de deux applications distinctes : le serveur et le langage. Le serveur est responsable de la production des sons. Le langage (également appelé *client* ou *interprète*) est utilisé pour contrôler le serveur. Le premier est appelé scsynth (SC-synthétiseur), le second sclang (SC-langage). La barre d'état nous indique l'état (activé/désactivé) de chacun de ces deux composants.

*Extrait de la documentation de SuperCollider : http://doc.sccode.org/Guides/SCIde.html. Visitez cette page pour en savoir plus sur l'interface IDE.

Ne vous inquiétez pas si cette distinction n'a pas beaucoup de sens pour vous maintenant. Les deux choses principales que vous devez savoir à ce stade sont les suivantes :

1. Tout ce que vous tapez dans SuperCollider se fait dans le langage SuperCollider (le client) : c'est là que vous écrivez et exécutez les commandes, et que vous voyez les résultats dans la fenêtre Post.

2. Tout ce qui produit un son dans SuperCollider provient du serveur - la "machine à sons", pour ainsi dire -, contrôlé par vous à travers le langage SuperCollider.

2.1 Démarrage du serveur

Votre programme "Hello World" n'a produit aucun son : tout s'est passé dans le langage et le serveur n'a pas été utilisé du tout. L'exemple suivant produira du son, nous devons donc nous assurer que le serveur est opérationnel.

La façon la plus simple de démarrer le serveur est d'utiliser le raccourci [ctrl+B]. Vous pouvez également cliquer sur les zéros de la barre d'état : un menu s'affiche et l'une des options est "Démarrer le serveur". Vous verrez une certaine activité dans la fenêtre Post au fur et à mesure que le serveur démarre. Lorsque vous aurez réussi à démarrer le serveur, les chiffres de la barre d'état deviendront verts. Vous devrez effectuer cette opération à chaque fois que vous lancerez SC, mais une seule fois par session.

3 Votre première onde sinusoïdale

"Hello World" est traditionnellement le premier programme que l'on crée lorsqu'on apprend un nouveau langage de programmation. Vous l'avez déjà fait dans SuperCollider.

La création d'une simple onde sinusoïdale pourrait être le "Hello World" des langages de musique assistée par ordinateur. Passons directement à l'action. Tapez et évaluez la ligne de code

suivante. Attention, cela peut être bruyant. Baissez le volume au maximum, évaluez la ligne, puis augmentez lentement le volume.

```
{SinOsc.ar}.play ;
```

C'est une belle onde sinusoïdale, douce, continue et peut-être un peu ennuyeuse. Vous pouvez arrêter le son avec [ctrl+] (c'est la touche *contrôle* plus la touche *point*). Mémorisez cette combinaison de touches, car vous l'utiliserez souvent pour arrêter n'importe quel son dans SC.

ASTUCE : Sur une ligne séparée, tapez et exécutez s.volume.gui si vous souhaitez disposer d'un curseur graphique pour contrôler le volume de la sortie du SuperCollider.

Rendons maintenant cette onde sinusoïdale un peu plus intéressante. Tapez ceci :

```
{SinOsc.ar(LFNoise0.kr(10).range(500, 1500), mul : 0.1)}.play ;
```

Rappelez-vous qu'il vous suffit de laisser votre curseur n'importe où dans la ligne et d'appuyer sur [ctrl+Entrée] pour l'évaluer. Vous pouvez également sélectionner la ligne entière avant de l'évaluer.

CONSEIL : Taper les exemples de code par vous-même est un excellent outil d'apprentissage. Cela vous aidera à gagner en confiance et à vous familiariser avec le langage. Lorsque vous lisez des tutoriels dans un forum numérique, vous pouvez être tenté de simplement copier et coller de courts extraits de code à partir des exemples. C'est très bien, mais vous apprendrez davantage si vous le tapez vous-même - essayez cela au moins dans les premières étapes de votre apprentissage du langage SC.

4 Messages d'erreur

Pas de son lorsque vous avez évalué le dernier exemple ? Si c'est le cas, votre code contenait probablement une faute de frappe : un caractère erroné, une virgule ou une parenthèse manquante,

etc. Lorsque quelque chose ne va pas dans votre code, la fonction

La fenêtre d'affichage affiche un message d'erreur. Les messages d'erreur peuvent être longs et énigmatiques, mais ne paniquez pas : avec le temps, vous apprendrez à les lire. Un message d'erreur court pourrait ressembler à ceci :

```
ERREUR : Class not
  defined. in file
  'selected text' line 1
  char 19 :

  {SinOsc.ar(LFNoiseO.kr(12).range(400, 1600), mul : 0.01)}.play ;

----------------------------------------
néant
```

Ce message d'erreur indique "Class not defined" (classe non définie) et pointe vers l'emplacement approximatif de l'erreur de saisie.

erreur ("line 1 char 19"). Les classes en SC sont ces mots bleus qui commencent par une majuscule (comme SinOsc et LFNoise0). Il s'avère que cette erreur est due au fait que l'utilisateur a tapé LFNoiseO avec un "O" majuscule à la fin. La classe correcte est LFNoise0, avec le chiffre zéro à la fin. Comme vous pouvez le constater, le souci du détail est crucial.

Si votre code contient une erreur, relisez-le, modifiez-le si nécessaire et réessayez jusqu'à ce qu'il soit corrigé. Si vous n'aviez pas d'erreur au départ, essayez d'en introduire une maintenant afin de voir à quoi ressemblerait le message d'erreur (par exemple, supprimez une virgule).

CONSEIL : Apprendre SuperCollider, c'est comme apprendre une autre langue comme l'allemand, le portugais ou le japonais. Il suffit d'essayer de la parler, d'enrichir son vocabulaire, de faire attention à la grammaire et à la syntaxe, et d'apprendre de ses erreurs. Le pire qui puisse arriver est de planter SuperCollider. Ce n'est pas aussi grave que de prendre le mauvais bus à São Paulo à cause d'une demande d'itinéraire mal prononcée.

5 Modification des paramètres

Voici un bel exemple adapté du premier chapitre du livre SuperCollider.* Comme pour les exemples précédents, ne vous inquiétez pas en essayant de tout comprendre. Appréciez simplement le résultat sonore et jouez avec les nombres.

```
1 {RLPF.ar(Dust.ar([12, 15]), LFNoise1.ar([0.3, 0.2]).range(100, 3000), 0.02)}.play ;
```

Arrêtez le son, changez certains chiffres et évaluez à nouveau. Par exemple, que se passe-t-il si vous remplacez les nombres 12 et 15 par des nombres inférieurs compris entre 1 et 5 ? Après LFNoise1, que se passerait-il si, au lieu de 0,3 et 0,2, vous essayiez quelque chose comme 1 et 2 ? Modifiez-les un par un. Comparez le nouveau son avec le précédent, écoutez les différences. Voyez si vous pouvez comprendre quel chiffre contrôle quoi. C'est une façon amusante d'explorer SuperCollider : prenez un bout de code qui produit quelque chose d'intéressant, et jouez avec les paramètres pour créer des variations. Même si vous ne comprenez pas parfaitement le rôle de chaque chiffre, vous pouvez toujours obtenir des résultats sonores intéressants.

CONSEIL : Comme pour tout logiciel, n'oubliez pas de sauvegarder fréquemment votre travail avec [ctrl+S] ! Lorsque vous travaillez sur des tutoriels comme celui-ci, vous trouverez souvent des sons intéressants en expérimentant avec les exemples fournis. Lorsque vous souhaitez conserver quelque chose qui vous plaît, copiez le code dans un nouveau document et sauvegardez-le. Remarquez que chaque fichier SuperCollider porte l'extension .scd, qui signifie "SuperCollider Document".

*Wilson, S., Cottle, D. et Collins, N. (éditeurs). The SuperCollider Book, MIT Press, 2011, p. 5. Plusieurs éléments du présent tutoriel ont été empruntés, adaptés ou inspirés par l'excellent "Beginner's Tutorial" de David Cottle, qui constitue le premier chapitre du livre. Ce tutoriel emprunte certains exemples et explications au chapitre de Cottle, mais - à la différence de celui-ci - suppose une moindre exposition à la musique assistée par ordinateur, et introduit la famille Pattern comme colonne vertébrale de l'approche pédagogique.

6 Commentaires

Tout texte dans votre code qui apparaît en rouge est un *commentaire*. Si vous êtes novice en matière de langages de programmation, les commentaires sont un moyen très utile de documenter votre code, à la fois pour vous et pour les autres personnes qui pourraient être amenées à le lire ultérieurement. Toute ligne commençant par une double barre oblique est un commentaire. Vous pouvez écrire des commentaires juste après une ligne de code valide ; la partie commentaire sera ignorée lors de l'évaluation. Dans SC, nous utilisons un point-virgule pour indiquer la fin d'une déclaration valide.

```
2 + 5 + 10 - 5 ; // je fais juste un peu de maths

rrand(10, 20) ; // génère un nombre aléatoire entre 10 et 20
```

Vous pouvez évaluer une ligne même si votre curseur se trouve au milieu du commentaire qui suit cette ligne. La partie commentaire est ignorée. Les deux paragraphes suivants seront écrits sous forme de "commentaires" pour les besoins de l'exemple.

```
// Vous pouvez rapidement commenter une ligne de code en utilisant le
raccourci [ctrl+/]. "Quelques lignes de code SC ici...".postln ;
2 + 2 ;

// Si vous écrivez un très long commentaire, votre texte peut s'interrompre pour
    former ce qui ressemble à une nouvelle ligne qui ne commence *pas* par une
    double barre oblique. Il s'agit toujours d'une seule ligne de commentaire.

/* Utilisez "barre oblique + astérisque" pour commencer un commentaire plus long de
    plusieurs lignes. Fermez le gros morceau de commentaire avec "astérisque +
    barre oblique". Le raccourci mentionné ci-dessus fonctionne également pour les
    gros morceaux : sélectionnez simplement les lignes de code que vous souhaitez
    commenter, et appuyez sur [ctrl+/]. Même chose pour annuler un commentaire. */
```

7 Priorité

SuperCollider suit un ordre de préséance de gauche à droite, quelle que soit l'opération. Cela signifie, par exemple, que la multiplication *n'est pas* effectuée en premier :

```
// Au lycée, le résultat était de 9 ; au CS, il est de 14 :
5 + 2 * 2 ;
// Utilisez des parenthèses pour imposer un ordre spécifique des opérations :
5 + (2 * 2) ; // égale 9.
```

Lors de la combinaison de messages et d'opérations binaires, les messages sont prioritaires. Par exemple, dans 5 + 2.au carré, la mise au carré a lieu en premier.

8 C'est toujours la dernière chose qui est postée

Un petit détail utile à comprendre : SuperCollider, par défaut, affiche toujours dans la fenêtre Post le résultat de la *dernière chose évaluée*. Cela explique pourquoi votre code Hello World s'imprime deux fois lorsque vous l'évaluez. Tapez les lignes suivantes dans un nouveau document, puis sélectionnez tout avec [ctrl+A] et évaluez toutes les lignes en même temps :

```
"Première
ligne".postln ;
"Deuxième
ligne".postln ; (2 +
2).postln ;
3 + 3 ;
"Fin".postln ;
```

Les cinq lignes sont exécutées par SuperCollider. Vous voyez le résultat de 2 + 2 dans la fenêtre Post parce qu'il y a eu une demande explicite de postln. Le résultat de 3 + 3 a été calculé, mais il n'y a pas eu de demande d'affichage, donc vous ne le voyez pas. Ensuite, la commande de la dernière ligne est exécutée (la commande

le mot "Finished" est affiché en raison de la demande postln). Enfin, le résultat de la toute dernière chose à évaluer est affiché par défaut : dans ce cas, il s'agit du mot "Finished".

9 Blocs de code

Sélectionner plusieurs lignes de code avant de les évaluer peut s'avérer fastidieux. Un moyen beaucoup plus simple d'exécuter un morceau de code en une seule fois est de créer un *bloc de code* : il suffit de mettre entre parenthèses toutes les lignes de code que l'on souhaite exécuter ensemble. Voici un exemple :

```
(
// Un petit poème
"Aujourd'hui, c'est
dimanche".postln ; "Pied de
tuyau".postln ;
"La pipe est en or".postln ; "Elle
peut battre le taureau".postln ;
)
```

Les parenthèses extérieures délimitent le bloc de code. Tant que votre curseur se trouve à l'intérieur des parenthèses, un simple [ctrl+Entrée] évalue toutes les lignes pour vous (elles sont exécutées dans l'ordre, de haut en bas, mais c'est tellement rapide que cela semble simultané).

L'utilisation de blocs de code vous évite de devoir sélectionner à nouveau toutes les lignes chaque fois que vous modifiez quelque chose et que vous souhaitez procéder à une nouvelle évaluation. Par exemple, modifiez certains mots entre guillemets et appuyez sur [ctrl+Entrée] juste après la modification. L'ensemble du bloc de code est évalué sans que vous ayez à sélectionner manuellement toutes les lignes. SuperCollider met le bloc en surbrillance pendant une seconde pour vous donner un indice visuel de ce qui est en cours d'exécution.

10 Comment nettoyer la fenêtre Post

Cette commande est tellement utile pour les adeptes du nettoyage qu'elle mérite une section à part entière : [ctrl+shift+P]. Évaluez cette ligne et profitez-en pour nettoyer la fenêtre Post :

```
100.do({"Imprimez cette ligne encore et encore...".scramble.postln}) ;
```

Il n'y a pas de quoi.

11 Enregistrement de la sortie de SuperCollider

Bientôt, vous voudrez commencer à enregistrer la sortie sonore de vos patchs SuperCollider. Voici une méthode rapide :

```
// ENREGISTREMENT RAPIDE
// Début de l'enregistrement :
s.record ;
// Produire des sons sympas
{Saw.ar(LFNoise0.kr([2, 3]).range(100, 2000), LFPulse.kr([4, 5]) * 0.1)}.play ;
// Arrêter l'enregistrement :
s.stopRecording ;
// Facultatif : Interface graphique avec bouton d'enregistrement, contrôle
du volume, bouton de sourdine : s.makeWindow ;
```

La fenêtre d'enregistrement indique le chemin d'accès au dossier dans lequel le fichier a été enregistré. Trouvez le fichier, ouvrez-le avec Audacity ou un programme similaire et vérifiez que le son a bien été enregistré. Pour plus d'informations, consultez le fichier d'aide "Server" (faites défiler la page jusqu'à "Recording Support"). Également en ligne à l'adresse http://doc.sccode.org/Classes/Server.html.

12 Variables

Vous pouvez stocker des nombres, des mots, des générateurs d'unités, des fonctions ou des blocs entiers de code dans des variables. Les variables peuvent être des lettres simples ou des mots entiers choisis par vous. Nous utilisons le signe égal (=) pour "attribuer" les variables. Exécutez ces lignes une par une et observez la fenêtre Post :

```
x = 10 ;
y = 660 ;
y ; // vérifier ce qu'il y a
là-dedans x ;
x + y
; y -
x ;
```

La première ligne attribue le nombre 10 à la variable x. La deuxième ligne place 660 dans la variable y. Les deux lignes suivantes prouvent que ces lettres "contiennent" maintenant ces nombres (les données). Enfin, les deux dernières lignes montrent que nous pouvons utiliser les variables pour effectuer n'importe quelle opération avec les données. Les lettres minuscules de a à z peuvent être utilisées à tout moment comme variables dans SuperCollider. La seule lettre simple que nous n'utilisons pas par convention est s, qui représente par défaut le serveur. N'importe quelle...

peut aller dans une variable :

```
a = "Hello, World" ; // une chaîne de caractères
b = [0, 1, 2, 3, 5] ; // une liste
c = Pbind(\note, Pwhite(0, 10), \dur, 0.1) ; // vous apprendrez tout sur Pbind plus
    tard, ne vous inquiétez pas

// ...et vous pouvez maintenant les utiliser comme vous le feriez avec les
données d'origine : a.postln ; // post it
b + 100 ; // fait des maths
c.play ; // joue ce Pbind
d = b * 5 ; // prendre b, le multiplier par 5 et l'affecter à une nouvelle variable
```

9

Il est souvent plus judicieux de donner des noms plus précis à vos variables, afin de vous aider à vous souvenir de leur signification dans votre code. Vous pouvez utiliser un ~ (tilde) pour déclarer une variable avec un nom plus long. Notez qu'il n'y a pas d'espace entre le tilde et le nom de la variable.

```
1 ~myFreqs = [415, 220, 440, 880, 220, 990] ;
2 ~myDurs = [0.1, 0.2, 0.2, 0.5, 0.2, 0.1] ;
3
4 Pbind(\freq, Pseq(~myFreqs), \dur, Pseq(~myDurs)).play ;
```

Les noms de variables doivent commencer par des lettres minuscules. Vous pouvez utiliser des chiffres, des traits de soulignement et des lettres majuscules dans le nom, mais pas comme premier caractère. Tous les caractères doivent être contigus (sans espace ni ponctuation). En bref, il faut s'en tenir aux lettres et aux chiffres et, à l'occasion, au trait de soulignement, et évitez tous les autres caractères lorsque vous nommez vos variables. ~myFreqs, ~theBestSineWave, et ~banana_3 sont des noms valables. ~MyFreqs, ~theBest&*#SineWave, et ~banana !!! sont de mauvais noms.

Il existe deux types de variables que vous pouvez créer : les variables "globales" et les variables locales.

12.1 "Global" et local

Les variables que vous avez vues jusqu'à présent (les lettres minuscules simples de a à z, et celles commençant par le caractère tilde (~)) peuvent être vaguement appelées "variables globales", car une fois déclarées, elles fonctionneront "globalement" n'importe où dans le patch, dans d'autres patchs, et même dans d'autres documents SC, jusqu'à ce que vous quittiez SuperCollider.*

*Techniquement parlant, les variables commençant par un tilde sont appelées variables d'environnement, et les variables en lettres minuscules (de a à z) sont appelées variables d'interprétation. Les débutants de SuperCollider n'ont pas besoin de se préoccuper de ces distinctions, mais gardez-les à l'esprit pour l'avenir. Le chapitre 5 du livre

SuperCollider explique ces différences en détail.

Les variables locales, quant à elles, sont déclarées avec le mot-clé réservé var en début de ligne. Vous pouvez attribuer une valeur initiale à une variable au moment de la déclaration (var apples = 4). Les variables locales n'existent que dans le cadre de ce bloc de code.

Voici un exemple simple comparant les deux types de variables. Évaluez ligne par ligne et observez la fenêtre Post.

```
// Variables d'environnement
~galaApples = 4 ;
~produits sanguins = 5 ;
~limes = 2 ;
~plantains = 1 ;

["Agrumes", ~oranges sanguines + ~limes]
; ["Non-agrumes", ~plantains +
~galaApples] ;

// Variables locales : valables uniquement à l'intérieur du bloc de code.
// Évaluer le bloc une fois et regarder la fenêtre
Post : (
var pommes = 4, oranges = 3, citrons = 8, bananes =
10 ; ["Agrumes", oranges + citrons].postln ; ["Non-
agrumes", bananes + pommes].postln ; "Fin".postln ;
)

~galaApples ; // existe
toujours apples ; // a
disparu
```

12.2 Réaffectation

Une dernière chose utile à comprendre à propos des variables est qu'elles peuvent être *réaffectées* : vous pouvez leur donner une nouvelle valeur à tout moment.

```
// Attribuer une
variable a = 10 + 3
;
a.postln ; // vérifier
a = 999 ; // réaffecter la variable (lui donner une nouvelle
valeur) a.postln ; // vérifier : l'ancienne valeur a
disparu.
```

Une pratique très courante et parfois déroutante pour les débutants consiste à *utiliser la variable elle-même dans sa propre réaffectation*. Prenons l'exemple suivant :

```
x = 10 ; // affecte 10 à la variable x
x = x + 1 ; // assigne x + 1 à la variable x
x.postln ; // le vérifie
```

La façon la plus simple de comprendre cette dernière ligne est de la lire comme suit : "prendre la valeur actuelle de la variable x, y ajouter 1, et affecter ce nouveau résultat à la variable x". Ce n'est vraiment pas compliqué, et vous verrez plus tard comment cela peut être utile.*

*Cet exemple démontre clairement que le signe égal, en programmation, n'est pas le même signe égal que celui que vous avez appris en mathématiques. En mathématiques, $x = x + 1$ est impossible (un nombre ne peut être égal à lui-même plus un). Dans un langage de programmation comme SuperCollider, le signe égal peut être vu comme une sorte d'action : *prendre le résultat de l'expression du côté droit du signe, et l'"assigner" à la variable du côté gauche*.

Partie II
MODÈLES

13 La famille Pattern

Essayons quelque chose de différent maintenant. Tapez et exécutez cette ligne de code :

```
1 Pbind(\degree, Pseries(0, 1, 30), \dur, 0.05).play ;
```

13.1 Rencontrer Pbind

Pbind est un membre de la famille Pattern dans SuperCollider. Le P majuscule de Pbind et Pseries signifie *Pattern* ; nous rencontrerons bientôt d'autres membres de la famille. Pour l'instant, examinons de plus près Pbind uniquement. Essayez cet exemple simplifié :

```
1 Pbind(\degree, 0).play ;
```

La seule chose que fait cette ligne de code dans la vie est de jouer la note *do moyen*, une fois par seconde. Le mot-clé \degree fait référence aux degrés de la gamme, et le nombre 0 signifie le premier degré de la gamme (une gamme de do majeur est supposée, donc c'est la note do elle-même). Notez que SuperCollider commence à compter les choses à partir de 0, et non à partir de 1. Dans une ligne simple comme celle ci-dessus, les notes C, D, E, F, G. . . seraient représentées par les nombres 0, 1, 2, 3, 4. . . Essayez de changer de numéro et remarquez comment la note change lorsque vous la réévaluez. Vous pouvez également choisir des notes en dessous du do moyen en utilisant des nombres négatifs (par exemple, -2 vous donnera la note A en dessous du do moyen). En bref, imaginez que le do central du piano est 0, puis comptez les touches blanches vers le haut ou vers le bas (nombres positifs ou négatifs) pour obtenir n'importe quelle autre note.

Maintenant, jouez un peu avec la durée des notes. Pbind utilise le mot-clé \dur pour spécifier les durées en secondes :

```
Pbind(\degree, 0, \dur, 0.5).play ;
```

Bien sûr, cela reste très rigide et inflexible - toujours la même note, toujours la même durée. Ne vous inquiétez pas : les choses s'amélioreront très bientôt.

13.2 Pseq

Jouons plusieurs notes en séquence, comme une gamme. Rendons également nos notes plus courtes, disons de 0,2 seconde.

```
Pbind(\degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 1), \dur, 0.2).play ;
```

Cette ligne présente un nouveau membre de la famille Pattern : Pseq. Comme son nom l'indique, ce motif traite des séquences. Tout ce dont Pseq a besoin pour jouer une séquence est :

- une liste d'éléments entre crochets
- un nombre de répétitions.

Dans l'exemple, la liste est [0, 1, 2, 3, 4, 5, 6, 7], et le nombre de répétitions est 1. Ce Pseq signifie simplement : "jouer une fois tous les éléments de la liste, dans l'ordre". Notez que ces deux éléments, la liste et le nombre de répétitions, se trouvent à l'intérieur des parenthèses de Pseq, et qu'ils sont séparés par une virgule.

Remarquez également l'apparition de Pseq dans le Pbind : il s'agit de la valeur d'entrée de \degree. Ceci est important : au lieu de fournir un seul nombre fixe pour le degré d'échelle (comme dans notre premier Pbind simple), *nous fournissons un Pseq entier : une recette pour une séquence de nombres*. En gardant cela à l'esprit, nous pouvons facilement développer cette idée et utiliser un autre Pseq pour contrôler également les durées :

```
Pbind(\degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 5), \dur, Pseq([0.2, 0.1, 0.1, 0.2,
    0,2, 0,35], inf)).play ;
```

Que se passe-t-il dans cet exemple ? Tout d'abord, nous avons modifié le nombre de répétitions du premier Pseq en le fixant à 5, de sorte que la gamme entière sera jouée cinq fois. Deuxièmement, nous avons remplacé la valeur \dur précédemment fixée à 0,2 par un autre Pseq. Ce nouveau Pseq comporte une liste de six éléments : [0.2, 0.1, 0.1, 0.2, 0.2, 0.35]. Ces nombres deviennent des valeurs de durée pour les notes résultantes. La valeur de répétition de ce deuxième Pseq est fixée à inf, ce qui signifie "infini". Cela signifie que le Pseq n'a aucune limite quant au nombre de fois qu'il peut répéter sa séquence. La séquence Pbind est-elle jouée indéfiniment ? Non : il s'arrête lorsque l'*autre* Pseq a terminé son travail, c'est-à-dire lorsque la séquence des degrés de la gamme a été jouée 5 fois.

Enfin, l'exemple comporte un total de huit notes différentes (la liste du premier Pseq), alors qu'il n'y a que six valeurs pour la durée (deuxième Pseq). Lorsque vous fournissez des séquences de tailles différentes comme celle-ci, Pbind les parcourt simplement en fonction des besoins.

Répondez à ces questions pour mettre en pratique ce que vous avez appris :

- Essayez d'utiliser le nombre 1 au lieu de inf comme argument de répétition du deuxième Pseq. Que se passe-t-il ?
- Comment faire en sorte que ce Pbind

fonctionne éternellement ? Les solutions se

trouvent à la fin du livre.[1]

13.3 Rendre votre code plus lisible

Vous avez peut-être remarqué que la ligne de code ci-dessus est assez longue. En fait, elle est si longue qu'elle s'étend sur une nouvelle ligne, bien qu'il s'agisse techniquement d'une seule

instruction. Les longues lignes de code peuvent être difficiles à lire. Pour éviter cela, il est courant de diviser le code en plusieurs lignes de code indentées.

L'objectif est de le rendre aussi clair et intelligible que possible. Le même Pbind ci-dessus peut être écrit comme suit :

```
(
Pbind(
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 5),
        \dur, Pseq([0.2, 0.1, 0.1, 0.2, 0.2, 0.35], inf)
) ;
)
```

A partir de maintenant, prenez l'habitude d'écrire vos Pbinds comme ceci. L'écriture d'un code bien ordonné et bien organisé vous aidera beaucoup dans l'apprentissage de SuperCollider.

Remarquez également que nous avons placé ce Pbind entre parenthèses pour créer un bloc de code (vous vous souvenez de la section 9 ?): comme il n'est plus sur une seule ligne, nous devons le faire pour pouvoir l'exécuter dans son ensemble. Assurez-vous simplement que le curseur se trouve n'importe où dans le bloc avant de l'évaluer.

13.4 Quatre façons de spécifier la hauteur

Pbind accepte d'autres façons de spécifier la hauteur, et pas seulement les degrés de l'échelle.

- Si vous souhaitez utiliser les douze notes chromatiques (touches noires et blanches du piano), vous pouvez utiliser \note au lieu de \ndegré. Le 0 correspondra toujours au do moyen, mais les pas incluront désormais les touches noires du piano (0 = do moyen, 1 = do*]*, 2 = ré, etc.).
- Si vous préférez utiliser la numérotation des notes MIDI, utilisez \midinote (60 = do moyen, 61 = do*]*, 62 = do, etc).
- Enfin, si vous préférez spécifier les fréquences directement en Herz,

utilisez \freq. Voir la figure 2 pour une comparaison des quatre

méthodes.
Dans l'exemple suivant, les quatre Pbinds jouent tous la même note : le la au-dessus du do moyen (A4).

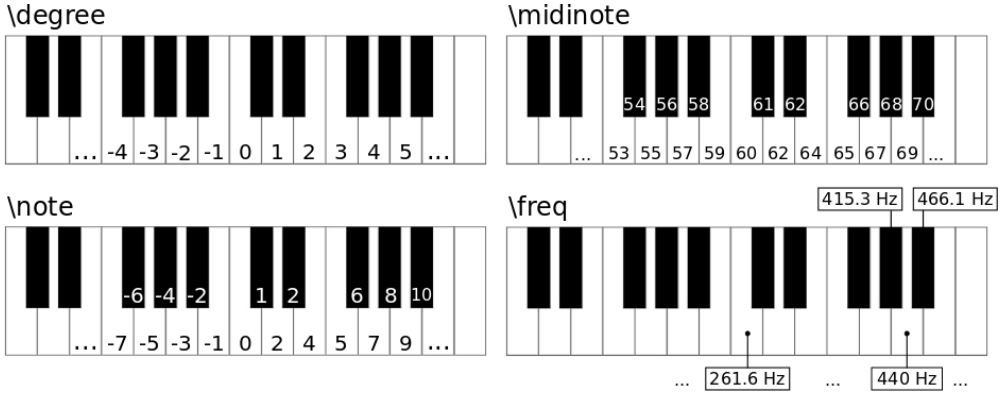

Figure 2 : Comparaison des degrés de la gamme, des nombres de notes, des notes moyennes et des fréquences

```
1 Pbind(\degree, 5).play ;
2 Pbind(\note, 9).play ;
3 Pbind(\midinote, 69).play ;
4 Pbind(\freq, 440).play ;
```

ASTUCE : Rappelez-vous que chaque type de spécification de hauteur attend des nombres dans une fourchette différente. Une liste de nombres comme [-1, 0, 1, 3] a du sens pour \degree et \note, mais n'en a pas pour \midinote et \freq. Le tableau ci-dessous compare quelques valeurs en utilisant le clavier du piano comme référence.

	A0 (note la plus basse du piano)	C4	A4	C5	C8 (note la plus haute du piano)
\degré de liberté	-23	0	5	7	21
\Note	-39	0	9	12	48
\n- note de bas de page	21	60	69	72	108
\freq	27.5	261.6	440	523.2	4186

13.5 Plus de mots-clés : amplitude et legato

L'exemple suivant introduit deux nouveaux mots-clés : \amp et \legato, qui définissent l'ampli- tude des événements et la quantité de legato entre les notes. Remarquez que le code est relativement facile à lire grâce à une bonne indentation et qu'il est réparti sur plusieurs lignes. Les parenthèses fermantes (en haut et en bas) sont utilisées pour délimiter un bloc de code afin d'en accélérer l'exécution.

```
(
Pbind(
        \degree, Pseq([0, -1, 2, -3, 4, -3, 7, 11, 4, 2, 0, -3], 5),
        \dur, Pseq([0.2, 0.1, 0.1], inf),
        \N- Pseq([0.7, 0.5, 0.3, 0.2], inf),
        \legato, 0.4
) ;
)
```

Pbind possède plusieurs de ces mots-clés prédéfinis, et avec le temps, vous en apprendrez davantage. Pour l'instant, nous allons nous en tenir à quelques-uns : un pour la hauteur

(choisissez parmi \degree, \note, \midinote, ou \freq), un pour la durée (\dur), un pour l'amplitude (\amp), et un pour le legato (\legato). Les durées sont exprimées en battements (dans ce cas, 1 battement par seconde, ce qui est la valeur par défaut) ; l'amplitude doit être comprise entre 0 et 1 (0 = silence, 1 = très fort) ; et le legato fonctionne mieux avec des valeurs comprises entre 0,1 et 1 (si vous

Si vous n'êtes pas sûr de ce que fait le legato, essayez simplement l'exemple ci-dessus avec 0,1, puis 0,2, puis 0,3, jusqu'à 1, et écoutez les résultats).

Prenez le dernier exemple comme point de départ et créez de nouveaux Pbinds. Changez la mélodie. Créez de nouvelles listes de durées et d'amplitudes. Expérimentez l'utilisation de \freq pour la hauteur. Re- member, vous pouvez toujours choisir d'utiliser un nombre fixe pour un paramètre donné, si c'est ce dont vous avez besoin. Par exemple, si vous voulez que toutes les notes de votre mélodie durent 0,2 seconde, il n'est pas nécessaire d'écrire Pseq[0,2, 0,2, 0,2, 0,2..., ni même Pseq([0,2], inf) - il suffit de supprimer toute la structure Pseq et d'écrire 0,2 à la place.

13.6 Prand

Prand est un proche cousin de Pseq. Il prend également en charge une liste et un certain nombre de répétitions. Mais au lieu de lire la liste dans l'ordre, Prand *choisit à chaque fois un élément aléatoire de la liste*. Essayez-le :

```
(
Pbind(
        \degree, Prand([2, 3, 4, 5, 6], inf),
        \dur, 0.15,
        \N-amp, 0.2,
        \legato, 0.1
) ;
)
```

Remplacez Prand par Pseq et comparez les résultats. Essayez maintenant d'utiliser Prand pour les durées, les amplitudes et le legato.

13.7 Pwhite

Un autre membre populaire de la famille Pattern est Pwhite. Il s'agit d'un générateur de nombres aléatoires à distribution égale (le nom vient de "bruit blanc"). Par exemple, Pwhite(100, 500) vous donnera des nombres aléatoires compris entre 100 et 500.

```
(
Pbind(
        \freq, Pwhite(100, 500),
        \dur, Prand([0.15, 0.25, 0.3], inf),
        \N-amp, 0.2,
        \legato, 0.3
).trace.play ;
)
```

L'exemple ci-dessus montre également une autre astuce utile : la trace de message juste avant le jeu. Il imprime dans la fenêtre Post les valeurs choisies pour chaque événement. Très utile pour le débogage ou simplement pour comprendre ce qui se passe !

Faites attention aux différences entre Pwhite et Prand : même si les deux ont un rapport avec le hasard, ils prennent des arguments différents et font des choses différentes. Dans les parenthèses de Pwhite, il suffit de fournir une limite basse et une limite haute : Pwhite(low, high). Des nombres aléatoires seront choisis à l'intérieur de cette fourchette. Prand, quant à lui, prend une liste d'éléments (nécessairement entre crochets) et un nombre de répétitions : Prand([liste, d'éléments], répétitions). Des éléments aléatoires seront choisis *dans la liste*.

Jouez avec les deux et assurez-vous de bien comprendre la différence.

ASTUCE : Un Pwhite avec deux nombres entiers ne générera que des nombres entiers. Par exemple, Pwhite(100, 500) produira des nombres comme 145, 568, 700, mais pas 145.6, 450.32, etc. Si vous souhaitez o b t e n i r des nombres à virgule flottante, écrivez Pwhite(100, 500.0). Ceci est très utile, par exemple, pour les amplitudes : si vous écrivez Pwhite(0, 1), vous n'obtiendrez que 0 ou 1, mais si vous écrivez Pwhite(0, 1.0), vous obtiendrez tout ce qui se trouve entre les deux.

Essayez les questions suivantes pour tester vos nouvelles connaissances :

a) Quelle est la différence de résultat entre Pwhite(0, 10) et Prand([0, 4, 1, 5, 9, 10, 2, 3], inf) ?

b) Si vous avez besoin d'un flux de nombres entiers choisis au hasard entre 0 et 100, pourriez-vous utiliser un Prand ?

c) Quelle est la différence de résultat entre Pwhite(0, 3) et Prand([0, 1, 2, 3], inf) ? Que se passe-t-il si vous écrivez Pwhite(0, 3.0) ?

d) Exécutez les exemples ci-dessous. Nous utilisons \note au lieu de \ndegré afin de jouer une gamme de C mineur (qui inclut les touches noires). La liste [0, 2, 3, 5, 7, 8, 11, 12] contient huit nombres, correspondant aux hauteurs C, D, *Eb*, F, G, *Ab*, B, C, mais combien d'événements chaque exemple joue-t-il réellement ? Mais combien d'événements chaque exemple joue-t-il réellement ? Pourquoi ?

```
// Pseq
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], 4),
        \dur, 0.15 ;
) ;
)
```

7

```
8
9  // Pseq
10 (
11 Pbind(
12         \note, Prand([0, 2, 3, 5, 7, 8, 11, 12], 4),
13         \dur, 0.15 ;
14 ) ;
15 )
16
17 // Pwhite
18 (
19 Pbind(
20         \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], 4),
21         \dur, Pwhite(0.15, 0.5) ;
22 ) ;
23 )
```

Les réponses se trouvent à la fin de ce tutoriel.[2]

ASTUCE : Un Pbind s'arrête de jouer lorsque le motif interne le plus court a fini d'être joué (tel que défini par l'argument repeats de chaque motif interne).

13.8 Élargir votre vocabulaire Pattern

Vous devriez maintenant être en mesure d'écrire des Pbinds simples par vous-même. Vous savez comment spécifier les hauteurs, les durées, les amplitudes et les valeurs de legato, et vous savez comment intégrer d'autres motifs (Pseq, Prand, Pwhite) pour générer des changements de paramètres intéressants.

Cette section vous permettra d'enrichir un peu votre vocabulaire sur les modèles. Les exemples ci-dessous présentent six autres membres de la famille Pattern. Essayez de comprendre par vous-même ce qu'ils font. Utilisez les stratégies suivantes :

- Écoutez la mélodie qui en résulte ; décrivez et analysez ce que vous entendez ;
- Examinez le nom du motif : suggère-t-il quelque chose ? (par exemple, Pshuf peut vous rappeler le mot "shuffle") ;
- Regardez les arguments (nombres) à l'intérieur du nouveau modèle ;
- Utilisez .trace.play comme indiqué précédemment pour observer les valeurs imprimées dans la fenêtre Post ;
- Enfin, confirmez vos suppositions en consultant les fichiers d'aide (sélectionnez le nom du motif et appuyez sur [ctrl+D] pour ouvrir le fichier d'aide correspondant).

```
// Élargir votre vocabulaire Pattern

// Pser
(
Pbind(
        \note, Pser([0, 2, 3, 5, 7, 8, 11, 12], 11),
        \dur, 0.15 ;
) ;
)

// Pxrand
// Comparez avec Prand pour voir la
différence (
p = Pbind(
    \note, Pxrand([0, 2, 3, 5, 7, 8, 11, 12], inf),
```

```
        \dur, 0.15 ;
) ;
)

// Pshuf
(
p = Pbind(
    \note, Pshuf([0, 2, 3, 5, 7, 8, 11, 12], 6),
        \dur, 0.15 ;
) ;
)

// Pslide
// Prend 4 arguments : list, repeats, length, step
(
Pbind(
        \note, Pslide([0, 2, 3, 5, 7, 8, 11, 12], 7, 3, 1),
        \dur, 0.15 ;
) ;
)

// Pseries
// Prend trois arguments : start, step, length
(
Pbind(
    \note, Pseries(0, 2, 15),
        \dur, 0.15 ;
) ;
)

// Pgeom
// Prend trois arguments : start, grow, length
```

```
(
Pbind(
        \note, Pseq([0, 2, 3, 5, 7, 8, 11, 12], inf),
        \dur, Pgeom(0.1, 1.1, 25) ;
) ;
)

// Pn
(
Pbind(
        \note, Pseq([0, Pn(2, 3), 3, Pn(5, 3), 7, Pn(8, 3), 11, 12], 1),
        \dur, 0.15 ;
) ;
)
```

Entraînez-vous à utiliser ces motifs - vous pouvez en faire beaucoup. Les Pbinds sont comme une recette pour une partition musicale, avec l'avantage que vous n'êtes pas limité à écrire des séquences fixes de notes et de rythmes : vous pouvez décrire des processus de paramètres musicaux en constante évolution (c'est ce qu'on appelle parfois la "composition algorithmique"). Et ce n'est là qu'un aspect des puissantes capacités de la famille Pattern.

À l'avenir, lorsque vous aurez besoin de plus d'objets de motifs, le meilleur endroit où aller est l'ouvrage de James Harkins "A Practical Guide to Patterns", disponible dans les fichiers d'aide intégrés.*

*Également en ligne à l'adresse suivante : http://doc.sccode.org/Tutorials/A-Practical-Guide/PG_01_Introduction.html

14 Plus d'astuces pour les patrons

14.1 Accords

Vous voulez écrire des accords dans Pbinds ? Écrivez-les sous forme de listes (valeurs séparées par des virgules et placées entre crochets) :

```
(
Pbind(
        \note, Pseq([[0, 3, 7], [2, 5, 8], [3, 7, 10], [5, 8, 12]], 3),
        \dur, 0.15
) ;
)
// Fun with strum
(
Pbind(
        \note, Pseq([[-7, 3, 7, 10], [0, 3, 5, 8]], 2),
        \dur, 1,
        \legato, 0.4,
        \strum, 0.1 // essayer 0, 0.1, 0.2, etc.
) ;
)
```

14.2 Balances

Lorsque vous utilisez \degree pour la spécification de votre pitch, vous pouvez ajouter une autre ligne avec le mot-clé
\scale pour changer d'échelle (note : ceci ne fonctionne qu'avec \degree, pas avec \note, \midinote, ou \freq) :

```
(
Pbind(
```

```
        \scale, Scale.harmonicMinor,
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], 1),
        \dur, 0.15 ;
) ;
)

// Évaluer cette ligne pour obtenir une liste de toutes les
échelles disponibles : Scale.directory ;

// Si vous avez besoin d'une note chromatique entre les degrés de
la gamme, faites ceci : (
Pbind(
        \degree, Pseq([0, 1, 2, 3, 3.1, 4], 1),
) ;
)

// Le 3.1 ci-dessus signifie un pas chromatique au-dessus du degré 3 de la gamme
    (dans ce cas, F# au-dessus de F). Notez que lorsque vous ne spécifiez pas
    explicitement une échelle, Scale.major est supposé.
```

14.3 Transposition

Utilisez le mot-clé \ctranspose pour obtenir une transposition chromatique. Cela fonctionne avec \degree, \note, et \midinote, mais pas avec \freq.

```
(
Pbind(
        \note, Pser([0, 2, 3, 5, 7, 8, 11, 12], 11),
        \ctranspose, 12, // transpose une octave au-dessus (= 12 demi-tons)
        \dur, 0.15 ;
) ;
```

```
)
```

14.4 Microtonalités

Comment écrire les microtons :

```
// Microtons avec \note et \midinote :
Pbind(\note, Pseq([0, 0.5, 1, 1.5, 1.75, 2], 1)).play ;
Pbind(\midinote, Pseq([60, 69, 68.5, 60.25, 70], 1)).play ;
```

14.5 Tempo

Les valeurs que vous attribuez à la clé \dur d'un Pbind sont exprimées en *nombre de battements, c'est-à-dire que* 1 signifie un battement, 0,5 signifie un demi-battement, et ainsi de suite. Sauf indication contraire, le tempo par défaut est de 60 BPM (battements par minute). Pour jouer à un tempo différent, il suffit de créer un nouveau TempoClock. Voici un Pbind jouant à 120 battements par minute (BPM) :

```
(
Pbind(\degree, Pseq([0, 0.1, 1, 2, 3, 4, 5, 6, 7]),
        \dur, 1 ;
).play(TempoClock(120/60)) ; // 120 battements sur 60 secondes : 120 BPM
)
```

Au fait, avez-vous vu que le Pseq ci-dessus ne prend qu'un seul argument (la liste) ? Où se trouve la valeur repeats qui vient toujours après la liste ? Vous pouvez entendre que l'exemple ne joue la séquence qu'une seule fois, mais pourquoi ? Il s'agit d'une propriété commune à tous les Patterns (et en fait, à de nombreux autres objets de SuperCollider) : si vous omettez un argument, il utilisera une valeur par défaut intégrée. Dans ce cas, la valeur par défaut de Pseq est 1.

Souvenez-vous de votre premier objet ridiculement simple, Pseq.

Pbind ? Il s'agissait d'un simple Pbind(\degree, 0).play et il ne savait jouer qu'une seule note. Vous n'avez pas fourni d'informations sur la durée, l'amplitude, le legato, etc. Dans ces cas-là, Pbind utilise simplement les valeurs par défaut.

14.6 Repose

Voici comment écrire les silences. Le nombre à l'intérieur du repos est la durée du repos en battements. Les silences peuvent être placés n'importe où dans le Pbind, pas seulement dans la ligne \dur.

```
(
Pbind(
        \degree, Pwhite(0, 10),
        \dur, Pseq([0.1, 0.1, 0.3, 0.6, Rest(0.3), 0.25], inf) ;
) ;
)
```

14.7 Jouer deux ou plusieurs Pbinds ensemble

Pour lancer plusieurs Pbinds simultanément, il suffit de les placer tous dans un seul bloc de code :

```
( // ouvrir un
grand bloc Pbind(
        \freq, Pn(Pseries(110, 111, 10)),
        \dur, 1/2,
        \legato, Pwhite(0.1, 1)
) ;

Pbind(
        \freq, Pn(Pseries(220, 222, 10)),
        \dur, 1/4,
```

```
        \legato, Pwhite(0.1, 1)
) ;

Pbind(
        \freq, Pn(Pseries(330, 333, 10)),
        \dur, 1/6,
        \legato, 0.1
) ;
) // fermer le grand bloc
```

Pour jouer les Pbinds de manière ordonnée dans le temps (autrement qu'en les évaluant manuellement l'un après l'autre), vous pouvez utiliser { }.fork :

```
// Exemple de fourche de base. Fenêtre
Watch Post : (
{
        "une chose".postln ;
        2.attendre ;
        "autre chose".postln ;
        1.5.wait ;
        "une dernière chose".postln ;
}.fork ;
)
// Un exemple plus intéressant :
(
t = TempoClock(76/60) ;
{
        Pbind(
                \note, Pseq([[4, 11], [6, 9]], 32),
                \dur, 1/6,
                \n-amp, Pseq([0.05, 0.03], inf)
        ).play(t) ;
```

```
20
21      2.attendre ;
22
23          Pbind(
24                  \note, Pseq([[-25, -13, -1], [-20, -8, 4], \rest], 3),
25                  \dur, Pseq([1, 1, Rest(1)], inf),
26                  \amp, 0.1,
27                  \legato, Pseq([0.4, 0.7, \rest], inf)
28
29
30
31
32
33                \n-ote, Pseq([23, 21, 25, 23, 21, 20, 18, 16, 20, 21, 23, 21], inf),
34                  \dur, Pseq([0.25, 0.75, 0.25, 1.75, 0.125, 0.125, 0.80, 0.20, 0.125,
                        0.125, 1], 1),
35                  \amp, 0.1,
36                  \legato, 0.5
37          ).play(t) ;
38  }.fork(t) ;
39  )
```

Pour des méthodes avancées permettant de jouer des Pbinds simultanément et en séquence, consultez Ppar et Pspawner. Pour en savoir plus sur la fourchette, consultez le fichier d'aide de la routine.

14.8 Utilisation de variables

Dans la section précédente "Élargir votre vocabulaire de motifs", avez-vous remarqué que vous deviez taper plusieurs fois la même liste de notes [0, 2, 3, 5, 7, 8, 11, 12] pour plusieurs Pbinds ? Ce n'est pas très efficace de copier la même chose à la main encore et encore, n'est-ce pas ? En programmation, chaque fois que vous vous retrouvez à faire la même tâche à plusieurs reprises, il

est probablement temps d'adopter une stratégie plus intelligente pour l'accomplir

le même objectif. Dans ce cas, nous pouvons utiliser des variables. Comme vous vous en souvenez peut-être, les variables vous permettent de faire référence à n'importe quel morceau de données d'une manière flexible et concise (revoir la section 12 si nécessaire). Voici un exemple :

```
// Vous utilisez souvent la même séquence de chiffres ? Enregistrez-
la dans une variable : c = [0, 2, 3, 5, 7, 8, 11, 12] ;

// Il suffit maintenant de s'y référer :
Pbind(\note, Pseq(c, 1), \dur, 0.15).play ;
Pbind(\note, Prand(c, 6), \dur, 0.15).play
;
Pbind(\note, Pslide(c, 5, 3, 1), \dur, 0.15).play ;
```

Un autre exemple pour s'entraîner à utiliser les variables : disons que nous voulons jouer deux Pbinds simultanément. L'un d'eux fait une gamme majeure ascendante, l'autre fait une gamme majeure descendante une octave au-dessus. Les deux utilisent la même liste de durées. Voici une façon d'écrire cela :

```
~échelle = [0, 1, 2, 3, 4, 5, 6, 7] ;
~durs = [0,4, 0,2, 0,2, 0,4, 0,8, 0,2, 0,2, 0,2]
; (
Pbind(
        \degree, Pseq(~scale),
        \dur, Pseq(~durs)
) ;

Pbind(
        \degree, Pseq(~scale.reverse + 7),
        \dur, Pseq(~durs)
) ;
)
```

Astuce intéressante : grâce aux variables, nous réutilisons la même liste de degrés et de durées de gammes pour les deux Pbinds. Nous voulions que la deuxième gamme soit descendante et une

octave au-dessus

le premier. Pour y parvenir, nous utilisons simplement le message .reverse pour inverser l'ordre de la liste (tapez ~scale.reverse sur une nouvelle ligne et évaluez pour voir exactement ce qu'il fait). Puis nous ajoutons 7 pour transposer d'une octave vers le haut (testez-le aussi pour voir le résultat).* Nous avons joué deux Pbinds à l'entrée de l'école.
en même temps en les enfermant dans un seul bloc de code.

Exercice : créez un Pbind supplémentaire à l'intérieur du bloc de code ci-dessus, de manière à entendre trois voix simultanées. Utilisez les deux variables (~scale et ~durs) d'une manière différente - par exemple, utilisez-les à l'intérieur d'un motif autre que Pseq ; changez le montant de la transposition ; inversez et/ou multipliez
durées ; etc.

15 Démarrage et arrêt indépendants de Pbinds

C'est une question très fréquente à propos des Pbinds, en particulier ceux qui s'éternisent avec l'inf : comment puis-je arrêter et démarrer des Pbinds individuels à volonté ? La réponse implique l'utilisation de variables, et nous verrons bientôt un exemple complet ; mais avant d'en arriver là, nous devons comprendre un peu mieux ce qui se passe lorsque vous jouez un Pbind.

15.1 Pbind comme une partition musicale

On peut considérer Pbind comme une sorte de partition musicale : c'est une recette pour produire des sons, un ensemble d'instructions pour réaliser un passage musical. Pour que la partition devienne de la musique, il faut la donner à un joueur : quelqu'un qui lira la partition et produira des sons sur la base de ces instructions. Séparons conceptuellement ces deux moments : la définition de la partition et son exécution.

```
1 // Définir le score
```

*Nous aurions également pu utiliser \ctranspose, 12 pour obtenir la même transposition.

```
(
p   = Pbind(
        \midinote, Pseq([57, 62, 64, 65, 67, 69], inf),
        \dur, 1/7
) ; // pas de .play ici !
)

// Demande de jouer la partition p.play
;
```

La variable p dans l'exemple ci-dessus contient simplement la partition - remarquez que le Pbind n'a pas de message .play juste après sa parenthèse de fermeture. Aucun son n'est produit à ce moment-là. Le deuxième moment est celui où vous demandez à SuperCollider de jouer à partir de cette partition : p.play.

Une erreur courante à ce stade est d'essayer p.stop, en espérant que cela arrêtera le lecteur. Essayez-le et vérifiez par vous-même que cela ne fonctionne pas de cette manière. Vous comprendrez pourquoi dans les prochains paragraphes.

15.2 Joueur de flux d'événements

Nettoyez la fenêtre Post avec [ctrl+shift+P] (ce n'est pas vraiment nécessaire, mais pourquoi pas ?) et évaluez à nouveau p.play. Regardez la fenêtre Post et vous verrez que le résultat est quelque chose appelé EventStreamPlayer. Chaque fois que vous appelez .play sur un Pbind, SuperCollider crée un lecteur pour réaliser cette action : c'est ce qu'est un EventStreamPlayer. C'est comme si un pianiste se matérialisait devant vous chaque fois que vous dites "Je veux que cette partition soit jouée maintenant". Sympa, non ?

Oui, sauf qu'une fois que ce joueur virtuel anonyme est arrivé et a commencé le travail, vous n'avez aucun moyen de lui parler - il n'a pas de nom. En termes un peu plus techniques, vous avez créé un objet, mais vous n'avez aucun moyen de vous référer à cet objet par la suite. À ce stade,

vous comprenez peut-être pourquoi p.stop n e fonctionne pas : c'est comme si vous essayiez de parler à la partition au lieu de parler à l'objet.

joueur. La partition (le Pbind stocké dans la variable p) ne sait rien du démarrage ou de l'arrêt : c'est juste une recette. C'est le *joueur* qui sait ce qu'est le démarrage, l'arrêt, "voulez-vous reprendre depuis le début", etc. En d'autres termes, vous devez parler au EventStreamPlayer. Tout ce que vous avez à faire, c'est de lui donner un nom, c'est-à-dire de le stocker dans une variable :

```
// Essayez ces lignes une par une :
~myPlayer = p.play ;
~myPlayer.stop ;
~myPlayer.resume ;
~myPlayer.stop.reset ;
~myPlayer.start ;
~myPlayer.stop ;
```

En résumé : appeler .play sur un Pbind génère un EventStreamPlayer ; et stocker vos EventStreamPlayers dans des variables vous permet d'y accéder plus tard pour démarrer et arrêter les motifs individuellement (pas besoin d'utiliser [ctrl+.], qui tue tout en même temps).

15.3 Exemple

Voici un exemple plus complexe pour conclure cette section. La mélodie supérieure est empruntée à l'Album pour la jeunesse de Tchaïkovski, et une mélodie inférieure est ajoutée en contrepoint. La figure 3 montre le passage en notation musicale.

```
// Définir le score
(
var myDurs = Pseq([Pn(1, 5), 3, Pn(1, 5), 3, Pn(1, 6), 1/2, 1/2, 1, 1, 3, 1, 3], inf
    ) * 0.4 ;
~upperMelody = Pbind(
        \midinote, Pseq([69, 74, 76, 77, 79, 81, Pseq([81, 79, 81, 82, 79, 81], 2),
            82, 81, 79, 77, 76, 74, 74], inf),
        \dur, myDurs
```

6

```
) ;
~lowerMelody = Pbind(
        \midinote, Pseq([57, 62, 61, 60, 59, 58, 57, 55, 53, 52, 50, 49, 50, 52, 50,
             55, 53, 52, 53, 55, 57, 58, 61, 62, 62], inf),
        \dur, myDurs
) ;
)
// Jouer les deux
ensemble : (
~player1 = ~upperMelody.play ;
~player2 = ~lowerMelody.play ;
)
// Arrêtez-les séparément :
~player1.stop ;
~player2.stop ;
// Autres messages disponibles
~player1.resume ;
~player1.reset ;
~player1.play ;
~player1.start ; // identique à .play
```

Tout d'abord, remarquez l'utilisation de variables. L'une d'entre elles, myDurs, est une variable locale. On peut dire qu'il s'agit d'une variable locale parce qu'elle ne commence pas par un tilde (~) et qu'elle est déclarée en haut avec le mot-clé réservé var. Cette variable contient un Pseq entier qui sera utilisé en tant que \dur dans les deux
le Pbinds. myDurs n'est vraiment nécessaire qu'au moment de la définition du score, il est donc logique d'utiliser une variable locale pour cela (bien qu'une variable d'environnement fonctionnerait très bien aussi). Les autres variables que vous voyez dans l'exemple sont des variables d'environnement - une fois déclarées, elles sont valables partout dans vos patchs SuperCollider.

Deuxièmement, remarquez la séparation entre le score et les joueurs, comme nous l'avons vu plus

haut. Lorsque le
Pbinds sont définis, ils ne sont pas joués immédiatement - il n'y a pas de .play
immédiatement après leur

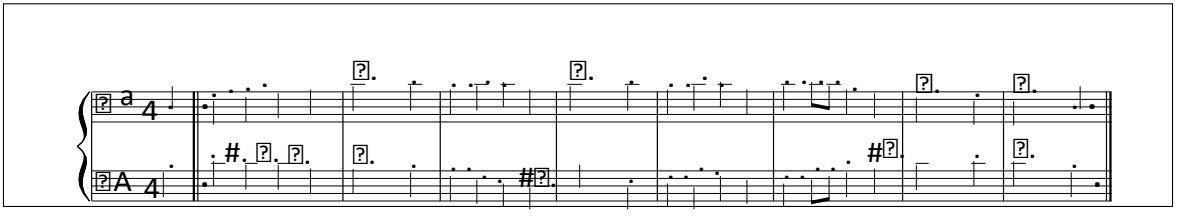

Figure 3 : Contrepoint Pbind avec une mélodie de Tchaïkovski

parenthèse fermante. Après avoir évalué le premier bloc de code, vous n'avez plus que deux définitions Pbind stockées dans les variables ~upperMelody et ~lowerMelody. Elles ne produisent pas encore de son - ce sont juste des partitions. La ligne ~player1 = ~upperMelody.play crée un fichier

EventStreamPlayer pour jouer la mélodie supérieure, et ce lecteur est appelé ~player1. Même chose pour ~player2. Grâce à cela, nous pouvons parler à chaque lecteur et lui demander de s'arrêter, de démarrer, de reprendre, etc.

Au risque d'être fastidieux, répétons-le une dernière fois :

- Un Pbind est simplement une recette pour produire des sons, comme une partition musicale ;
- Lorsque vous appelez le message play sur un Pbind, un objet EventStreamPlayer est créé ;
- Si vous stockez cet EventStreamPlayer dans une variable, vous pourrez y accéder ultérieurement pour utiliser des commandes telles que stop et resume.

Partie III
EN SAVOIR PLUS SUR LA LANGUE

16 Objets, classes, messages, arguments

SuperCollider est un langage de programmation orienté objet, comme Java ou C++. Il n'entre pas dans le cadre de ce tutoriel d'expliquer ce que cela signifie, nous vous laisserons donc chercher sur le web si vous êtes curieux. Nous nous contenterons ici d'expliquer quelques concepts de base que vous devez connaître pour mieux comprendre ce nouveau langage que vous êtes en train d'apprendre.

Dans SuperCollider, tout est un *objet*. Même les simples nombres sont des objets dans SC. Les objets se comportent différemment et contiennent différents types d'informations. Vous pouvez demander une information ou une action à un objet en lui envoyant un *message*. Lorsque vous écrivez quelque chose comme 2.carré, le message carré est envoyé à l'objet récepteur 2. Le point qui les sépare fait le lien. Les messages sont également appelés *méthodes*.

Les objets sont spécifiés hiérarchiquement dans des *classes*. SuperCollider est livré avec une vaste collection de classes prédéfinies, chacune avec son propre ensemble de méthodes.

Voici une bonne façon de le comprendre. Imaginons qu'il existe une classe abstraite d'objets appelée Animal. La classe Animal définit quelques méthodes générales (messages) communes à tous les animaux. Des méthodes comme âge, poids, photo peuvent être utilisées pour obtenir des informations sur l'animal. Des méthodes telles que bouger, manger, dormir permettent à l'animal d'effectuer une action spécifique. Nous pourrions alors avoir deux sous-classes d'Animal : l'une appelée Animal de compagnie, l'autre appelée Animal sauvage. Chacune de ces sous-classes pourrait avoir encore plus de sous-classes dérivées (comme Dog et Cat dérivées de Pet). Les sous-classes héritent de toutes les méthodes de leurs classes parentes et implémentent de nouvelles méthodes qui leur sont propres pour ajouter des

fonctionnalités spécialisées. Par exemple, les objets Chien et Chat répondraient volontiers au message .eat, hérité de la classe Animal. Les méthodes Dog.name et Cat.name renvoient le nom de l'animal.

l'animal de compagnie : cette méthode est commune à tous les objets dérivés de Pet. Le chien a une méthode bark, vous pouvez donc appeler Dog.bark et il saura quoi faire. Cat.bark vous enverrait un message d'erreur : ERROR : Message 'bark' non compris.

Dans tous ces exemples hypothétiques, les mots commençant par une majuscule sont des *classes* qui représentent des *objets*. Les mots en minuscules après le point sont des *messages* (ou des *méthodes*) envoyés à ces objets. L'envoi d'un message à un objet renvoie toujours une information. Enfin, les messages acceptent parfois (ou même exigent) des *arguments*. Les arguments sont les éléments qui se trouvent entre parenthèses juste après un message. Dans Cat.eat("sardines", 2), le message eat est envoyé à Cat avec des informations très spécifiques : ce qu'il faut manger et la quantité. Parfois, vous verrez des arguments déclarés explicitement à l'intérieur des parenthèses (mots-clés se terminant par deux points). C'est souvent pratique pour rappeler au lecteur à quoi l'argument fait référence. Dog.bark(volume : 10) est plus explicite que Dog.bark(10).

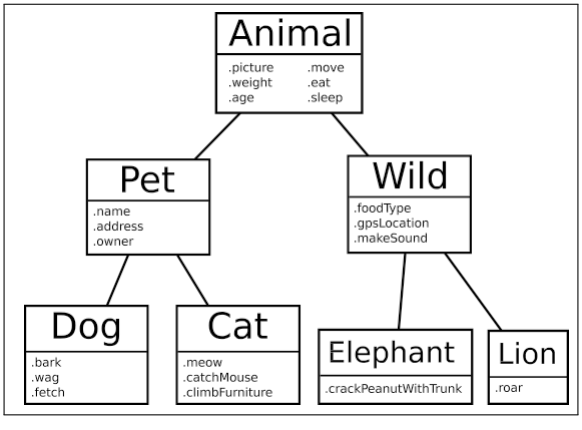

Figure 4 : hiérarchie hypothétique des classes.

OK - Assez de cette explication rapide et sale de la programmation orientée objet. Essayons quelques exemples que vous pouvez réellement exécuter dans SuperCollider. Lancez une ligne après l'autre et voyez si vous pouvez identifier le message, l'objet récepteur et les arguments (s'il y en a). La structure de base est Receiver.message(arguments) Réponses à la fin de ce document.[3]

```
[1, 2, 3, "wow"].reverse ;
"hello".dup(4) ;
3.1415.round(0.1) ; // notez que le premier point est le cas décimal de
3.1415 100.rand ; // évaluez cette ligne plusieurs fois
// L'enchaînement des messages est
amusant :
100.0.rand.round(0.01).dup(4) ;
```

17 Notation du récepteur, notation fonctionnelle

Il existe plusieurs façons d'écrire vos expressions dans SuperCollider. Celle que nous venons de voir ci-dessus s'appelle la *notation du récepteur* : 100.rand, où le point relie l'objet (100) au message (rand). La même chose peut également être écrite comme ceci : rand(100). C'est ce qu'on appelle la *notation fonctionnelle*.

Vous pouvez utiliser l'une ou l'autre façon d'écrire. Voici comment cela fonctionne lorsqu'un message prend deux arguments ou plus.

```
5.dup(20) ; // notation du récepteur
dup(5, 20) ; // même chose en notation fonctionnelle

3.1415.round(0.1) ; // notation du récepteur
round(3.1415, 0.1) ; // notation
fonctionnelle
```

Dans les exemples ci-dessus, vous pourriez lire dup(5, 20) comme "dupliquer le nombre 5 vingt fois", et round(3.1415, 0.1) comme "arrondir le nombre 3.1415 à une décimale". À

l'inverse, la fonction

Les versions de notation des récepteurs pourraient être lues comme "Numéro 5, duplique-toi vingt fois !". (pour 5.dup(20)) et "Nombre 3,1415, arrondis à une décimale !" (pour 3,1415.round(0,1)). (pour 3.1415.round(0.1)).

En bref : Receiver.message(argument) est équivalent à message(Receiver, argument). Le choix d'un style d'écriture par rapport à un autre est une question de préférence personnelle et de convention.

Parfois, une méthode est plus claire que l'autre. Quel que soit le style que vous préférez (et vous pouvez les mélanger), l'important est d'être cohérent. Une convention très répandue parmi les utilisateurs de SuperCollider est que les classes (mots commençant par des lettres majuscules) sont presque toujours écrites sous la forme Receiver.message(argument). Par exemple, vous verrez toujours SinOsc.ar(440), mais vous ne verrez presque jamais ar(SinOsc, 440), même si les deux sont corrects.

Exercice : réécrivez l'énoncé suivant en utilisant uniquement la notation fonctionnelle :
100,0.rand.round(0,01).dup(4) ;
Solution à la fin.[4]

18 L'emboîtement

La solution du dernier exercice vous a amené à imbriquer des choses les unes dans les autres. David Cottle a une excellente explication de l'imbrication dans le livre SuperCollider, nous nous contenterons donc de la citer ici.*

> *Pour clarifier davantage l'idée d'imbrication, considérons un exemple hypothétique dans lequel SC vous prépare un déjeuner. Pour ce faire, vous pourriez utiliser un* message de service. Les *arguments pourraient être la salade, le plat principal et le dessert. Mais le simple fait de dire* serve(lettuce, fish, banana) *peut ne pas donner les résultats escomptés. Pour plus de sécurité, vous pouvez donc*

clarifier ces arguments, en remplaçant chacun d'entre eux par un message et un argument imbriqués.

servir(mélanger(laitue, tomate, fromage), cuire(poisson, 400, 20), mélanger(banane, glace))

*Cottle, D. "Tutoriel du débutant". The SuperCollider Book, MIT Press, 2011, pp. 8-9.

SC servirait alors non seulement de la laitue, du poisson et une banane, mais aussi une salade composée avec de la laitue, de la tomate et du fromage, un poisson cuit au four et un sundae à la banane. Ces commandes internes peuvent être clarifiées en imbriquant un message (arg) pour chaque ingrédient : laitue, tomate, fromage, et ainsi de suite. Chaque message interne produit un résultat qui est à son tour utilisé comme argument par le message externe.

```
// Pseudo-code pour préparer
le dîner : serve(
        lancer
        (
                wash(laitue, eau, 10),
                couper la tomate en petits dés,
                saupoudrer(choisir([bleu, feta,
                gouda]))
        ),
        bake(catch(lagoon, hook, bamboo), 400, 20),
        mélanger(
                slice(peel(banana), 20),
                cook(mix(lait, sucre, amidon), 200, 10)
        )
) ;
```

Lorsque l'imbrication comporte plusieurs niveaux, nous pouvons utiliser de nouvelles lignes et des retraits pour plus de clarté. Certains messages et arguments sont laissés sur une seule ligne, d'autres sont répartis avec un argument par ligne - selon ce qui est le plus clair. Chaque niveau d'indentation doit indiquer un niveau d'imbrication. (Notez que vous pouvez avoir n'importe quelle quantité d'espace blanc - nouvelles lignes, tabulations ou espaces - entre les morceaux de code).

[Dans l'exemple du dîner, le *programme de déjeuner doit maintenant laver la laitue dans de l'eau pendant 10 minutes et couper la tomate en petits morceaux avant de les jeter dans le saladier et de les saupoudrer de fromage. Vous avez également*

précisé où pêcher le poisson et le faire cuire à 400 degrés pendant 20 minutes avant de le servir, et ainsi de suite.

Pour "lire" ce type de code, il faut commencer par le message le plus interne imbriqué et passer à chaque couche successive. Voici un exemple aligné pour montrer comment le message le plus proche est imbriqué dans les messages extérieurs.

```
                exprand(1.0, 1000.0) ;
           dup({exprand(1.0, 1000.0)}, 100) ;
      sort(dup({exprand(1.0, 1000.0)}, 100)) ;
round(sort(dup({exprand(1.0, 1000.0)}, 100)), 0.01) ;
```

Le code ci-dessous est un autre exemple d'imbrication. Répondez aux questions qui suivent. Vous n'avez pas besoin d'expliquer ce que font les nombres - la tâche consiste simplement à identifier les arguments dans chaque couche d'imbrication. (L'exemple et les questions de l'exercice sont également empruntés et légèrement adaptés au tutoriel de Cottle).

```
// Imbrication et indentation correcte
(
{
        CombN.ar(
                SinOsc.ar(
                        midicps(
                                LFNoise1.ar(3, 24,
                                        LFSaw.ar([5, 5.123], 0, 3, 80)
                                )
                        ),
                        0, 0.4
                ),
                1, 0.3, 2)
}.play ;
)
```

a) Quel est le deuxième argument de LFNoise1.ar ?

b) Quel est le premier argument de LFSaw.ar ?

c) Quel est le troisième argument de LFNoise1.ar ?

d) Combien d'arguments y a-t-il dans midicps ?

e) Quel est le troisième argument de SinOsc.ar ?

f) Quels sont les deuxième et troisième arguments de CombN.ar ? Les réponses se trouvent à la fin de ce document.[5]

ASTUCE : Si, pour une raison quelconque, votre code n'est plus correctement indenté, il vous suffit de le sélectionner entièrement et d'aller dans le menu Édition→Autoindenter la ligne ou la région, et il sera corrigé.

19 Enceintes

Il existe quatre types d'encadrements : (parenthèses), [crochets], {attaches} et "guillemets".

Chacun de ceux que vous ouvrez devra être fermé ultérieurement. C'est ce qu'on appelle "l'équilibrage", c'est-à-dire le maintien de paires d'enceintes correctement appariées dans l'ensemble du code.

L'IDE SuperCollider indique automatiquement les parenthèses correspondantes (ainsi que les crochets et les accolades) lorsque vous fermez une paire - elles apparaissent en rouge. Si vous

cliquez sur une parenthèse qui n'a pas de correspondance entre l'ouverture et la fermeture, vous verrez une sélection rouge foncé qui vous indiquera qu'il manque quelque chose.

L'équilibrage est un moyen rapide de sélectionner de grandes sections de code pour les évaluer, les supprimer ou les copier/coller. Vous pouvez double-cliquer sur une parenthèse ouvrante ou fermante (ainsi que sur les crochets et les accolades) pour sélectionner tout ce qui s'y trouve.

19.1 Guillemets

Les guillemets sont utilisés pour entourer une séquence de caractères (y compris les espaces) en tant qu'unité unique. C'est ce qu'on appelle les chaînes de caractères. Les guillemets simples créent des symboles, qui sont légèrement différents des chaînes de caractères. Les symboles peuvent également être créés à l'aide d'une barre oblique inverse placée immédiatement avant le texte. Ainsi, 'greatSymbol' et \greatSymbol sont équivalents.

```
1 "Voici une belle chaîne" ;
2 "greatSymbol" ;
```

19.2 Parenthèses

Les parenthèses peuvent être utilisées pour :

- entourer les listes d'arguments : rrand(0, 10) ;
- la préséance de la force : 5 + (10 * 4) ;
- créer des blocs de code (plusieurs lignes de code à évaluer ensemble).

19.3 Supports

Les crochets définissent une collection d'éléments, comme [1, 2, 3, 4, "hello"]. Ces éléments sont généralement appelés "tableaux". Un tableau peut contenir n'importe quoi : des nombres, des

chaînes de caractères, des fonctions, des motifs, etc. Tableaux

comprendre des messages tels que inverser, brouiller, refléter, choisir, pour n'en citer que quelques-uns. Vous pouvez également effectuer des opérations mathématiques sur les tableaux.

```
[1, 2, 3, 4, "hello"].scramble ;
[1, 2, 3, 4, "hello"].mirror ;
[1, 2, 3, 4].verso + 10 ;
// convertir midi en fréquence en Hz
[60, 62, 64, 65, 67, 69, 71].midicps.round(0.1) ;
```

Plus d'informations sur les tableaux dans la section 22.

19.4 Les bretelles frisées

Les accolades (ou "accolades bouclées") définissent les fonctions. Les fonctions encapsulent une sorte d'opération ou de tâche qui sera probablement utilisée et réutilisée plusieurs fois, avec éventuellement des résultats différents à chaque fois. L'exemple ci-dessous est tiré du livre SuperCollider.

```
exprand(1, 1000.0) ;
{exprand(1, 1000.0)} ;
```

David Cottle nous présente son exemple : "*la première ligne choisit un nombre aléatoire, qui s'affiche dans la fenêtre d'affichage. La seconde imprime un résultat très différent : une fonction. Que fait cette fonction ? Elle choisit un nombre aléatoire. Comment cette différence peut-elle affecter le code ? Examinez les lignes ci-dessous. La première choisit un nombre aléatoire et le duplique. La seconde exécute la fonction de sélection d'un nombre aléatoire 5 fois et collecte les résultats dans un tableau.*"*

```
rand(1000.0).dup(5) ; // choisit un nombre et le duplique
{rand(1000.0)}.dup(5) ; // duplique la fonction de sélection d'un nombre
{rand(1000.0)}.dup(5).round(0.1) ; // tout ce qui précède, puis arrondir
```

*Cottle, D. "Beginner's Tutorial". The SuperCollider Book, MIT Press, 2011, p. 13.

```
// essentiellement, ceci (qui a un résultat similaire)
[rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0), rand(1000.0)]
```

Nous reviendrons bientôt sur les fonctions. Pour l'instant, voici un résumé de tous les boîtiers possibles :

Collections [liste, des, éléments]

Fonctions { souvent plusieurs lignes de code }

Chaînes "mots entre guillemets"

Symboles "guillemets simples" ou précédés d'une barre oblique inversée

20 Conditionnelles : if/else et case

S'il pleut, je prendrai un parapluie pour sortir. S'il fait beau, je prendrai mes lunettes de soleil. Nos journées sont remplies de ce type de prise de décision. En programmation, il s'agit des moments où votre code doit tester une condition et prendre différentes mesures en fonction du résultat du test (vrai ou faux). Il existe de nombreux types de structures conditionnelles. Examinons-en deux simples : *if/else* et *case*.

La syntaxe d'un if/else dans SC est la suivante : if(condition, {action vraie}, {action fausse}). La condition est un test booléen (elle doit retourner vrai ou faux). Si le test renvoie vrai, la première fonction est évaluée ; sinon, c'est la deuxième fonction qui l'est. Essayez-le :

```
// if / else
if(100 > 50, { "très vrai".postln }, { "très faux".postln }) ;
```

Le tableau ci-dessous, emprunté au SuperCollider book*, présente quelques opérateurs booléens courants que vous pouvez utiliser. Notez la distinction entre un seul signe égal (x = 10) et deux signes égaux (x == 10). Le signe simple signifie "*assigner 10 à la variable x*", tandis que le signe double signifie "*x est-il égal à 10 ?* "Tapez et exécutez certains des exemples des colonnes vrai ou faux, et vous verrez les résultats vrai ou faux dans la fenêtre d'affichage.

Symbole	**Signification**	**Exemple vrai**	**Faux exemple**
==	égal à ?	10 == 10	10 == 99
!=	n'est pas égal à ?	10 != 99	10 != 10
>	plus grand que ?	10 > 5	10 > 99
<	moins que ?	10 < 99	10 < 5
>=	supérieur ou égal à ?	10 >= 10, 10 >= 3	10 >= 99
<=	inférieur ou égal à ?	10 <= 99, 10 <= 10	10 <= 9
impair	est-ce étrange ?	15.impairs	16.impair
même	Est-ce que c'est encore le cas ?	22,même	21,même
isInteger	s'agit-il d'un nombre entier ?	3.isInteger	3.1415.isInteger
isFloat	s'agit-il d'un flotteur ?	3.1415.isFloat	3.isFloat
et	les deux conditions	11.impair.et(12.pair)	11.impair.et(13.pair)
ou	l'une ou l'autre condition	ou(1.odd, 1.even)	ou(2.odd, 1.even)

Les deux dernières lignes (et, ou) indiquent comment écrire les expressions plus longues en notation réceptrice ou en notation fonctionnelle.

Une autre structure utile est le cas. Elle fonctionne en définissant des paires de fonctions à évaluer dans l'ordre jusqu'à ce que l'un des tests renvoie un résultat positif :

cas

*Cottle, D. "Beginner's Tutorial". The SuperCollider Book, MIT Press, 2011, p. 33.

{test1} {action1}
{test2} {action2}
{test3} {action3}
...
{testN} {actionN} ;

L'expression contenue dans chaque test doit renvoyer soit vrai, soit faux. Si le test 1 renvoie faux, le programme ignore l'action 1 et passe au test 2. S'il est faux, l'action 2 est également ignorée et nous passons au test 3. S'il s'avère que l'expression est vraie, l'action 3 est exécutée et le cas s'arrête (aucun autre test ou action n'est exécuté). Notez qu'il n'y a pas de virgule entre les fonctions. Il suffit d'utiliser un point-virgule à la toute fin pour marquer la fin de l'instruction case.

```
1  // case
2  (
3  ~num = -2 ;
4
5  cas
6  {~num == 0} {"WOW".postln}
7  {~num == 1} {"ONE !".postln}
8  {~num < 0} {"nombre négatif !".postln}
9  {true} {"last case scenario".postln} ;
10 )
```

Essayez de modifier le code ci-dessus pour obtenir tous les résultats possibles. Remarquez l'astuce utile (et facultative) de la dernière ligne de cas dans l'exemple ci-dessus : puisque true évalue toujours true, vous pouvez définir une action "dernier cas" qui se produira toujours si toutes les conditions précédentes sont fausses.

Pour en savoir plus, consultez le fichier d'aide sur les structures de contrôle.

21 Fonctions

Lorsque vous effectuez plusieurs fois la même tâche, il peut être judicieux de créer une fonction réutilisable. Une fonction, comme vous l'avez appris dans la section Enclosures, est quelque chose qui s'écrit à l'intérieur d'accolades. David Touretzky présente l'idée de fonction de la manière suivante : "Pensez à une fonction comme à une boîte à travers laquelle les données circulent. La fonction opère sur les données d'une manière ou d'une autre, et le résultat est ce qui s'écoule." *

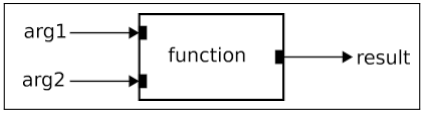

Figure 5 : Idée générale d'une fonction.

La première ligne de l'exemple ci-dessous définit une fonction et l'affecte à la variable f. La deuxième ligne met en œuvre la fonction.

```
1 f = { 2 + 2 } ; // définit la fonction
2 f.value ; // fait fonctionner la
fonction
```

La fonction ci-dessus n'est pas très utile, car elle ne sait faire qu'une seule chose (ajouter 2 et 2). Normalement, vous voudrez définir des fonctions qui peuvent donner des résultats différents en fonction des arguments d'entrée que vous lui donnez. Nous utilisons le mot-clé arg pour spécifier les entrées qu'une fonction peut accepter. L'exemple ci-dessous ressemble davantage au dessin de la figure 5.

```
1 f = {arg a, b ; ["a plus b", a+b, "a fois b", a*b].postln} ; // définir la fonction
2 f.value(3, 7) ; // maintenant vous pouvez donner deux nombres quelconques comme
arguments à la fonction
```

*Touretzky, David. COMMON LISP : A Gentle Introduction to Symbolic Computation. The Benjamin/Cum-

mings Publishing Company, Inc, 1990, p. 1. C'est le livre qui a inspiré le titre de ce tutoriel.

```
f.value(10, 14) ;

// Comparez :
~sillyRand = rrand(0, 10) ; // n'est pas une fonction
~sillyRand.value ; // évaluer plusieurs fois
~sillyRand2 = {rrand(0, 10)} ; // une fonction
~sillyRand2.value ; // évaluer plusieurs fois
```

En guise de dernier exemple, voici une fonction très utile.

```
// Utilisez cette fonction pour décider comment passer vos
journées d'été (
~whatToDo = {
var today, dayName, actions ;
        today = Date.getDate.dayOfWeek ;
        dayName =
        cas
        {today==0} {"Sunday"}
        {today==1} {"Monday"}
        {today==2} {"Tuesday"}
        {aujourd'hui==3} {"Mercredi"}
        {today==4} {"Thursday"}
        {aujourd'hui==5} {"Vendredi"}
        {today==6} {"Saturday"} ;
        actions = ["lancer de boomerang", "bras de fer", "monter des escaliers",
             "jouer aux échecs", "hockey subaquatique", "tirer des pois", "marathon
             de la sieste"] ;
        "Ah, " ++ dayName ++ "... ! " ++ "Quelle bonne journée pour " ++
        actions.choose ;
} ;
)

// L'exécuter le matin
~whatToDo.value ;
```

ASTUCE : Une autre notation courante pour déclarer des arguments au début d'une fonction est : f =
{|a, b| a + b}. Ceci est équivalent à f = {arg a, b ; a + b}

22 S'amuser avec les tableaux

Les tableaux sont le type de collection le plus courant dans SuperCollider. Chaque fois que vous écrivez une collection d'éléments entre crochets, comme [0, 1, 2], il s'agit d'une instance de la classe Tableau. Vous serez souvent amené à manipuler des tableaux de différentes manières. Voici une petite sélection de méthodes intéressantes que les tableaux comprennent :

```
// Créer un tableau
a = [10, 11, 12, 13, 14, 15, 16, 17]
;

a.reverse ; // reverse
a.scramble ;// scramble
a.choose ;// renvoie un élément au hasard
a.size ;   // renvoie la taille du tableau
a[0] ;  // récupère l'élément à la position spécifiée
a.wrapAt(9) ; // même chose que ci-dessus
a.size ["wow", 99] ++ a ; // récupère l'élément à la position spécifiée, en l'entourant si >
a ++ \hi ; // concatène les deux tableaux en un nouveau tableau
unique a ++ 'hi' ; // un symbole est un caractère
a ++ "hi" ; // comme ci-dessus
a.add(44) ;  // une chaîne est une collection de caractères
a.insert(5, "wow") ; // crée un nouveau tableau avec un nouvel
élément à la fin
    vers l'avant ( retourne un nouveau tableau)
a ; // insère "wow" à la position 5, pousse les autres éléments
    modifié le tableau d'origine
```

```
a.put(2, "oops") ; // met "oops" à l'index 2 (destructif ; évaluer à nouveau la
    ligne ci-dessus pour vérifier)
a.permute(3) ; // permute : l'élément en position 3 va en position 0, et vice-versa
a.mirror ; // en fait un palindrome
a.powerset ; // renvoie toutes les combinaisons possibles des éléments du tableau
```

Vous pouvez faire des calculs avec des tableaux :

```
[1, 2, 3, 4, 5] + 10 ;
[1, 2, 3, 4, 5] * 10 ;
([1, 2, 3, 4, 5] / 7).round(0.01) ; // remarquez les parenthèses pour la
priorité x = 11 ; y = 12 ; // essayez quelques variables
[x, y, 9] * 100 ;
// mais assurez-vous de ne faire des calculs qu'avec
des nombres corrects [1, 2, 3, 4, "oops", 11] + 10 ;
// résultat étrange
```

22.1 Création de nouveaux tableaux

Voici quelques façons d'utiliser la classe Array pour créer de nouvelles collections :

```
// Série arithmétique
Array.series(size : 6, start : 10, step : 3) ;
// Série géométrique
Array.geom(size : 10, start : 1, grow : 2) ;
// Comparez les deux :
Array.series(7, 100, -10) ; // 7 éléments ; commence à 100, pas de -10
Array.geom(7, 100, 0.9) ; // 7 éléments ; commencez à 100 ; multipliez par 0.9 à
chaque fois
// Utiliser la méthode .fill
Array.fill(10, "same") ;
// Comparez :
Array.fill(10, rrand(1, 10)) ;
Array.fill(10, {rrand(1, 10)}) ; // la fonction est réévaluée 10 fois
```

```
// La fonction de la méthode .fill peut prendre un argument par défaut qui est un
compteur.
// Le nom de l'argument peut être ce que vous
voulez. Array.fill(10, {arg counter ; counter *
10}) ;
// Par exemple, générer une liste de fréquences harmoniques :
Array.fill(10, {arg wow ; wow+1 * 440}) ;
// La méthode .newClear
a = Array.newClear(7) ; // crée un tableau vide de taille donnée
a[3] = "wow" ; // même chose que a.put(3, "wow")
```

22.2 Ce drôle de point d'exclamation

Ce n'est qu'une question de temps avant que vous ne voyiez quelque chose comme 30!4 dans le code de quelqu'un d'autre. Cette notation raccourcie crée simplement un tableau contenant le même élément un certain nombre de fois :

```
// Notation abrégée :
30!4 ;
"hello" ! 10 ;
// Cela donne les mêmes résultats que ce qui suit :
30.dup(4) ;
"hello".dup(10) ;
// ou
Array.fill(4, 30)
;
Array.fill(10, "hello") ;
```

22.3 Les deux points entre parenthèses

Voici un autre raccourci syntaxique couramment utilisé pour créer des tableaux.

```
// Qu'est-ce que
c'est ? (50..79) ;
```

```
// C'est un raccourci pour générer un tableau avec une série arithmétique de nombres.
// Ce qui précède donne le même résultat que
: series(50, 51, 79) ;
// ou
Array.series(30, 50, 1) ;
// Pour un pas différent de 1, on peut faire ceci :
(50, 53 .. 79) ; // pas de 3
// Même résultat
que : series(50,
53, 79) ;
Array.series(10, 50, 3) ;
```

Notez que chaque commande implique une façon de penser légèrement différente. La commande (50..79) vous permet de penser de la manière suivante : "*Donnez-moi un tableau de 50 à 79*". Vous ne pensez pas nécessairement au nombre d'éléments que le tableau finira par contenir. D'un autre côté, Array.series vous permet de penser : "*donnez-moi un tableau avec 30 éléments au total, en comptant à partir de 50*". Vous ne pensez pas nécessairement au dernier chiffre de la série.

Notez également que le raccourci utilise des parenthèses et non des crochets. Le tableau résultant sera bien entendu entre crochets.

22.4 Comment "faire" un tableau

Il est souvent nécessaire d'effectuer une action sur tous les éléments d'une collection. Nous pouvons utiliser la méthode do
pour cela :

```
~myFreqs = Array.fill(10, {rrand(440, 880)}) ;

// Effectuons maintenant une action simple sur chaque élément de la liste :
~myFreqs.do({arg item, count ; ("Item " ++ count ++ " is " ++ item ++ " Hz. La
    midinote la plus proche est " ++ item.cpsmidi.round).postln}) ;
```

5

```
// Si vous n'avez pas besoin du compteur, utilisez un seul argument :
~myFreqs.do({arg item ; {SinOsc.ar(item, 0, 0.1)}.play}) ;
~myFreqs.do({arg item ; item.squared.postln}) ;

// Bien entendu, une chose aussi simple que la dernière pourrait être réalisée de
cette manière :
~myFreqs.squared ;
```

En résumé : lorsque vous "faites" un tableau, vous fournissez une fonction. Le message do va itérer à travers les éléments du tableau et évaluer cette fonction à chaque fois. La fonction peut prendre deux arguments par défaut : l'élément du tableau à l'itération courante, et un compteur qui garde la trace du nombre d'itérations. Les noms de ces arguments peuvent être ce que vous voulez, mais ils sont toujours dans l'ordre suivant : item, count.

Voir aussi la méthode collect, qui est très similaire à do, mais qui renvoie une nouvelle collection avec tous les résultats intermédiaires.

23 Obtenir de l'aide

Apprenez à faire bon usage des fichiers d'aide. Vous trouverez souvent des exemples utiles au bas de chaque page d'aide. Ne manquez pas de faire défiler la page pour les consulter, même (ou surtout) si vous ne comprenez pas tout à fait les explications textuelles au début. Vous pouvez exécuter les exemples directement à partir du navigateur d'aide, ou copier et coller le code dans une nouvelle fenêtre pour jouer avec.

Sélectionnez n'importe quelle classe ou méthode valide dans votre code SuperCollider (double-cliquez sur le mot pour le sélectionner) et appuyez sur [ctrl+D] pour ouvrir le fichier d'aide correspondant. Si vous sélectionnez un nom de classe (par exemple, MouseX), vous serez dirigé vers le fichier d'aide de la classe. Si vous sélectionnez une méthode, vous serez dirigé vers une liste de classes qui comprennent cette méthode (par exemple, demandez de l'aide sur la méthode

méthode de brouillage).*

*Attention : SuperCollider affichera en bleu tout mot commençant par une lettre majuscule. Cela signifie que le

Les liens "Browse" et "Search" sont d'autres moyens d'explorer les fichiers d'aide dans l'IDE SuperCollider. Utilisez "Browse" pour naviguer dans les fichiers par catégories, et "Search" pour rechercher des mots dans tous les fichiers d'aide. Note importante concernant le navigateur d'aide dans l'IDE SuperCollider :

- Utilisez le champ en haut à droite (où il est indiqué "Rechercher...") pour rechercher des mots spécifiques *dans le fichier d'aide actuellement ouvert* (comme vous le feriez pour une recherche sur un site web) ;
- Utilisez le lien "Recherche" (à droite de "Parcourir") pour rechercher du texte *dans tous les fichiers d'aide*.

Lorsque vous ouvrez pour la première fois des parenthèses pour ajouter des arguments à une méthode donnée, SC affiche une petite "info-bulle" pour vous indiquer quels sont les arguments attendus. Par exemple, tapez le début de la ligne que vous voyez dans la figure 6. Juste après l'ouverture de la première parenthèse, l'infobulle indique que les arguments de SinOsc.ar sont freq, phase, mul et add. Elle indique également les valeurs par défaut. Il s'agit exactement des mêmes informations que celles que vous obtiendriez dans le fichier d'aide de SinOsc. Si l'infobulle a disparu, vous pouvez la faire réapparaître avec [ctrl+Shift+Space].

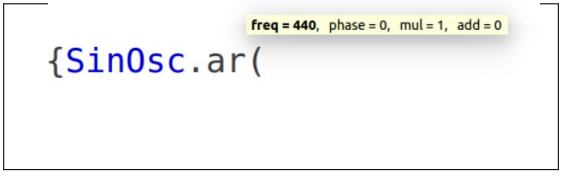

Figure 6 : Des informations utiles s'affichent au fur et à mesure de la saisie.

Autre raccourci : si vous souhaitez nommer explicitement vos arguments (comme SinOsc.ar(freq : 890)), essayez d'appuyer sur la touche tab juste après avoir ouvert les

parenthèses. SC complétera automatiquement le nom de l'argument

---

La couleur bleue *ne garantit pas que* le mot est exempt de fautes de frappe : par exemple, si vous tapez Sinosc (avec un "o" minuscule erroné), il apparaîtra quand même en bleu.

Le nom de l'argument correct pour vous, dans l'ordre, au fur et à mesure que vous tapez (appuyez sur la touche de tabulation après la virgule pour les noms d'arguments suivants).

CONSEIL : créez un dossier contenant vos propres "fichiers d'aide personnalisés". Chaque fois que vous comprendrez
Pour apprendre un nouveau truc ou un nouvel objet, écrivez un exemple simple avec des explications dans vos propres mots et gardez-le pour l'avenir. Il pourra s'avérer utile dans un mois ou un an.

Les mêmes fichiers d'aide sont également disponibles en ligne à l'adresse http://doc.sccode.org/.

Partie IV
LA SYNTHÈSE ET LE TRAITEMENT DU SON

A ce stade, vous en savez déjà beaucoup sur le SuperCollider. La dernière partie de ce tutoriel vous a présenté les détails du langage lui-même, des variables aux enceintes et plus encore. Vous avez également appris à créer des Pbinds intéressants en utilisant plusieurs membres de la famille Pattern.

Cette partie du tutoriel va (enfin !) vous présenter la synthèse et le traitement du son avec SuperCollider. Nous commencerons par le sujet des générateurs d'unités (UGens).*

24 UGens

Vous avez déjà vu quelques générateurs d'unités (UGen) en action dans les sections 3 et 18. Qu'est-ce qu'un UGen ? Un générateur d'unité est un objet qui génère des signaux sonores ou des signaux de contrôle. Ces signaux sont toujours calculés dans le serveur. Il existe de nombreuses classes de générateurs d'unités, qui dérivent toutes de la classe UGen. SinOsc et LFNoise0 sont des exemples d'UGen. Pour plus de détails, consultez les fichiers d'aide intitulés "Unit Generators and Synths" et "Tour of UGens".

Lorsque vous avez joué vos Pbinds plus tôt dans ce tutoriel, le son par défaut était toujours le même : un synthé simple ressemblant à un piano. Ce synthé est constitué d'une combinaison de générateurs d'unités.† Vous apprendrez à

*La plupart des tutoriels commencent d'emblée par les générateurs d'unités ; dans cette introduction à SC, cependant, nous avons choisi de mettre l'accent sur la famille des patrons (`Pbind` et ses amis) pour une approche pédagogique différente.

†Puisque vous avez utilisé des Pbinds pour produire du son dans SuperCollider jusqu'à présent, vous pourriez

être tenté de penser *: "Je vois, donc le Pbind est un générateur d'unité !"* Ce n'est pas le cas. `Pbind n'` est pas un générateur d'unités - c'est juste une recette pour créer des événements musicaux (partition). "*Alors le EventStreamPlayer, la chose qui résulte lorsque j'appelle play sur un Pbind, CELA doit être un UGen !* La réponse est toujours non. L'`EventStreamPlayer` n'est que le joueur, comme un pianiste, et il n'est pas un UGen.

comment combiner des générateurs d'unités pour créer toutes sortes d'instruments électroniques avec des sons synthétiques et traités. L'exemple suivant part de votre première onde sinusoïdale pour créer un instrument électronique que vous pouvez jouer en direct à l'aide de la souris.

24.1 Contrôle de la souris : Theremin instantané

Voici un synthé simple que vous pouvez jouer en direct. Il s'agit d'une simulation du Theremin, l'un des plus anciens instruments de musique électronique :

```
1 {SinOsc.ar(freq : MouseX.kr(300, 2500), mul : MouseY.kr(0, 1))}.play ;
```

Si vous ne savez pas ce qu'est un Theremin, arrêtez tout de suite et cherchez "Clara Rockmore Theremin" sur YouTube. Revenez ensuite ici et essayez de jouer la chanson du Cygne avec votre SC Theremin.

SinOsc, MouseX et MouseY sont des UGens. SinOsc génère une onde sinusoïdale. Les deux autres capturent le mouvement de votre curseur sur l'écran (X pour le mouvement horizontal, Y pour le mouvement vertical), et utilisent les nombres pour fournir des valeurs de fréquence et d'amplitude à l'onde sinusoïdale. C'est très simple et très amusant.

24.2 Vue et impulsion ; intrigue et portée

Le thérémine ci-dessus utilise un oscillateur sinusoïdal. Il existe d'autres formes d'onde que vous pourriez utiliser pour produire le son. Exécutez les lignes ci-dessous - elles utilisent la méthode de tracé pratique - pour étudier la forme de l'oscillateur sinusoïdal.

---

le pianiste ne produit pas de son. Pour rester dans cette métaphore limitée, le *piano* est l'instrument qui vibre et produit le son. C'est une analogie plus appropriée pour un UGen : ce n'est ni la partition, ni le joueur : c'est l'instrument. Lorsque vous faisiez de la musique avec Pbinds, SC créait un `EventStreamPlayer` pour jouer votre partition avec le synthétiseur de piano intégré. Vous n'aviez pas à vous soucier de créer le piano ou quoi que ce soit d'autre - SuperCollider faisait tout le travail sous le capot pour vous. Ce piano synthé caché est constitué

d'une combinaison d e quelques générateurs d'unités.

SinOsc, et comparez-la à Saw et Pulse. Les lignes ci-dessous ne produiront pas de son - elles vous permettent simplement de visualiser un instantané de la forme d'onde.

```
{ SinOsc.ar }.plot ; // onde sinusoïdale
{ Saw.ar }.plot ; // onde en dents de scie
{ Pulse.ar }.plot ; // onde carrée
```

Réécrivez maintenant votre ligne de thérémine en remplaçant SinOsc par Saw, puis Pulse. Ecoutez comme ils sonnent différemment. Enfin, essayez .scope au lieu de .play dans le code de votre thérémine, et vous pourrez observer une représentation de la forme d'onde en temps réel (une fenêtre "Stéthoscope" s'ouvrira).

25 Taux audio, taux de contrôle

Il est très facile de repérer un UGen dans le code de SuperCollider : ils sont presque toujours suivis des messages .ar ou .kr. Ces lettres signifient Audio Rate et Control Rate. Voyons ce que cela signifie.

Dans le fichier d'aide "Unit Generators and Synths" :

> Un générateur d'unité est créé en envoyant le message ar ou kr à l'objet de classe du générateur d'unité. Le message ar crée un générateur d'unité qui fonctionne à la fréquence audio. Le message kr crée un générateur d'unité qui fonctionne à la fréquence de contrôle. Les générateurs de taux de contrôle sont utilisés pour les signaux de contrôle à basse fréquence ou à variation lente. Les générateurs d'unités de taux de contrôle ne produisent qu'un seul échantillon par cycle de contrôle et utilisent donc moins de puissance de traitement que les générateurs d'unités de taux audio.*

En d'autres termes, lorsque vous écrivez SinOsc.ar, vous envoyez le message "taux audio" à la

fonction
SinOsc UGen. En supposant que votre ordinateur fonctionne à la fréquence d'échantillonnage courante de 44100 Hz,

*http://doc.sccode.org/Guides/UGens-and-Synths.html

cet oscillateur sinusoïdal génère 44100 échantillons par seconde à envoyer au haut-parleur. Nous entendons alors l'onde sinusoïdale.

Réfléchissez un instant à ce que vous venez de lire : lorsque vous envoyez le message ar à un UGen, vous lui demandez de générer *quarante-quatre mille cent* nombres par seconde. Cela fait beaucoup de chiffres. Vous écrivez {SinOsc.ar}.play dans le langage, et le langage communique votre demande au serveur. C'est le serveur, le "moteur sonore" de SuperCollider, qui se charge de générer tous ces échantillons.

Lorsque vous utilisez kr au lieu de ar, le travail est également effectué par le serveur, mais il y a quelques différences :

1. Le nombre de numéros générés par seconde avec .kr est beaucoup plus faible. {SinOsc.ar}.play génère 44100 numéros par seconde, tandis que {SinOsc.kr}.play génère un peu moins de 700 numéros par seconde (si vous êtes curieux, le nombre exact est 44100 / 64, où 64 est ce que l'on appelle la "période de contrôle").

2. Le signal généré par kr n'est pas envoyé à vos haut-parleurs. Au lieu de cela, il est normalement utilisé pour contrôler les paramètres d'autres signaux - par exemple, le MouseX.kr de votre thérémine contrôlait la fréquence d'un SinOsc.

Les UGens sont des générateurs de nombres incroyablement rapides. Certains de ces nombres deviennent des signaux sonores, d'autres des signaux de contrôle. Jusqu'ici, tout va bien. Mais de quels nombres s'agit-il ? Grands ? Petits ? Positifs ? Négatifs ? Il s'agit en fait de très petits nombres, souvent compris entre -1 et +1, parfois simplement entre 0 et 1. Tous les UGens peuvent être divisés en deux catégories en fonction de la gamme de nombres qu'ils génèrent : les UGens unipolaires et les UGens bipolaires.

Les UGens unipolaires génèrent des nombres entre 0 et 1.

Les UGens bipolaires génèrent des nombres compris entre -1 et +1.

25.1 La méthode des sondages

L'examen de la sortie de certains UGens devrait rendre les choses plus claires. Nous ne pouvons pas attendre de SuperCollider qu'il imprime des milliers de nombres par seconde dans la fenêtre Post, mais nous pouvons lui demander d'en imprimer quelques-uns chaque seconde, juste pour le plaisir. Tapez et exécutez les lignes suivantes une à la fois (assurez-vous que votre serveur fonctionne), et observez la fenêtre Post :

```
1 // regarder la fenêtre d'affichage (pas de son)
2 {SinOsc.kr(1).poll}.play ;
3 // appuyer sur ctrl+période, puis évaluer la ligne suivante :
4 {LFPulse.kr(1).poll}.play ;
```

Les exemples ne produisent aucun son parce que nous utilisons kr - le résultat est un signal de contrôle, donc rien n'est envoyé aux haut-parleurs. Il s'agit ici d'observer la sortie typique d'un SinOsc. Le message poll récupère 10 nombres par seconde de la sortie du SinOsc et les imprime dans la fenêtre Post. L'argument 1 est la fréquence, ce qui signifie simplement que l'onde sinusoïdale prendra une seconde pour accomplir un cycle complet. D'après ce que vous avez observé, SinOsc est-il unipolaire ou bipolaire ? Qu'en est-il de LFPulse?[6]

Baissez le volume avant d'évaluer la ligne suivante, puis remontez-le lentement. Vous devriez entendre des clics doux.

```
1 {LFNoise0.ar(1).poll}.play ;
```

Parce que nous lui avons envoyé le message ar, ce générateur de bruit à basse fréquence envoie 44100 échantillons par seconde à votre carte son - c'est un signal audio. Chaque échantillon est un nombre compris entre
-1 et +1 (il s'agit donc d'un UGen bipolaire). Avec le poll, vous n'en voyez que dix par seconde. LFNoise0.ar(1) choisit un nouveau nombre aléatoire toutes les secondes. Tout cela est fait par le serveur.

Arrêtez les clics avec [ctrl+.] et essayez de changer la fréquence de LFNoise0. Essayez des nombres comme 3, 5, 10, puis plus élevés. Observez les chiffres de sortie et écoutez les résultats.

26 Arguments d'UGen

La plupart du temps, vous voudrez spécifier des arguments aux UGens que vous utilisez. Vous l'avez déjà vu : lorsque vous écrivez {SinOsc.ar(440)}.play, le nombre 440 est un argument pour SinOsc.ar ; il spécifie la fréquence que vous allez entendre. Vous pouvez être explicite sur le nom des arguments, comme ceci : {SinOsc.ar(freq : 440, mul : 0.5)}.play. Les noms des arguments sont freq et mul (notez les deux points immédiatement après les mots dans le code). Le mot mul signifie "multiplicateur" et représente essentiellement l'amplitude de la forme d'onde. Si vous ne spécifiez pas mul, SuperCollider utilise la valeur par défaut de 1 (amplitude maximale). L'utilisation d'un mul : 0,5 signifie que la forme d'onde est multipliée par deux, en d'autres termes, elle sera jouée à la moitié de l'amplitude maximale. Dans le code de votre thérémine, les arguments SinOsc freq et mul ont été explicitement nommés. Vous vous souvenez peut-être que MouseX.kr(300, 2500) a été utilisé pour contrôler la fréquence du thérémine. MouseX.kr prend deux arguments : une limite basse et une limite haute pour sa plage de sortie. C'est ce que faisaient les nombres 300 et 2500. Même chose pour MouseY.kr(0, 1) qui contrôle l'amplitude.

Ces arguments à l'intérieur des UGens de la souris n'ont pas été explicitement nommés, mais ils pourraient l'être.

Comment savoir quels sont les arguments acceptés par un UGen ? Il suffit de se rendre dans le fichier d'aide correspondant : double-cliquez sur le nom de l'UGen pour le sélectionner et appuyez sur [ctrl+D] pour ouvrir la page de documentation. Faites-le maintenant pour, disons, MouseX. Après la section Description, vous voyez la section Class Methods. À cet endroit, il est indiqué que les arguments de la méthode kr sont minval, maxval, warp et lag. Sur la même page, vous pouvez apprendre ce que fait chacun d'entre eux.

Si vous ne fournissez pas d'argument, SC utilisera les valeurs par défaut indiquées dans le fichier d'aide. Si vous ne nommez pas explicitement les arguments, vous devez les fournir dans

l'ordre exact indiqué dans le fichier d'aide. Si vous les nommez explicitement, vous pouvez les placer dans n'importe quel ordre, et même en sauter certains au milieu. Nommer les arguments explicitement est également un bon outil d'apprentissage, car cela vous aide à mieux comprendre votre code. Un exemple est donné ci-dessous.

```
// minval et maxval fournis dans l'ordre, pas de mots-clés
```

```
{MouseX.kr(300, 2500).poll}.play ;
// minval, maxval et lag fournis, warp sauté
{MouseX.kr(minval : 300, maxval : 2500, lag : 10).poll}.play ;
```

27 Plages de mise à l'échelle

Le véritable plaisir commence lorsque vous utilisez certains UGens pour contrôler les paramètres d'autres UGens. C'est exactement ce qu'a fait l'exemple du thérémine. Vous disposez maintenant de tous les outils nécessaires pour comprendre exactement ce qui se passe dans l'un des exemples de la section 3. Les trois dernières lignes de l'exemple démontrent pas à pas comment le LFNoise0 est utilisé pour contrôler la fréquence :

```
{SinOsc.ar(freq : LFNoise0.kr(10).range(500, 1500), mul : 0.1)}.play ;

// La décomposition :
{LFNoise0.kr(1).poll}.play ; // regarder un LFNoise0 simple en action
{LFNoise0.kr(1).range(500, 1500).poll}.play ; // maintenant avec .range
{LFNoise0.kr(10).range(500, 1500).poll}.play ; // maintenant plus rapide
```

27.1 Échelle en fonction de l'étendue de la méthode

La gamme de méthodes permet simplement de rééchelonner la sortie d'un UGen. Rappelez-vous que LFNoise0 produit des nombres entre -1 et +1 (c'est un UGen bipolaire). Ces nombres bruts ne seraient pas très utiles pour contrôler la fréquence (nous avons besoin de nombres raisonnables dans la plage d'audition humaine). La fonction .range prend la sortie entre -1 et +1 et la met à l'échelle des valeurs basses et hautes que vous fournissez comme arguments (dans ce cas, 500 et 1500). Le nombre 10, qui est l'argument de LFNoise0.kr, spécifie la fréquence de l'UGen : combien de fois par seconde il choisira un nouveau nombre

aléatoire.

En bref : pour utiliser un UGen afin de contrôler un paramètre d'un autre UGen, vous devez d'abord savoir quelle plage de nombres vous souhaitez obtenir. S'agit-il de fréquences ? Voulez-vous qu'ils se situent entre, disons, 100 et 1000 ? Ou s'agit-il d'amplitudes ? Peut-être voulez-vous que les amplitudes soient comprises entre 0,1 (faible) et 0,5 (moitié du maximum) ? Ou essayez-vous de contrôler le nombre d'harmoniques ? Voulez-vous qu'il soit compris entre 5 et 19 ?

Une fois que vous connaissez la plage dont vous avez besoin, utilisez la méthode .range pour que l'UGen de contrôle fasse ce qu'il faut.

Exercice : écrivez un code simple qui joue une onde sinusoïdale dont la fréquence est contrôlée par un LFPulse.kr (fournissez-lui les arguments appropriés). Ensuite, utilisez la méthode .range pour mettre à l'échelle la sortie de LFPulse afin d'obtenir quelque chose que vous souhaitez entendre.

27.2 Échelle avec mul et ajouter

Vous savez maintenant comment mettre à l'échelle la sortie des UGens dans le serveur en utilisant la méthode .range. La même chose peut être accomplie à un niveau plus fondamental en utilisant les arguments mul et add, que pratiquement tous les UGens possèdent. Le code ci-dessous montre l'équivalence entre les approches range et mul/add, à la fois avec un UGen bipolaire et un UGen unipolaire.

```
// Ceci :
{SinOsc.kr(1).range(100, 200).poll}.play ;
// ...est le même que celui-ci :
{SinOsc.kr(1, mul : 50, add : 150).poll}.play ;

// Ceci :
{LFPulse.kr(1).range(100, 200).poll}.play ;
// ...est le même que celui-ci :
{LFPulse.kr(1, mul : 50, add : 100).poll}.play ;
```

La figure 7 permet de visualiser le fonctionnement de mul et add dans le redimensionnement des sorties UGen (un SinOsc est utilisé comme démonstration).

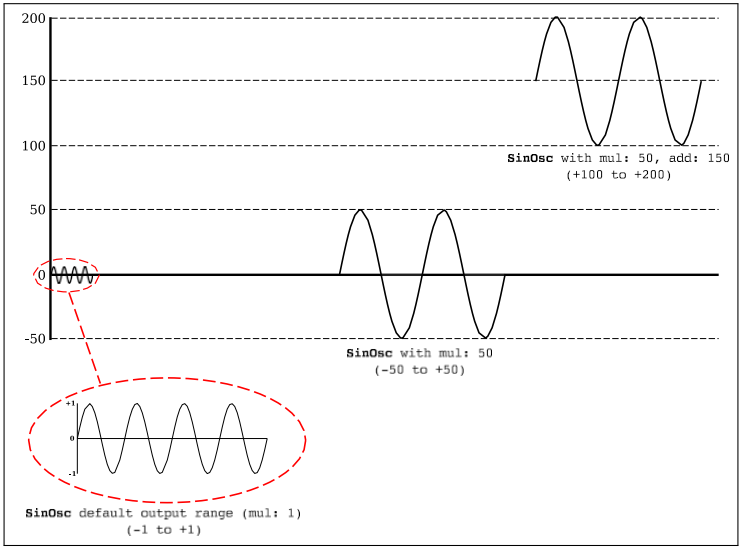

Figure 7 : Mise à l'échelle des plages UGen avec mul et add

27.3 linlin et ses amis

Pour toute autre mise à l'échelle arbitraire de plages, vous pouvez utiliser les méthodes pratiques linlin, linexp, explin, expexp. Les noms des méthodes indiquent ce qu'elles font : convertir un intervalle linéaire en un autre intervalle linéaire (linlin), linéaire en exponentiel (linexp), etc.

```
// Un tas de chiffres
a = [1, 2, 3, 4, 5, 6, 7] ;
// Rééchelle de 0 à 127, linéaire à
linéaire a.linlin(1, 7, 0, 127).round(1)
;
// Rééchelle de 0 à 127, linéaire à exponentiel
a.linexp(1, 7, 0.01, 127).round(1) ; // ne pas utiliser zéro pour une plage
exponentielle
```

Pour revoir les notions de linéaire et d'exponentiel, consultez en ligne la différence entre les suites arithmétiques et les suites géométriques. En bref, les suites linéaires (arithmétiques) sont du type "1, 2, 3, 4, 5, 6" ou "3, 6, 9",
12, 15", etc. ; et les séquences exponentielles (géométriques) sont du type "1, 2, 4, 8, 16, 32" ou "3, 9, 27, 81",
243", etc.

28 Arrêt des synthés individuels

Voici une façon très courante de démarrer plusieurs synthés et de pouvoir les arrêter séparément. L'exemple est explicite :

```
// Exécuter une ligne à la fois (ne pas arrêter le son entre les deux) :
a = { Saw.ar(LFNoise2.kr(8).range(1000, 2000), mul : 0.2) }.play ;
b = { Saw.ar(LFNoise2.kr(7).range(100, 1000), mul : 0.2) }.play ;
c = { Saw.ar(LFNoise0.kr(15).range(2000, 3000), mul : 0.1) }.play ;
// Arrêter les synthés
individuellement : a.free ;
b.libre
;
c.libre
;
```

29 Le message du `set`

Comme pour toute fonction (voir section 21), les arguments spécifiés au début de la fonction du synthé sont accessibles à l'utilisateur. Cela vous permet de modifier les paramètres du synthé à la volée (pendant que le synthé fonctionne). Le jeu de messages est utilisé à cette fin. Exemple simple :

```
1 x = {arg freq = 440, amp = 0.1 ; SinOsc.ar(freq, 0, amp)}.play ;
2 x.set(\freq, 778) ;
3 x.set(\amp, 0.5) ;
4 x.set(\freq, 920, \amp, 0.2) ;
5 x.free ;
```

Il est conseillé de fournir des valeurs par défaut (comme les 440 et 0,1 ci-dessus), sinon le synthé ne jouera pas tant que vous n'aurez pas défini une valeur correcte pour les paramètres "vides".

30 Bus audio

Les bus audio sont utilisés pour acheminer les signaux audio. Ils sont comme les canaux d'une table de mixage. SuperCollider possède 128 bus audio par défaut. Il y a aussi des bus de contrôle (pour les signaux de contrôle), mais pour l'instant nous allons nous concentrer uniquement sur les bus audio.*

Appuyez sur [ctrl+M] pour ouvrir la fenêtre Meter. Elle affiche les niveaux de toutes les entrées et sorties. La figure 8 montre une capture d'écran de cette fenêtre et sa correspondance avec les bus par défaut de SuperCollider. Dans SuperCollider, les bus audio sont numérotés de 0 à 127. Les huit premiers (0-7) sont par défaut réservés aux canaux de sortie de votre carte son. Les huit suivants (8-15) sont réservés aux entrées de votre carte son. Tous les autres (16 à 127) sont libres d'être utilisés comme vous le souhaitez, par exemple lorsque vous devez acheminer des

signaux audio d'un UGen à un autre.

*Nous examinerons rapidement les bus de contrôle à la section 41.

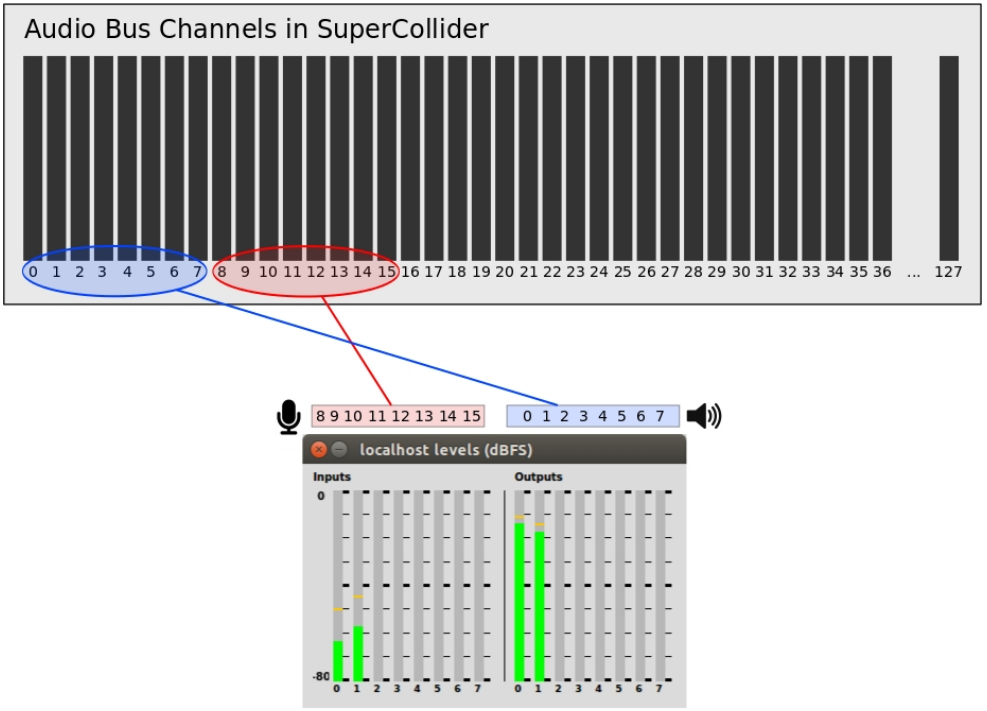

Figure 8 : Bus audio et fenêtre Meter dans SC.

30.1 Les UGens de l'extérieur et de l'intérieur

Essayez maintenant la ligne de code suivante :

```
{Out.ar(1, SinOsc.ar(440, 0, 0.1))}.play ; // canal droit
```

L'UGen Out se charge de l'acheminement des signaux vers des bus spécifiques.

Le premier argument de Out est le bus cible, c'est-à-dire l'endroit où vous voulez que ce signal aille. Dans l'exemple ci-dessus, le chiffre 1 signifie que nous voulons envoyer le signal au bus 1, qui est le canal droit de votre carte son.

Le deuxième argument de Out.ar est le signal réel que vous voulez "écrire" dans ce bus. Il peut s'agir d'un seul UGen ou d'une combinaison d'UGens. Dans l'exemple, il s'agit simplement d'une onde sinusoïdale. Vous ne devriez l'entendre que sur votre haut-parleur droit (ou sur votre oreille droite si vous utilisez un casque).

La fenêtre des compteurs étant ouverte et visible, modifiez le premier argument de Out.ar : essayez n'importe quel nombre entre 0 et 7, et observez les compteurs. Vous verrez que le signal va là où vous le souhaitez.

ASTUCE : Il est très probable que votre carte son ne puisse lire que deux canaux (gauche et droite),
vous n'entendrez donc la tonalité sinusoïdale que lorsque vous l'enverrez au bus 0 ou au bus 1. Lorsque vous l'envoyez à d'autres bus (3 à 7), vous verrez toujours le compteur correspondant afficher le signal : SC envoie en fait le son à ce bus, mais à moins d'avoir une carte son à 8 canaux, vous ne pourrez pas entendre la sortie des bus 3 à 7.

Un exemple simple d'utilisation d'un bus audio pour un effet est illustré ci-dessous.

```
// démarrer l'effet
f = {Out.ar(0, BPF.ar(in : In.ar(55), freq : MouseY.kr(1000, 5000), rq : 0.1))}.play
;
// démarrer la source
n = {Out.ar(55, WhiteNoise.ar(0.5))}.play;
```

4

La première ligne déclare un synthé (stocké dans la variable f) constitué d'un filtre UGen (un filtre passe-bande). Un filtre passe-bande prend n'importe quel son en entrée, et *filtre toutes les fréquences à l'exception de celle que vous voulez laisser passer*. In.ar est l'UGen que nous utilisons pour lire un bus audio ; ainsi, avec In.ar(55) utilisé comme entrée du BPF, n'importe quel son envoyé au bus 55 sera transmis au filtre passe-bande. Remarquez que ce premier synthé n'émet aucun son au début : lorsque vous évaluez la première ligne, le bus 55 est encore vide. Il n'émettra de son que lorsque nous enverrons de l'audio dans le bus 55, ce qui se produit sur la deuxième ligne.

La deuxième ligne crée un synthé et le stocke dans la variable n. Ce synthé génère simplement du bruit blanc et le transmet *non pas directement aux haut-parleurs, mais au bus audio 55*. C'est précisément le bus que notre synthétiseur de filtre écoute, donc dès que vous évaluez la deuxième ligne, vous devriez commencer à entendre le bruit blanc filtré par le synthétiseur f. En bref, le routage ressemble à ceci :

noise synth → bus 55 → filter synth

L'ordre d'exécution est important. L'exemple précédent ne fonctionnera pas si vous évaluez la source *avant* l'effet. Ce point sera abordé plus en détail dans la section 42, "Ordre d'exécution".

Une dernière chose : lorsque vous avez écrit dans les exemples précédents des synthés comme {SinOsc.ar(440)}.play, SC faisait en fait {Out.ar(0, SinOsc.ar(440))}.play sous le capot : il supposait que vous vouliez envoyer le son sur le bus 0, donc il enveloppait automatiquement le premier UGen avec un Out.ar(0, …) UGen. En fait, quelques autres choses se passent dans les coulisses, mais nous y reviendrons plus tard (section 39).

31 Entrée du microphone

L'exemple ci-dessous montre comment vous pouvez facilement accéder à un son provenant de votre carte son à l'aide de la commande
SoundIn UGen.*

```
// Avertissement : utilisez des écouteurs pour éviter tout effet de rétroaction
{SoundIn.ar(0)}.play ; // identique à In.ar(8) : prend le son du premier bus d'entrée

// Version stéréo
{SoundIn.ar([0, 1])}.play ; // première et deuxième entrées

// Un peu de réverbération juste pour le plaisir ?
{FreeVerb.ar(SoundIn.ar([0, 1]), mix : 0.5, room : 0.9)}.play ;
```

32 Expansion multicanal

Avec votre fenêtre Meter ouverte - [ctrl+M]-, regardez ceci.

```
{Out.ar(0, Saw.ar(freq : [440, 570], mul : Line.kr(0, 1, 10)))}.play ;
```

Nous utilisons un UGen Line.kr pour faire passer l'amplitude de 0 à 1 en 10 secondes. C'est très bien. Mais il y a une magie plus intéressante qui s'opère ici. Avez-vous remarqué qu'il y a deux canaux de sortie (gauche et droite) ? Avez-vous entendu qu'il y a une note différente sur chaque canal ? Et que ces deux notes proviennent d'une *liste* -[440, 570]- qui est passée à Saw.ar comme argument freq ?

*Puisque `In.ar` lit à partir de n'importe quel bus, et que vous savez que les entrées de votre carte son sont par défaut assignées aux bus 8-15, vous pourriez écrire `In.ar(8)` pour obtenir le son de votre microphone. Cela fonctionne très bien, mais `SoundIn.ar` est une option plus pratique.

C'est ce qu'on appelle l'expansion multicanal.

David Cottle plaisante en disant que "l'expansion multicanal est une [application des réseaux] qui frise le vaudou"* C'est l'une des caractéristiques les plus puissantes et les plus uniques de SuperCollider, et qui peut laisser perplexe au premier abord.

En bref : si vous utilisez un tableau n'importe où comme l'un des arguments d'un UGen, *l'ensemble du patch est dupliqué*. Le nombre de copies créées correspond au *nombre d'éléments du tableau*. Ces UGen dupliqués sont envoyés à autant de *bus adjacents* que nécessaire, en commençant par le bus spécifié comme premier argument de Out.ar.

Dans l'exemple ci-dessus, nous avons Out.ar(0, …). La fréquence de l'onde Saw est un tableau de deux éléments : [440, 570]. Que fait SC ? Il procède à une "expansion multicanal", en créant deux copies de l'ensemble du patch. La première copie est une onde en dents de scie de fréquence 440 Hz, envoyée au bus 0 (votre canal gauche) ; la seconde copie est une onde en dents de scie de fréquence 570 Hz, envoyée au bus 1 (votre canal droit) !

Allez-y et vérifiez par vous-même. Remplacez ces deux fréquences par d'autres valeurs de votre choix. Écoutez les résultats. L'une va sur le canal gauche, l'autre sur le canal droit. Allez encore plus loin et ajoutez une troisième fréquence à la liste (disons [440, 570, 980]). Observez la fenêtre des compteurs. Vous verrez que les trois premières sorties s'allument (mais vous ne pourrez entendre que la troisième si vous avez une carte son multicanal).

De plus, vous pouvez utiliser des tableaux supplémentaires dans d'autres arguments du même UGen, ou dans les arguments d'autres UGen dans le même synthé. SuperCollider fera le ménage et générera des synthés qui suivront ces valeurs en conséquence. Par exemple : en ce moment, les deux fréquences [440, 570] passent de 0 à 1 en 10 secondes. Mais changez le code en Line.kr(0, 1, [1, 15]) et vous obtiendrez un fondu en 1 seconde pour le son 440 Hz et en 15 secondes pour le son 570 Hz. Essayez-le.

---

*Cottle, D. "Beginner's Tutorial". The SuperCollider Book, MIT Press, 2011, p. 14.

Exercice : écoutez cette simulation de la "tonalité occupée" d'un vieux téléphone. Elle utilise l'expansion multicanal pour créer deux oscillateurs sinusoïdaux, chacun jouant une fréquence différente sur un canal différent. Faites en sorte que le canal gauche pulse 2 fois par seconde et le canal droit 3 fois par seconde.[7]

```
a = {Out.ar(0, SinOsc.ar(freq : [800, 880], mul : LFPulse.ar(2)))}.play ;
a.free ;
```

33 L'objet Bus

Voici un exemple qui utilise tout ce que vous venez d'apprendre dans les deux sections précédentes : les bus audio et l'expansion multicanal.

```
// Exécutez ceci en premier ("activer la réverbération" -- vous n'entendrez rien au
début)
r = {Out.ar(0, FreeVerb.ar(In.ar(55, 2), mix : 0.5, room : 0.9, mul : 0.4))}.play ;

// Exécutez maintenant cette seconde ('feed the busy tone into
the reverb bus') a = {Out.ar(55, SinOsc.ar([800, 880], mul :
LFPulse.ar(2)))}.play ; a.free ;
```

Grâce à l'expansion multicanal, la tonalité d'occupation utilise deux canaux. Lorsque (dans le synthé a) nous acheminons la tonalité d'occupation vers le bus 55, deux bus sont en fait utilisés - le bus 55 et le bus 56 immédiatement adjacent. Dans la réverbération (synth r), nous indiquons avec In.ar(55, 2) que nous voulons lire 2 canaux à partir du bus 55 : ainsi, les deux bus 55 et 56 entrent dans la réverbération. La sortie de la réverbération est à son tour étendue à deux canaux, de sorte que le synthé r envoie le son aux bus 0 et 1 (canaux gauche et droit de notre carte son).

Or, ce choix du numéro de bus (55) pour relier un synthé source à un synthé effet est arbitraire : il aurait pu être n'importe quel autre numéro entre 16 et 127 (rappelons que les bus 0 à 15 sont

réservés aux sorties et entrées des cartes son). Quel inconvénient il y aurait à devoir suivre le bus

nous-mêmes. Dès que nos patchs sont devenus plus complexes, imaginez le cauchemar : "Quel numéro de bus ai-je encore choisi pour la réverbération ? Était-ce 59 ou 95 ? Et le numéro de bus pour mon delay ? Je suppose que c'était 27 ? Je ne me souviens plus..." et ainsi de suite.

SuperCollider s'en charge pour vous avec les objets Bus. Dans les exemples ci-dessus, nous n'avons attribué à la main le fameux bus 55 que pour les besoins de la démonstration. Dans votre vie quotidienne avec SuperCollider, vous devriez simplement utiliser l'objet Bus. L'objet Bus se charge de choisir un bus disponible pour vous et d'en garder la trace. Voici comment l'utiliser :

```
// Créer le bus
~myBus = Bus.audio(s, 2) ;
// Activation de la réverbération : lecture à partir de myBus (source sonore)
r = {Out.ar(0, FreeVerb.ar(In.ar(~myBus, 2), mix : 0.5, room : 0.9, mul :
0.4))}.play ;
// Introduire la tonalité d'occupation dans ~myBus
b = {Out.ar(~myBus, SinOsc.ar([800, 880], mul : LFPulse.ar(2)))}.play ;
// Libère les deux
synthés r.free ;
b.free ;
```

Le premier argument de Bus.audio est la variable s, qui représente le serveur. Le deuxième argument est le nombre de canaux dont vous avez besoin (2 dans l'exemple). Vous stockez ensuite ces données dans une variable portant un nom significatif (~myBus dans l'exemple, mais il pourrait s'agir de ~reverbBus, ~source, ~tangerine, ou ce qui vous semble logique dans votre patch). Ensuite, chaque fois que vous aurez besoin de faire référence à ce bus, il suffit d'utiliser la variable que vous avez créée.

34 Panoramique

Le panoramique est l'étalement d'un signal audio dans un champ sonore stéréo ou multicanal. Voici un signal mono qui rebondit entre les canaux gauche et droit grâce à Pan2:*

```
p = {Pan2.ar(in : PinkNoise.ar, pos : SinOsc.kr(2), level : 0.1)}.play
; p.free ;
```

Dans le fichier d'aide de Pan2, vous pouvez voir que l'argument pos (position) attend un nombre entre -1 (à gauche) et +1 (à droite), 0 étant le centre. C'est pourquoi vous pouvez utiliser un SinOsc directement dans cet argument : l'oscillateur sinusoïdal est un UGen bipolaire, il produit donc des nombres entre -1 et +1 par défaut.

Voici un exemple plus élaboré. Une onde en dents de scie passe à travers un filtre passe-bande très fin (rq : 0,01). Remarquez l'utilisation de variables locales pour modulariser les différentes parties du code. Analysez et essayez de comprendre autant que possible l'exemple ci-dessus. Répondez ensuite aux questions ci-dessous.

```
(
x = {
        var lfn = LFNoise2.kr(1) ;
        var saw = Saw.ar(
                Fréquence : 30,
                mul : LFPulse.kr(
                        freq : LFNoise1.kr(1).range(1, 10),
                        largeur : 0.1)) ;
        var bpf = BPF.ar(in : saw, freq : lfn.range(500, 2500), rq : 0.01, mul : 20)
        ; Pan2.ar(in : bpf, pos : lfn) ;
}.play ;
```

*Pour les panoramiques multicanaux, jetez un coup d'œil à Pan4 et PanAz. Les utilisateurs avancés peuvent jeter un coup d'œil aux plug-ins SuperCollider pour Ambisonics.

```
)
x.gratuit ;
```

Questions :

(a) La variable lfn est utilisée à deux endroits différents. Pourquoi (quel est le résultat ?)

(b) Que se passe-t-il si vous changez l'argument mul : du BPF de 20 à 10, 5 ou 1 ? Pourquoi un nombre aussi élevé que 20 a-t-il été utilisé ?

(c) Quelle partie du code contrôle le rythme ? Les

réponses se trouvent à la fin de ce document.[8]

35 Mélanger et étaler

Voici une astuce intéressante. Vous pouvez utiliser l'expansion multicanal pour générer des sons complexes, puis mixer le tout en mono ou en stéréo avec Mix ou Splay :

```
// 5 canaux de sortie (regarder la fenêtre Meter)
a = { SinOsc.ar([100, 300, 500, 700, 900], mul : 0.1) }.play ;
a.libre ;
// Le mixage est effectué en mono :
b = { Mix(SinOsc.ar([100, 300, 500, 700, 900], mul : 0.1)) }.play ;
b.libre ;
// Mixer en stéréo (répartir uniformément de gauche à droite)
c = { Splay.ar(SinOsc.ar([100, 300, 500, 700, 900], mul : 0.1)) }.play ;
c.free
// S'amuser avec Splay :
(
d = {arg fundamental = 110 ;
```

```
        var harmoniques = [1, 2, 3, 4, 5, 6, 7, 8, 9] ;
        var snd = BPF.ar(
                dans : Saw.ar(32, LFPulse.ar(harmonics, width :
                0.1)),
                freq : harmoniques * fondamental,
                rq : 0,01,
                mul : 20) ;
        Splay.ar(snd) ;
}.play ;
)
d.set(\fundamental, 100) ; // changer la fondamentale juste pour
le plaisir d.free ;
```

Pouvez-vous voir l'expansion multicanal à l'œuvre dans le dernier exemple de Splay ? La seule différence est que le tableau est d'abord stocké dans une variable (harmonics) avant d'être utilisé dans l'UGens. Le tableau harmonics a 9 éléments, donc le synthé va s'étendre à 9 canaux. Ensuite, juste avant le .play, Splay prend le tableau de neuf canaux et le mixe en stéréo, en répartissant l e s canaux uniformément de gauche à droite.*

Mix dispose d'une autre astuce intéressante : la méthode de **remplissage**. Elle crée un ensemble de synthétiseurs et les mixe en mono en une seule fois.

```
// Générateur instantané de grappes
c = { Mix.fill(16, {SinOsc.ar(rrand(100, 3000), mul : 0.01)}) }.play ;
c.free ;
// Une note avec 12 partiels d'amplitudes
décroissantes (
n = { Mix.fill(12, {arg counter ;
        var partial = counter + 1 ; // nous voulons que le compteur commence à 1 et
        non à 0
```

*La dernière ligne avant le `.play` pourrait être explicitement écrite comme `Out.ar(0, Splay.ar(snd))`. Rappelez-vous que SuperCollider remplit gracieusement les vides et ajoute un `Out.ar(0...)` ici - c'est ainsi que le synthé sait qu'il doit jouer dans vos canaux gauche (bus 0) et droit (bus 1).

```
8           SinOsc.ar(partial * 440, mul : 1/partial.squared) * 0.1
9           })
10  }.play ;
11  FreqScope.new ;
12  )
13  n.libre ;
```

Vous donnez deux choses à Mix.fill : la taille du tableau que vous voulez créer et une fonction (entre accolades) qui sera utilisée pour remplir le tableau. Dans le premier exemple ci-dessus, Mix.fill évalue la fonction 16 fois. Notez que la fonction comprend une composante variable : la fréquence de l'oscillateur sinusoïdal peut être n'importe quel nombre aléatoire entre 100 et 3000. Seize ondes sinusoïdales seront créées, chacune avec une fréquence aléatoire différente. Elles seront toutes mixées en mono et vous entendrez le résultat sur votre canal gauche. Le deuxième exemple montre que la fonction peut prendre un argument "counter" qui comptabilise le nombre d'itérations (tout comme Array.fill). Douze oscillateurs sinusoïdaux sont générés en suivant la série harmonique, et mixés en une seule note en mono.

36 Lecture d'un fichier audio

Tout d'abord, vous devez charger le fichier son dans un tampon. Le deuxième argument de Buffer.read est le chemin de votre fichier son entre guillemets. Vous devrez le modifier en conséquence pour qu'il pointe vers un fichier WAV ou AIFF sur votre ordinateur. Une fois les tampons chargés, il suffit d'utiliser l'UGen PlayBuf pour les lire de différentes manières.

ASTUCE : Un moyen rapide d'obtenir le chemin correct d'un fichier son sauvegardé sur votre ordinateur est de faire glisser le fichier sur un document SuperCollider vierge. SC vous donnera automatiquement le chemin complet, toujours entre guillemets !

```
// Chargement des fichiers dans les tampons :
~buf1 = Buffer.read(s, "/home/Music/wheels-mono.wav") ; // un fichier son
~buf2 = Buffer.read(s, "/home/Music/mussorgsky.wav") ; // un autre fichier son

// Lecture :
{PlayBuf.ar(1, ~buf1)}.play ; // nombre de canaux et tampon
{PlayBuf.ar(1, ~buf2)}.play ;

// Obtenir des informations sur les fichiers :
[~buf1.bufnum, ~buf1.numChannels, ~buf1.path, ~buf1.numFrames] ;
[~buf2.bufnum, ~buf2.numChannels, ~buf2.path, ~buf2.numFrames] ;

// Modification de la vitesse de lecture avec "rate
{PlayBuf.ar(numChannels : 1, bufnum : ~buf1, rate : 2, loop : 1)}.play ;
{PlayBuf.ar(1, ~buf1, 0.5, loop : 1)}.play ; // jouer à la moitié de la vitesse
{PlayBuf.ar(1, ~buf1, Line.kr(0.5, 2, 10), loop : 1)}.play ; // accélération
{PlayBuf.ar(1, ~buf1, MouseY.kr(0.5, 3), loop : 1)}.play ; // contrôle de la souris

// Changement de direction (marche arrière)
{PlayBuf.ar(1, ~buf2, -1, loop : 1)}.play ; // inverser le son
{PlayBuf.ar(1, ~buf2, -0.5, loop : 1)}.play ; // lecture à la moitié de la vitesse ET
inversée
```

37 Nœuds de synthèse

Dans les exemples précédents de PlayBuf, vous deviez appuyer sur [ctrl+.] après chaque ligne pour arrêter le son. Dans d'autres exemples, vous avez assigné le synthé à une variable (comme x = {WhiteNoise.ar}.play) afin de pouvoir l'arrêter directement avec x.free.

Chaque fois que vous créez un synthé dans SuperCollider, vous savez qu'il s'exécute dans le serveur, notre "moteur sonore". Chaque synthé en cours d'exécution dans le serveur est représenté

par un *nœud*. Nous pouvons jeter un coup d'œil

à cet arbre de nœuds avec la commande s.plotTree. Essayez-le. Une fenêtre nommée NodeTree s'ouvre.

```
// ouvrir l'interface
graphique s.plotTree ;
// exécutez-les un par un (n'arrêtez pas le son) et observez l'arbre
des nœuds : w = { SinOsc.ar(60.midicps, 0, 0.1) }.play ;
x = { SinOsc.ar(64.midicps, 0, 0.1) }.play ;
y = { SinOsc.ar(67.midicps, 0, 0.1) }.play ;
z = { SinOsc.ar(71.midicps, 0, 0.1) }.play
; w.free ;
x.libre
;
y.libre
;
z.libre
;
```

Chaque rectangle que vous voyez dans l'arbre des nœuds est un nœud de synthétiseur. Chaque synthé reçoit un nom temporaire (quelque chose comme temp_101, temp_102, etc) et reste à cette place tant qu'il fonctionne. Essayez maintenant de jouer à nouveau les quatre sinus, et appuyez sur [ctrl+.] (regardez la fenêtre de l'arborescence des nœuds). Le raccourci [ctrl+.] arrête impitoyablement et immédiatement tous les nœuds en cours d'exécution dans le serveur. En revanche, avec la méthode .free, vous pouvez être plus subtil et libérer des nœuds spécifiques un par un.

Il est important de savoir que les synthés peuvent continuer à fonctionner dans le serveur même s'ils ne génèrent que du silence. Voici un exemple. L'amplitude de cet UGen WhiteNoise passe de 0,2 à 0 en deux secondes. Après cela, nous n'entendons plus rien. Mais vous verrez que le noeud synth est toujours là, et qu'il ne disparaîtra pas tant que vous ne l'aurez pas libéré.

```
// Évaluer et regarder la fenêtre de l'arbre des nœuds
pendant quelques secondes x = {WhiteNoise.ar(Line.kr(0.2, 0,
2))}.play ;
x.gratuit ;
```

37.1 Le glorieux doneAction : 2

Heureusement, il est possible de rendre les synthés plus intelligents à cet égard : par exemple, ne serait-il pas formidable de demander à Line.kr de notifier le synthé lorsqu'il a terminé son travail (la rampe de 0.2 à 0), ce qui permettrait au synthé de se libérer automatiquement ?

Introduire l'argument doneAction : 2 pour résoudre tous nos problèmes.

Jouez les exemples ci-dessous et comparez leur comportement avec et sans doneAction : 2. Surveillez l'arborescence des nœuds pendant que vous exécutez les lignes.

```
// sans doneAction : 2
{WhiteNoise.ar(Line.kr(0.2, 0, 2))}.play ;
{PlayBuf.ar(1, ~buf1)}.play ; // PS. ceci suppose que vous avez toujours votre
    fichier son chargé dans ~buf1 depuis la section précédente.

// avec doneAction : 2
{WhiteNoise.ar(Line.kr(0.2, 0, 2, doneAction : 2))}.play ;
{PlayBuf.ar(1, ~buf1, doneAction : 2)}.play ;
```

Les synthés avec doneAction : 2 se libèrent automatiquement dès que leur travail est terminé (c'est-à-dire dès que la rampe Line.kr est terminée dans le premier exemple, et dès que PlayBuf.ar a fini de jouer le fichier son dans le second exemple). Cette connaissance sera très utile dans la section suivante : Les enveloppes.

38 Enveloppes

Jusqu'à présent, la plupart de nos exemples concernaient des sons continus. Il est temps d'apprendre à façonner l'enveloppe d'amplitude d'un son. Un bon exemple pour commencer est l'enveloppe d'une percussion. Imaginez un coup de cymbale. Le temps nécessaire pour que le son passe du silence à l'amplitude maximale est très court - quelques millisecondes, peut-être. C'est ce qu'on appelle le *temps d'attaque*. Le temps nécessaire pour que le son passe du silence à l'amplitude

maximale est très petit - quelques millisecondes, par exemple.

le son de la cymbale pour passer de l'amplitude maximale au silence (zéro) est un peu plus long, peut-être quelques secondes. C'est ce qu'on appelle le *temps de relâchement*.

Pensez à une enveloppe d'amplitude simplement comme un nombre qui change dans le temps pour être utilisé comme multiplicateur (mul) de n'importe quel UGen produisant du son. Ces nombres doivent être compris entre 0 (silence) et 1 (ampli à fond), parce que c'est ainsi que SuperCollider comprend l'amplitude. Vous avez peut-être déjà réalisé que le dernier exemple comprenait déjà une enveloppe d'amplitude : dans {WhiteNoise.ar(Line.kr(0.2, 0, 2, doneAction : 2))}.play, nous faisons passer l'amplitude du bruit blanc de 0.2 à 0 en 2 secondes. Une Line.kr n'est cependant pas un type d'enveloppe très flexible.

Env est l'objet que vous utiliserez en permanence pour définir toutes sortes d'enveloppes. Env possède de nombreuses méthodes utiles ; nous n'en examinerons que quelques-unes ici. N'hésitez pas à consulter le fichier d'aide Env pour en savoir plus.

38.1 Env.perc

Env.perc est un moyen pratique d'obtenir une enveloppe de percussion. Il prend en compte quatre arguments : attackTime, releaseTime, level, et curve. Jetons un coup d'oeil à quelques formes typiques, en dehors de tout synthé.

```
Env.perc.plot ; // utilise tous les args
par défaut Env.perc(0.5).plot ; //
attackTime : 0.5
Env.perc(attackTime : 0.3, releaseTime : 2, level : 0.4).plot ;
Env.perc(0.3, 2, 0.4, 0).plot ; // même chose que ci-dessus, mais curve:0 signifie lignes droites
```

Il suffit maintenant de le brancher sur un synthétiseur comme celui-ci :

```
{PinkNoise.ar(Env.perc.kr(doneAction : 2))}.play ; // args Env.perc par défaut
{PinkNoise.ar(Env.perc(0.5).kr(doneAction : 2))}.play ;
{PinkNoise.ar(Env.perc(0.3, 2, 0.4).kr(2))}.play ;
{PinkNoise.ar(Env.perc(0.3, 2, 0.4, 0).kr(2))}.play ;
```

4

Il suffit d'ajouter le bit .kr(doneAction : 2) juste après Env.perc, et le tour est joué. En fait, vous pouvez même supprimer la déclaration explicite de doneAction dans ce cas et vous contenter de .kr(2). Le .kr indique à SC d'"exécuter" cette enveloppe à la vitesse de contrôle (comme d'autres signaux de vitesse de contrôle que nous avons vus auparavant).

38.2 Env.triangle

Env.triangle ne prend que deux arguments : la durée et le niveau. Exemples :

```
// Voir le site :
Env.triangle.plot ;
// Entendez-le :
{SinOsc.ar([440, 442], mul : Env.triangle.kr(2))}.play ;
// D'ailleurs, une enveloppe peut être un multiplicateur n'importe où dans votre code
{SinOsc.ar([440, 442]) * Env.triangle.kr(2)}.play ;
```

38.3 Env.lin

Env.linen décrit une enveloppe de ligne avec attaque, portion de sustain et relâchement. Vous pouvez également spécifier le niveau et le type de courbe. Exemple :

```
// Voir le site :
Env.linen.plot ;
// Entendez-le :
{SinOsc.ar([300, 350], mul : Env.linen(0.01, 2, 1, 0.2).kr(2))}.play ;
```

38.4 Env.paires

Besoin de plus de flexibilité ? Avec Env.pairs, vous pouvez obtenir des enveloppes de la forme et de la durée de votre choix. Env.pairs prend deux arguments : un tableau de paires [time, level] et un type de courbe (voir le fichier d'aide Env pour connaître tous les types de courbes disponibles).

```
(
{
        var env = Env.pairs([[0, 0], [0.4, 1], [1, 0.2], [1.1, 0.5], [2, 0]], \lin)
        ;
        env.plot ;
        SinOsc.ar([440, 442], mul : env.kr(2)) ;
}.play ;
)
```

Lisez le tableau de paires comme suit :

Au temps 0, être au niveau 0 ; Au temps 0,4, être au niveau 1 ; Au temps 1, être au niveau 0,2 ; Au temps 1,1, être au niveau 0,5 ;
Au moment 2, être au niveau 0.

38.4.1 Les enveloppes - pas seulement pour l'amplitude

Rien ne vous empêche d'utiliser ces mêmes formes pour contrôler autre chose que l'amplitude. Il suffit de les mettre à l'échelle de la plage de nombres souhaitée. Par exemple, vous pouvez créer une enveloppe pour contrôler la variation des fréquences dans le temps :

```
(
{
```

2

```
        var freqEnv = Env.pairs([[0, 100], [0.4, 1000], [0.9, 400], [1.1, 555], [2,
            440]], \lin) ;
        SinOsc.ar(freqEnv.kr, mul : 0.2) ;
}.play ;
)
```

Les enveloppes sont un moyen puissant de contrôler les paramètres d'un synthétiseur qui doivent changer au fil du temps.

38.5 Enveloppe ADSR

Toutes les enveloppes vues jusqu'à présent ont un point commun : elles ont une durée fixe et prédéfinie. Il existe cependant des situations où ce type d'enveloppe n'est pas adéquat. Imaginez par exemple que vous jouez sur un clavier MIDI. L'*attaque de* la note est déclenchée lorsque vous appuyez sur une touche. Le *relâchement* est déclenché lorsque vous retirez votre doigt de la touche. Mais la durée pendant laquelle vous maintenez le doigt enfoncé n'est pas connue à l'avance. Ce dont nous avons besoin dans ce cas, c'est d'une "enveloppe soutenue". En d'autres termes, après la partie attaque, l'enveloppe doit maintenir la note pendant une durée indéterminée et ne déclencher la partie relâchement qu'après une sorte de repère ou de message, c'est-à-dire au moment où vous "relâchez la touche".

Une enveloppe ASR (Attack, Sustain, Release) convient parfaitement. Une variante plus populaire est l'enveloppe ADSR (Attack, Decay, Sustain, Release). Examinons les deux.

```
// ASR
// Jouer la note ("appuyer sur la touche")
// attackTime : 0.5 seconds, sustainLevel : 0.8, releaseTime : 3 seconds
x = {arg gate = 1, freq = 440 ; SinOsc.ar(freq : freq, mul : Env.asr(0.5, 0.8,
    3).kr( doneAction : 2, gate : gate))}.play ;
// Note d'arrêt ("doigt en dehors de la touche" -activation de la phase de
relâchement)
x.set(\gate, 0) ; // alternativement, x.release

// ADSR (attack, decay, sustain, release)
```

8

```
// Jouer la note :
(
d = {arg gate = 1 ;
        var snd, env ;
        env = Env.adsr(0.01, 0.4, 0.7, 2) ;
        snd = Splay.ar(BPF.ar(Saw.ar((32.1, 32.2..33)), LFNoise2.kr(12).range(100,
            1000), 0.05, 10)) ;
        Out.ar(0, snd * env.kr(doneAction : 2, gate : gate)) ;
}.play ;
)
// Note d'arrêt :
d.release ; // ceci est équivalent à d.set(\gate, 0) ;
```

Concepts clés :

Attaque Temps (en secondes) nécessaire pour passer de zéro (silence) à l'amplitude maximale.

Décroissance Temps (en secondes) nécessaire pour passer de l'amplitude maximale à l'amplitude de maintien.

Sustain L'amplitude (entre 0 et 1) à laquelle la note doit être maintenue (important : cela n'a rien à voir avec le temps).

Release Temps (en secondes**)** nécessaire pour passer du niveau de maintien à zéro (silence).

Comme les enveloppes soutenues n'ont pas une durée totale connue à l'avance, elles ont besoin d'une notification pour savoir quand commencer (déclencher l'attaque) et quand s'arrêter (déclencher le relâchement). Cette notification est appelée "*gate*". Le gate est ce qui indique à l'enveloppe de s'ouvrir (1) et de se fermer (0), démarrant et arrêtant ainsi la note.

Pour qu'une enveloppe ASR ou ADSR fonctionne dans votre synthé, vous devez déclarer un argument gate. Normalement, la valeur par défaut est gate = 1 parce que vous voulez que

le synthé commence à jouer tout de suite. Lorsque

si vous voulez que le synthé s'arrête, envoyez simplement le message .release ou .set(\gate, 0) : la partie release de l'enveloppe sera alors déclenchée. Par exemple, si votre temps de relâchement est de 3, la note mettra trois secondes à disparaître à *partir du moment où vous envoyez le message* .set(\gate, 0).

38.6 EnvGen

Pour mémoire, vous devez savoir que la construction que vous avez apprise dans cette section pour générer des enveloppes est un raccourci, comme le montre le code ci-dessous.

```
// Ceci :
{ SinOsc.ar * Env.perc.kr(doneAction : 2) }.play ;
// ... est un raccourci pour cela :
{ SinOsc.ar * EnvGen.kr(Env.perc, doneAction : 2) }.play ;
```

EnvGen est l'UGen qui lit les enveloppes de points d'arrêt définies par Env. Pour des raisons pratiques, vous pouvez continuer à utiliser le raccourci. Cependant, il est utile de savoir que ces notations sont équivalentes, car vous verrez souvent EnvGen utilisé dans les fichiers d'aide ou dans d'autres exemples en ligne.

39 Définitions des synthétiseurs

Jusqu'à présent, nous avons *défini des* synthés de manière transparente et les avons *joués* immédiatement. De plus, le message .set nous a donné une certaine flexibilité pour modifier les contrôles des synthés en temps réel. Cependant, dans certaines situations, vous pouvez vouloir définir vos synthés en premier (sans les jouer immédiatement), et ne les jouer que plus tard. En substance, cela signifie que nous devons séparer le moment de l'écriture de la recette (la définition du synthé) du moment de la cuisson du gâteau (la création du son).

39.1 SynthDef et Synth

SynthDef est ce que nous utilisons pour "écrire la recette" d'un synthé. Ensuite, vous pouvez le jouer avec Synth. Voici un exemple simple.

```
// Définition du synthé avec l'objet SynthDef
SynthDef("mySine1", {Out.ar(0, SinOsc.ar(770, 0, 0.1))}).add ;
// Jouer une note avec l'objet
Synth x = Synth("mySine1") ;
x.gratuit ;

// Un exemple un peu plus souple utilisant des arguments
// et une enveloppe auto-terminante (doneAction : 2)
SynthDef("mySine2", {arg freq = 440, amp = 0.1 ;
        var env = Env.perc(level : amp).kr(2)
                ;
        var snd = SinOsc.ar(freq, 0, env) ;
        Out.ar(0, snd) ;
}).add ;

Synth("mySine2") ; // utilisation des valeurs
par défaut
Synth("mySine2", [\freq, 770, \amp, 0.2]) ;
Synth("mySine2", [\freq, 415, \amp, 0.1]) ;
Synth("mySine2", [\freq, 346, \amp, 0.3]) ;
Synth("mySine2", [\freq, rrand(440, 880)]) ;
```

Le premier argument de SynthDef est un nom défini par l'utilisateur pour le synthé. Le second argument est une fonction dans laquelle vous spécifiez le graphe UGen (c'est ainsi que votre combinaison d'UGens est appelée). Notez que vous devez explicitement utiliser Out.ar pour indiquer à quel bus vous voulez envoyer le signal. Enfin, SynthDef reçoit le message .add à la fin, ce qui signifie que vous l'ajoutez à la collection de synthés que SC connaît. Ceci sera valable jusqu'à ce que vous quittiez SuperCollider.

Après avoir créé une ou plusieurs définitions de synthé avec SynthDef, vous pouvez les jouer avec Synth : le premier argument est le nom de la définition de synthé que vous voulez utiliser, et le second argument (optionnel) est un tableau avec tous les paramètres que vous souhaitez spécifier (freq, amp, etc.).

39.2 Exemple

Voici un exemple plus long. Après avoir ajouté le SynthDef, nous utilisons un tableau pour créer un accord de 6 notes avec des hauteurs et des amplitudes aléatoires. Chaque synthé est stocké dans l'un des emplacements du tableau, de sorte que nous pouvons les libérer indépendamment.

```
// Créer SynthDef
(
SynthDef("wow", {arg freq = 60, amp = 0.1, gate = 1, wowrelease = 3 ;
        var chorus, source, filtermod, env, snd ;
        chorus = Lag.kr(freq, 2) * LFNoise2.kr([0.4, 0.5, 0.7, 1, 2, 5, 10]).range
            (1, 1.02) ;
        source = LFSaw.ar(chorus) * 0.5 ;
        filtermod = SinOsc.kr(1/16).range(1, 10)
        ;
        env = Env.asr(1, amp, wowrelease).kr(2, gate) ;
        snd = LPF.ar(in : source, freq : freq * filtermod, mul : env) ;
Out.ar(0, Splay.ar(snd))
}).add ;
)

// Observer l'arbre des nœuds
s.plotTree ;

// Créer un accord de 6 notes
a = Array.fill(6, {Synth("wow", [\freq, rrand(40, 70).midicps, \amp, rrand(0.1, 0.5)
    ])}) ; // le tout en une seule ligne
```

```
// Libérer les notes une à une
a[0].set(\gate, 0) ;
a[1].set(\gate, 0) ;
a[2].set(\gate, 0) ;
a[3].set(\gate, 0) ;
a[4].set(\gate, 0) ;
a[5].set(\gate, 0) ;

// AVANCÉ : exécuter à nouveau l'accord de 6 notes, puis évaluer cette ligne.
// Pouvez-vous comprendre ce qui se passe ?
SystemClock.sched(0, {a[5.rand].set(\freq, rrand(40, 70).midicps) ; rrand(3, 10)}) ;
```

Pour vous aider à comprendre la SynthDef ci-dessus :

- Le son obtenu est la somme de sept oscillateurs en dents de scie étroitement accordés passant par un filtre passe-bas.
- Ces sept oscillateurs sont créés grâce à l'expansion multicanal.
- Qu'est-ce que le chorus variable ? C'est la fréquence freq multipliée par un LFNoise2.kr. L'expansion multicanal commence ici, parce qu'un tableau de 7 éléments est donné en argument à LFNoise2. Le résultat est que sept copies de LFNoise2 sont créées, chacune fonctionnant à une vitesse différente tirée de la liste [0.4, 0.5, 0.7, 1, 2, 5, 10]. Leurs sorties sont limitées à la plage de 1,0 à 1,02.
- Comme caractéristique supplémentaire, remarquez que freq est enfermé dans un Lag.kr. Chaque fois qu'une nouvelle valeur de fréquence est introduite dans ce Synth, l'UGen Lag crée simplement une rampe entre l'ancienne et la nouvelle valeur. Le "lag time" (durée de la rampe), dans ce cas, est de 2 secondes. C'est ce qui provoque l'effet de glissando que vous entendez après avoir exécuté la dernière ligne de l'exemple.

- Le son source LFSaw.ar prend la variable chorus comme fréquence. Dans un exemple concret : pour une valeur de freq de 60 Hz, la variable chorus résulterait en une expression comme

  60 * [1.01, 1.009, 1.0, 1.02, 1.015, 1.004, 1.019]

  dans lequel les nombres à l'intérieur de la liste changeraient constamment vers le haut et vers le bas en fonction des vitesses de chaque LFNoise2. Le résultat final est une liste de sept fréquences glissant toujours entre 60 et 61,2 (60 * 1,02). C'est ce qu'on appelle l'*effet chorus*, d'où le nom de la variable.

- Lorsque le chorus variable est utilisé comme fréquence de LFSaw.ar, l'expansion multicanal se produit : nous avons maintenant sept ondes en dents de scie avec des fréquences légèrement différentes.

- Le filtermod variable est juste un oscillateur sinusoïdal se déplaçant très lentement (1 cycle sur 16 secondes), avec sa plage de sortie échelonnée de 1 à 10. Elle sera utilisée pour moduler la fréquence de coupure du filtre passe-bas.

- La variable snd contient le filtre passe-bas (LPF), qui prend la source en entrée et filtre toutes les fréquences supérieures à sa fréquence de coupure. Cette fréquence de coupure n'est pas une valeur fixe : c'est l'expression freq * filtermod. Ainsi, dans l'exemple supposant que freq = 60, cela devient un nombre entre 60 et 600. Rappelez-vous que filtermod est un nombre oscillant entre 1 et 10, de sorte que la multiplication serait 60 * (1 à 10).

- Le LPF s'étend également en multicanal à sept copies. L'enveloppe d'amplitude env est également appliquée à cet endroit.

- Enfin, Splay prend cet ensemble de sept canaux et le mixe en stéréo.

39.3 Sous le capot

Ce processus en deux étapes de création d'un SynthDef (avec un nom unique) et d'appel d'un Synth est ce que SC fait tout le temps quand vous écrivez des instructions simples comme {SinOsc.ar}.play. SuperCollider décompose cela en (a) la création d'un SynthDef temporaire, et (b) la lecture immédiate de celui-ci (d'où les noms temp_01, temp_02 que vous voyez dans la fenêtre Post). Tout cela se passe dans les coulisses, pour votre confort.

```
// Lorsque vous faites cela :
{SinOsc.ar(440)}.play ;
// Voici ce que fait SC :
{Out.ar(0, SinOsc.ar(440))}.play ;
// Ce qui, à son tour, revient à dire ceci :
SynthDef("tempName", {Out.ar(0, SinOsc.ar(440))}).play ;

// Et tous sont des raccourcis pour cette opération en deux étapes :
SynthDef("tempName", {Out.ar(0, SinOsc.ar(440))}).add ; // créer une définition de
synthé Synth("tempName") ; // la jouer
```

40 Pbind peut jouer votre SynthDef

L'un des avantages de créer vos synthés en tant que SynthDefs est que vous pouvez utiliser Pbind pour les jouer.

En supposant que le SynthDef "wow" est toujours chargé en mémoire (il devrait l'être, à moins que vous n'ayez quitté et rouvert SC après le dernier exemple), essayez les Pbinds ci-dessous :

```
(
Pbind(
        \Ninstrument, "wow",
        \degree, Pwhite(-7, 7),
```

```
        \dur, Prand([0.125, 0.25], inf),
        \amp, Pwhite(0.5, 1),
        \n-owrelease, 1
) ;
)

(
Pbind(
        \Ninstrument, "wow",
        \scale, Pstutter(8, Pseq([
                Scale.lydienne,
                Scale.majeure,
                Scale.mixolydienne,
                Scale.mineure,
                Scale.phrygienne],
                inf)),
        \degree, Pseq([0, 1, 2, 3, 4, 5, 6, 7], inf),
        \dur, 0.2,
        \amp, Pwhite(0.5, 1),
        \n-owrelease, 4,
        \legato, 0.1
) ;
)
```

Lorsque vous utilisez Pbind pour jouer un de vos SynthDefs personnalisés, gardez à l'esprit les points suivants :

- Utilisez la touche Pbind \instrument pour déclarer le nom de votre SynthDef.
- Tous les arguments de votre SynthDef sont accessibles et contrôlables depuis Pbind : il suffit de les utiliser comme des clés Pbind. Par exemple, remarquez l'argument appelé \wowrelease utilisé ci-dessus. Ce n'est pas l'une des clés par défaut comprises par Pbind - au contraire, elle est unique à

la définition du synthé wow (le nom idiot a été choisi à dessein).

- Afin d'utiliser toutes les possibilités de conversion de hauteur de Pbind (les touches \degree, \note, et \midinote), assurez-vous que votre SynthDef a un argument d'entrée pour freq (il doit être orthographié exactement comme cela). Pbind fera le calcul pour vous.
- Si vous utilisez une enveloppe soutenue telle que Env.adsr, assurez-vous que votre synthé a un argument par défaut gate = 1 (gate doit s'écrire exactement comme cela, car Pbind l'utilise en coulisses pour arrêter les notes au bon moment).
- Si vous n'utilisez pas d'enveloppe soutenue, assurez-vous que votre SynthDef inclut une doneAction : 2 dans un UGen approprié, afin de libérer automatiquement les nœuds de synthèse dans le serveur.

Exercice : écrivez un ou plusieurs Pbinds pour jouer le SynthDef "pluck" fourni ci-dessous. Pour l'argument mutedString, essayez des valeurs entre 0.1 et 0.9. Demandez à l'un de vos Pbinds de jouer une séquence lente d'accords. Essayez d'arpéger les accords avec \strum.

```
(
SynthDef("pluck", {arg amp = 0.1, freq = 440, decay = 5, mutedString = 0.1 ;
var env, snd ;
env = Env.linen(0, decay, 0).kr(doneAction : 2) ;
snd = Pluck.ar(
        dans : WhiteNoise.ar(amp),
        trig : Impulsion.kr(0),
        maxdelaytime : 0.1,
        delaytime : freq.reciprocal,
        decaytime : decay,
        coef : mutedString) ;
```

```
12      Out.ar(0, [snd, snd]) ;
13 }).add ;
14 )
```

41 Bus de contrôle

Plus tôt dans ce tutoriel, nous avons parlé des bus audio (section 30) et de l'objet bus (section 33). Nous avons choisi de laisser de côté les bus de contrôle pour nous concentrer sur le concept de routage audio.

Les bus de contrôle de SuperCollider servent à acheminer les signaux de contrôle, pas l'audio. Hormis cette différence, il n'y a pas d'autre distinction pratique ou conceptuelle entre les bus audio et les bus de contrôle. Vous créez et gérez un bus de contrôle de la même manière que vous le faites avec les bus audio, en utilisant simplement .kr au lieu de .ar. SuperCollider dispose par défaut de 4096 bus de contrôle.

La première partie de l'exemple ci-dessous utilise un numéro de bus arbitraire à des fins de démonstration. La seconde partie utilise l'objet Bus, qui est la méthode recommandée pour créer des bus.

```
// Écriture d'un signal de contrôle dans le bus de contrôle 55
{Out.kr(55, LFNoise0.kr(1))}.play ;
// Lecture d'un signal de contrôle sur le bus 55
{In.kr(55).poll}.play ;

// Utilisation de l'objet Bus
~myControlBus = Bus.control(s, 1) ;
{Out.kr(~myControlBus, LFNoise0.kr(5).range(440, 880))}.play ;
{SinOsc.ar(freq : In.kr(~myControlBus))}.play ;
```

L'exemple suivant montre un signal de contrôle unique utilisé pour moduler deux synthés différents en même temps. Dans le synthé Blip, le signal de commande est redimensionné pour contrôler le nombre d'harmoniques entre 1 et 10. Dans le second synthé, le même signal de contrôle est redimensionné pour moduler la fréquence de l'oscillateur Pulse.

```
// Créer le bus de contrôle
~myControl = Bus.control(s, 1) ;
```

3

```
// Introduire le signal de contrôle dans le bus
c = {Out.kr(~myControl, Pulse.kr(freq : MouseX.kr(1, 10), mul : MouseY.kr(0, 1)))}.
    play ;

// Joue les sons contrôlés
// (déplacer la souris pour
entendre les changements) (
{
        Blip.ar(
                freq : LFNoise0.kr([1/2, 1/3]).range(50,
                60), numharm : In.kr(~myControl).range(1,
                10), mul : LFTri.kr([1/4, 1/6]).range(0,
                0.1))
}.play ;

{
        Splay.ar(
                Pulse.ar(
                        freq : LFNoise0.kr([1.4, 1, 1/2, 1/3]).range(100, 1000)
                        * In.kr(~myControl).range(0.9, 1.1),
                        mul : SinOsc.ar([1/3, 1/2, 1/4, 1/8]).range(0, 0.03))
        )
}.play ;
)

// Désactiver le signal de contrôle pour
la comparaison c.free ;
```

41.1 asMap

Dans l'exemple suivant, la méthode asMap est utilisée pour mapper directement un bus de contrôle à un nœud de synthé en cours d'exécution. De cette façon, vous n'avez même pas besoin de In.kr dans la définition du synthé.

```
// Créer un SynthDef
SynthDef("simple", {arg freq = 440 ; Out.ar(0, SinOsc.ar(freq, mul : 0.2))}).add ;
// Création de bus de contrôle
~oneBus = Bus.control(s, 1) ;
~anotherBus = Bus.control(s, 1) ;
// Commencer les contrôles
{Out.kr(~oneBus, LFSaw.kr(1).range(100, 1000))}.play ;
{Out.kr(~anotherBus, LFSaw.kr(2, mul : -1).range(500, 2000))}.play ;
// Commencer une note
x = Synth("simple", [\freq, 800]) ;
x.set(\freq, ~oneBus.asMap) ;
x.set(\freq, ~anotherBus.asMap) ;
x.free ;
```

42 Ordre d'exécution

Lorsque nous avons abordé la question des bus audio dans la section 30, nous avons fait allusion à l'importance de l'ordre d'exécution. Le code ci-dessous est une version étendue de l'exemple de bruit filtré de cette section. La discussion qui suit expliquera le concept de base de l'ordre d'exécution, en démontrant pourquoi il est important.

```
// Créer un bus audio
~fxBus = Bus.audio(s, 1) ;
~masterBus = Bus.audio(s, 1) ;
// Créer des
SynthDefs (
SynthDef("noise", {Out.ar(~fxBus, WhiteNoise.ar(0.5))}).add ;
SynthDef("filter", {Out.ar(~masterBus, BPF.ar(in : In.ar(~fxBus), freq : MouseY.kr
    (1000, 5000), rq : 0.1))}).add ;
```

```
8  SynthDef("masterOut", {arg amp = 1 ; Out.ar(0, In.ar(~masterBus) * Lag.kr(amp,
       1))}). add ;
9  )
10 // Ouvrir la fenêtre de l'arbre
11 des nœuds : s.plotTree ;
12 // Joue les synthés (observe
13 l'arbre des nœuds) m =
14 Synth("masterOut") ;
15 f = Synth("filter")
16 ; n = Synth("noise")
17 ;
// Volume principal
m.set("amp", 0.1);
```

Premièrement, deux bus audio sont assignés aux variables ~fxbus et ~masterBus. Ensuite, trois SynthDefs sont créés :

- "noise" est une source de bruit qui envoie du bruit blanc à un bus d'effets ;
- "filter" est un filtre passe-bande qui prend son entrée dans le bus d'effets et envoie le son traité dans le bus maître ;
- "masterOut" reçoit le signal du bus maître et lui applique un simple contrôle de volume, envoyant le son final avec un volume ajusté aux haut-parleurs.

Observez l'arborescence des nœuds pendant que vous exécutez les synthétiseurs dans l'ordre.

Les nœuds de synthèse dans la fenêtre de l'arborescence des nœuds sont affichés de *haut en bas*. Les synthés les plus récents sont ajoutés au sommet par défaut. Dans la figure 9, vous pouvez voir que "noise" est en haut, "filter" en deuxième et "masterOut" en dernier. C'est le bon ordre que nous voulons : en lisant de haut en bas, la source de bruit passe dans le filtre, et le résultat du filtre passe dans le bus maître. Si vous essayez à nouveau d'exécuter l'exemple, mais en évaluant les lignes m, f et n dans l'ordre inverse, vous n'entendrez rien, car les signaux sont calculés dans le mauvais ordre.

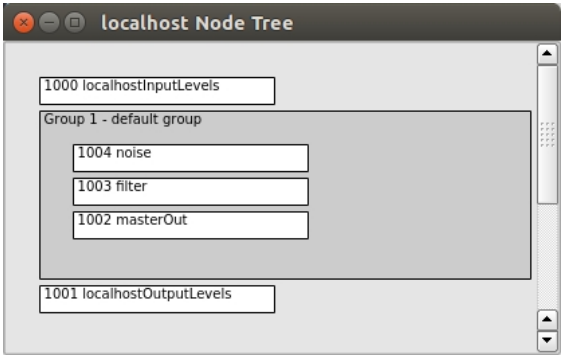

Figure 9 : Nœuds Synth dans la fenêtre Arborescence des nœuds

Évaluer les bonnes lignes dans le bon ordre est une bonne chose, mais cela peut s'avérer délicat lorsque votre code devient plus complexe. Afin de faciliter ce travail, SuperCollider vous permet de définir explicitement où placer les synthétiseurs dans l'arbre des nœuds. Pour cela, nous utilisons les arguments target et addAction.

```
1 n = Synth("noise", addAction : 'addToHead') ;
2 m = Synth("masterOut", addAction : 'addToTail') ;
3 f = Synth("filter", target : n, addAction : 'addAfter') ;
```

Maintenant, quel que soit l'ordre dans lequel vous exécutez les lignes ci-dessus, vous pouvez être sûr que les nœuds tomberont au bon endroit. Le synthé "noise" est explicitement ajouté à la tête de l'arbre de nœuds ; "masterOut" est ajouté à la queue ; et le filtre est explicitement ajouté juste après la cible n (le synthé noise).

42.1 Groupes

Lorsque vous commencez à avoir beaucoup de synthétiseurs - certains pour les sons sources, d'autres pour les effets, ou tout ce dont vous avez besoin - il peut être judicieux de les organiser en groupes. Voici un exemple de base :

```
// Continue à observer tout ce qui se trouve
dans le NodeTree s.plotTree ;

// Créer des bus
~reverbBus = Bus.audio(s, 2) ;
~masterBus = Bus.audio(s, 2) ;

// Définir des
groupes (
~sources = Group.new ;
~effects = Group.new(~sources, \addAfter) ;
~master = Group.new(~effects, \addAfter) ;
)

// Lancer tous les
synthés en même temps (
// Son d'une seule source
{
  Out.ar(~reverbBus, SinOsc.ar([800, 890])*LFPulse.ar(2)*0.1)
}.play(target : ~sources) ;

// Un autre son de source
{
  Out.ar(~reverbBus, WhiteNoise.ar(LFPulse.ar(2, 1/2, width : 0.05)*0.1))
}.play(target : ~sources) ;

// Un peu de réverbération
```

```
{
  Out.ar(~masterBus, FreeVerb.ar(In.ar(~reverbBus, 2), mix : 0.5, room : 0.9))
}.play(target : ~effects) ;

// Contrôle du volume principal à l'aide de la souris
{
  Out.ar(0, In.ar(~masterBus, 2) * MouseY.kr(0, 1))
}.play(target : ~master) ;
)
```

Pour plus d'informations sur l'ordre d'exécution, consultez les fichiers d'aide "Synth", "Ordre d'exécution" et "Groupe".

Partie V
QUELLE EST LA PROCHAINE ÉTAPE ?

Si vous avez lu et plus ou moins compris tout ce qui a été dit dans ce tutoriel jusqu'à présent, vous n'êtes plus un utilisateur débutant de SuperCollider ! Nous avons couvert beaucoup de terrain, et à partir de maintenant vous avez tous les outils de base nécessaires pour commencer à développer vos projets personnels, et continuer à apprendre par vous-même. Les sections suivantes fournissent une brève introduction à quelques sujets populaires de niveau intermédiaire. La toute dernière section présente une liste concise d'autres tutoriels et ressources d'apprentissage.

43 MIDI

Une présentation détaillée des concepts et astuces MIDI dépasse le cadre de ce didacticiel. Les exemples ci-dessous supposent une certaine familiarité avec les appareils MIDI et sont fournis uniquement pour vous aider à démarrer.

```
1  // Moyen rapide de connecter tous les appareils disponibles à SC
2  MIDIIn.connectAll ;
3
4  // Moyen rapide de voir tous les messages MIDI entrants
5  MIDIFunc.trace(true) ;
6  MIDIFunc.trace(false) ; // l'arrêter
7
8  // Moyen rapide d'inspecter toutes les entrées CC
9  MIDIdef.cc(\someCC, {arg a, b ; [a, b].postln}) ;
10
11 // L'entrée ne provient que de cc 7, canal 0
12 MIDIdef.cc(\someSpecificControl, {arg a, b ; [a, b].postln}, ccNum : 7, chan : 0) ;
13
```

```
// Un SynthDef pour des tests rapides
SynthDef("quick", {arg freq, amp ; Out.ar(0, SinOsc.ar(freq) * Env.perc(level :
am
    kr(2))}).add ;                                                              p).

// Jouer à partir d'un clavier ou d'une batterie
(
MIDIdef.noteOn(\someKeyboard, { arg vel, note ;
        Synth("quick", [\freq, note.midicps, \amp, vel.linlin(0, 127, 0, 1)])
                                                                                ;
}) ;
)

// Créer un motif et le jouer au clavier
(
a = Pbind(
        \instrument, "quick",
        \degree, Pwhite(0, 10, 5),
        \amp, Pwhite(0.05, 0.2),
        \dur, 0.1
) ;
)

// test
a.jouer ;

// Déclenchement d'un motif à partir d'un pad ou
d'un clavier
MIDIdef.noteOn(\quneo, {arg vel, note ; a.play})
```

; Une question fréquemment posée est de savoir comment gérer les messages d'activation et de désactivation de la note pour les notes soutenues. En d'autres termes, lorsque vous utilisez une enveloppe ADSR, vous souhaitez que chaque note soit soutenue tant qu'une touche est enfoncée. La phase de relâchement n'intervient que lorsque le doigt quitte la touche correspondante (revoir

la section sur les enveloppes ADSR si nécessaire).

Pour ce faire, SuperCollider doit simplement garder la trace du nœud de synthé correspondant à chaque touche. Nous pouvons utiliser un tableau à cette fin, comme le montre l'exemple ci-dessous.

```
 // Un SynthDef avec enveloppe ADSR
SynthDef("quick2", {arg freq = 440, amp = 0.1, gate = 1 ;
        var snd, env ;
        env = Env.adsr(0.01, 0.1, 0.3, 2, amp).kr(2, gate) ;
        snd = Saw.ar([freq, freq*1.5], env) ;
        Out.ar(0, snd)
}).add ;

// Jouez-le avec un clavier
MIDI (
var noteArray = Array.newClear(128) ; // le tableau comporte un emplacement par note MIDI possible

MIDIdef.noteOn(\myKeyDown, {arg vel, note ;
        noteArray[note] = Synth("quick2", [\freq, note.midicps, \amp, vel.linlin(0, 127, 0, 1)]) ;
        ["NOTE ON", note].postln ;
}) ;

MIDIdef.noteOff(\myKeyUp, {arg vel, note
        ; noteArray[note].set(\gate, 0)
        ; ["NOTE OFF", note].postln ;
}) ;
)
// PS. Assurez-vous que les connexions SC MIDI sont effectuées (MIDIIn.connectAll)
```

Pour vous aider à comprendre le code ci-dessus :

- Le SynthDef "quick2" utilise une enveloppe ADSR. L'argument gate est responsable de l'activation et de la désactivation des notes.

- Un tableau appelé "noteArray" est créé pour assurer le suivi des notes jouées. Les indices du tableau sont censés correspondre aux numéros de note MIDI joués.
- Chaque fois qu'une touche est pressée sur le clavier, un synthé commence à jouer (un nœud de synthé est créé dans le serveur), et *la référence à ce nœud de synthé est stockée dans un emplacement unique du tableau* ; l'index du tableau est simplement le numéro de la note MIDI elle-même.
- Chaque fois qu'une touche est relâchée, le message .set(\gate, 0) est envoyé au nœud de synthétiseur approprié, extrait du tableau par numéro de note.

Dans cette courte démonstration MIDI, nous n'avons parlé que de l'*entrée de messages* MIDI *dans* SuperCollider. Pour obtenir des messages MIDI à *partir* de SuperCollider, consultez le fichier d'aide MIDIOut.

44 OSC

OSC (Open Sound Control) est un excellent moyen de communiquer tout type de message entre différentes applications ou différents ordinateurs sur un réseau. Dans de nombreux cas, il s'agit d'une alternative beaucoup plus souple aux messages MIDI. Nous n'avons pas la place de l'expliquer plus en détail ici, mais l'exemple ci-dessous devrait servir de bon point de départ.

L'objectif de la démo est d'envoyer des messages OSC d'un smartphone à votre ordinateur, ou d'un ordinateur à un autre.

Sur l'ordinateur récepteur, évaluez ce simple bout de code :

```
1 (
2 OSCdef(
3         clé : \Peu importe,
          func : {arg ...args ; args.postln},
```

4

```
        path : '/stuff')
)
```

Note : en appuyant sur [ctrl+.], vous interromprez l'OSCdef et vous ne recevrez plus de messages.

44.1 Envoi d'OSC à partir d'un autre ordinateur

Cela suppose que les deux ordinateurs utilisent SuperCollider et sont connectés à un réseau. Trouvez l'adresse IP de l'ordinateur récepteur et évaluez les lignes suivantes dans l'ordinateur expéditeur :

```
// A utiliser sur la machine qui envoie les messages
~destination = NetAddr("127.0.0.1", 57120) ; // utiliser l'adresse IP correcte de
    l'ordinateur de destination

~destination.sendMsg("/stuff", "heelloooo") ;
```

44.2 Envoi d'OSC à partir d'un smartphone

- Installez une application OSC gratuite sur le téléphone (par exemple, gyroscope) ;
- Entrez l'adresse IP de l'ordinateur récepteur dans l'application OSC (en tant que "cible") ;
- Entrez le port de réception de SuperCollider dans l'application OSC (généralement 57120) ;
- Vérifiez le chemin exact du message que l'application utilise pour envoyer l'OSC, et modifiez votre OSCdef en conséquence ;
- Assurez-vous que votre téléphone est connecté au réseau

Si votre téléphone envoie les messages au bon chemin, vous devriez les voir arriver sur l'ordinateur.

45 Quarks et plug-ins

Vous pouvez étendre les fonctionnalités de SuperCollider en ajoutant des classes et des UGens créés par d'autres utilisateurs. Les *quarks* sont des paquets de classes SuperCollider, qui étendent ce que vous pouvez faire dans le langage SuperCollider. Les *plugins UGen* sont des extensions pour le serveur de synthèse audio SuperCollider.

Veuillez consulter le site http://supercollider.sourceforge.net/ pour obtenir des informations actualisées sur la manière d'ajouter des plug-ins et des quarks à votre installation de SuperCollider. Le fichier d'aide "Using Quarks" est également un bon point de départ : http://doc.sccode.org/Guides/UsingQuarks.html. À partir de n'importe quel document SuperCollider, vous pouvez évaluer Quarks.gui pour voir une liste de tous les quarks disponibles (il s'ouvre dans une nouvelle fenêtre).

46 Ressources supplémentaires

C'est la fin de cette introduction à SuperCollider. Quelques ressources pédagogiques supplémentaires sont listées ci-dessous. Nous vous souhaitons beaucoup de plaisir !

- Une excellente série de tutoriels YouTube par Eli Fieldsteel : http://www.youtube.com/playlist?list=PLPYzvS8A_rTaNDweXe6PX4CXSGq4iEWYC.
- Le tutoriel standard de démarrage de SC par Scott Wilson et James Harkins, disponible en ligne et dans les fichiers d'aide intégrés : http://doc.sccode.org/Tutorials/Getting-Started/ 00-Getting-Started-With-SC.html
- Tutoriel en ligne de Nick Collins : http://composerprogrammer.com/teaching/supercollider/ sctutorial/tutorial.html

- La liste de diffusion officielle de SuperCollider est le meilleur moyen d'obtenir une aide amicale de la part d'un grand nombre d'utilisateurs. Les débutants sont les bienvenus pour poser des questions sur cette liste. Vous pouvez vous inscrire ici : http://www.birmingham.ac.uk/facilities/BEAST/research/supercollider/ mailinglist.aspx

- Trouvez un groupe de rencontre SuperCollider dans votre ville. La liste de diffusion officielle sc-users est le meilleur moyen de savoir s'il en existe un dans votre ville. S'il n'y a pas de groupe de rencontre dans votre région, créez-en un !

- De nombreux extraits de code intéressants peuvent être trouvés ici : http://sccode.org/. Créez un compte et partagez votre code.

- Avez-vous entendu parler des tweets de SuperCollider ? http://supercollider.github.io/community/ sc140.html

Notes

[1]Première question : lorsque vous utilisez le nombre 1 au lieu de `inf` comme argument de `répétition` du deuxième `Pseq`, le Pbind s'arrête après que 6 notes ont été jouées (c'est-à-dire après qu'une séquence complète de valeurs de durée a été exécutée). Deuxième question : pour qu'un Pbind soit joué indéfiniment, il suffit d'utiliser `inf` comme valeur de `répétition` de tous les motifs internes.

[2]

a) `Pwhite(0, 10)` génère n'importe quel nombre entre 0 et 10. `Prand([0, 4, 1, 5, 9, 10, 2, 3], inf)` ne choisira que dans la liste, qui contient *certains* nombres entre 0 et 10, mais pas tous (6, 7, 8 ne sont pas là, donc ils n'apparaîtront jamais dans ce `Prand`).

b) Techniquement, vous pourriez utiliser un `Prand` si vous fournissiez une liste de tous les nombres entre 0 et 100, mais il est plus logique d'utiliser un `Pwhite` pour cette tâche : `Pwhite(0, 100)`.

c) `Prand([0, 1, 2, 3], inf)` choisit des éléments de la liste au hasard. `Pwhite(0, 3)` aboutit au même type de par des moyens différents : il générera des nombres entiers aléatoires entre 0 et 3, ce qui revient au même ensemble d'options que le `Prand` ci-dessus. Cependant, si vous écrivez `Pwhite(0, 3.0)`, la sortie est maintenant différente : parce que l'un des arguments d'entrée de `Pwhite` est écrit sous la forme d'un flottant (3.0), il produira maintenant n'importe quel nombre à virgule flottante entre 1 et 3, comme 0.154, 1.0, 1.45, 2.999.

d) Le premier `Pbind` joue 32 notes (4 fois la séquence de 8 notes). Le deuxième `Pbind` ne joue que 4 notes : quatre choix aléatoires tirés de la liste (rappelez-vous que `Prand`, contrairement à `Pseq`, n'a aucune obligation de jouer toutes les notes de la liste : il choisira simplement autant de notes aléatoires que vous lui indiquerez). Le troisième et dernier `Pbind` joue 32 notes, comme le premier.

[3]Première ligne : le tableau `[1, 2, 3, "wow"]` est l'objet récepteur ; `reverse` est le message. Deuxième ligne : la chaîne "hello" est l'objet récepteur ; `dup` est le message ; `4` est l'argument de `dup`. Troisième ligne : 3.1415 est l'objet récepteur ; `round` est le message ; `0.1` est l'argument de `round`. Quatrième ligne : `100` est l'objet récepteur, `rand` est le message. Dernière ligne : `100.0` est le récepteur du message `rand`, dont le résultat est un nombre aléatoire compris entre 0 et 100. Ce nombre devient le récepteur du message `round` avec l'argument `0.01`, de sorte que le nombre aléatoire est arrondi à deux décimales. Ensuite, ce résultat devient l'objet récepteur du message `dup` avec l'argument `4`, qui crée une liste avec quatre doublons de ce nombre.

[4]Réécriture en utilisant uniquement la notation fonctionnelle : `dup(round(rand(100.0), 0.01), 4) ;`

[5] Réponses :

a) 24

b) [5, 5.123] (chiffres et parenthèses)

c) Toute la ligne `LFSaw`

d) Un seul

e) 0.4

f) 1 et 0,3

[6] `SinOsc` est bipolaire parce qu'il émet des nombres entre -1 et +1. `LFPulse` est unipolaire parce que sa plage de sortie est de 0 à 1 (en fait, `LFPulse` en particulier ne produit que des zéros ou des uns, rien entre les deux).

[7] Solution : `a = {Out.ar(0, SinOsc.ar(freq : [800, 880], mul : LFPulse.ar([2, 3])))}.play ;`

[8] (a) La variable `lfn` contient simplement un `LFNoise2`. Le rôle de `LFNoise2` dans la vie est de générer un nouveau nombre aléatoire toutes les secondes (entre -1 et +1), et de glisser jusqu'à lui à partir du nombre aléatoire précédent (contrairement à `LFNoise0`, qui saute au nouveau nombre immédiatement). La première utilisation de cette variable `lfn` est dans l'argument `freq` du BPF : `lfn.range(500, 2500)`. Cette variable prend les nombres compris entre -1 et +1 et les met à l'échelle de la plage 500-2500. Ces nombres sont ensuite utilisés comme fréquence centrale du filtre. Ces fréquences sont les hauteurs que nous entendons glisser vers le haut et vers le bas. Enfin, `lfn` est à nouveau utilisé pour contrôler la position du panner `Pan2`. Il est utilisé directement (sans message `.range`) parce que les nombres sont déjà dans la plage que nous voulons (-1 à +1). L'avantage est que nous couplons le changement de fréquence avec le changement de position. Comment ? Toutes les secondes, `LFNoise2` commence à glisser vers un nouveau nombre aléatoire, et cela devient un changement synchronisé de la fréquence du filtre et de la position du panoramique. Si nous avions deux `LFNoise2` différents à chaque endroit, les changements ne seraient pas corrélés (ce qui pourrait être bien aussi, mais c'est un résultat sonore différent).
(b) un `mul :` de 1 serait tout simplement trop doux. Parce que le filtre est si pointu, il enlève tellement du signal original que l'amplitude diminue `trop`. Nous devons renforcer le signal `pour` qu'il redevienne raisonnablement audible, `c'est` pourquoi nous avons un `mul : 20` à la fin de la ligne `BPF.`
(c) Le rythme est piloté par le `LFPulse` qui est l'argument `mul :` du `Saw`. La fréquence du `LFPulse` (nombre d'impulsions par seconde) est contrôlée par un `LFNoise1` qui produit des nombres entre 1 et 10 (interpolation entre
). Ces nombres correspondent au "nombre de notes par seconde" de ce patch.