(*) Comment bien démarrer avec SuperCollider ?

> (*) : Il manque des informations. Cette page est en cours de construction et n'est donc pas finalisée.

Avant toute chose, SuperCollider est un outil puissant, mais **qui nécessite des bases solides en programmation et en acoustique** pour être utilisé. Si vous ne savez pas encore déclarer une fonction, ou que vous ne savez pas pourquoi émettre deux sons simultanés à 440 et 447 HZ est un choix esthétique fort, je vous conseille de vous documenter sur ces deux sujets, et de commencer par composer sur Sonic Pi, qui est une excellente porte d'entrée à SuperCollider.

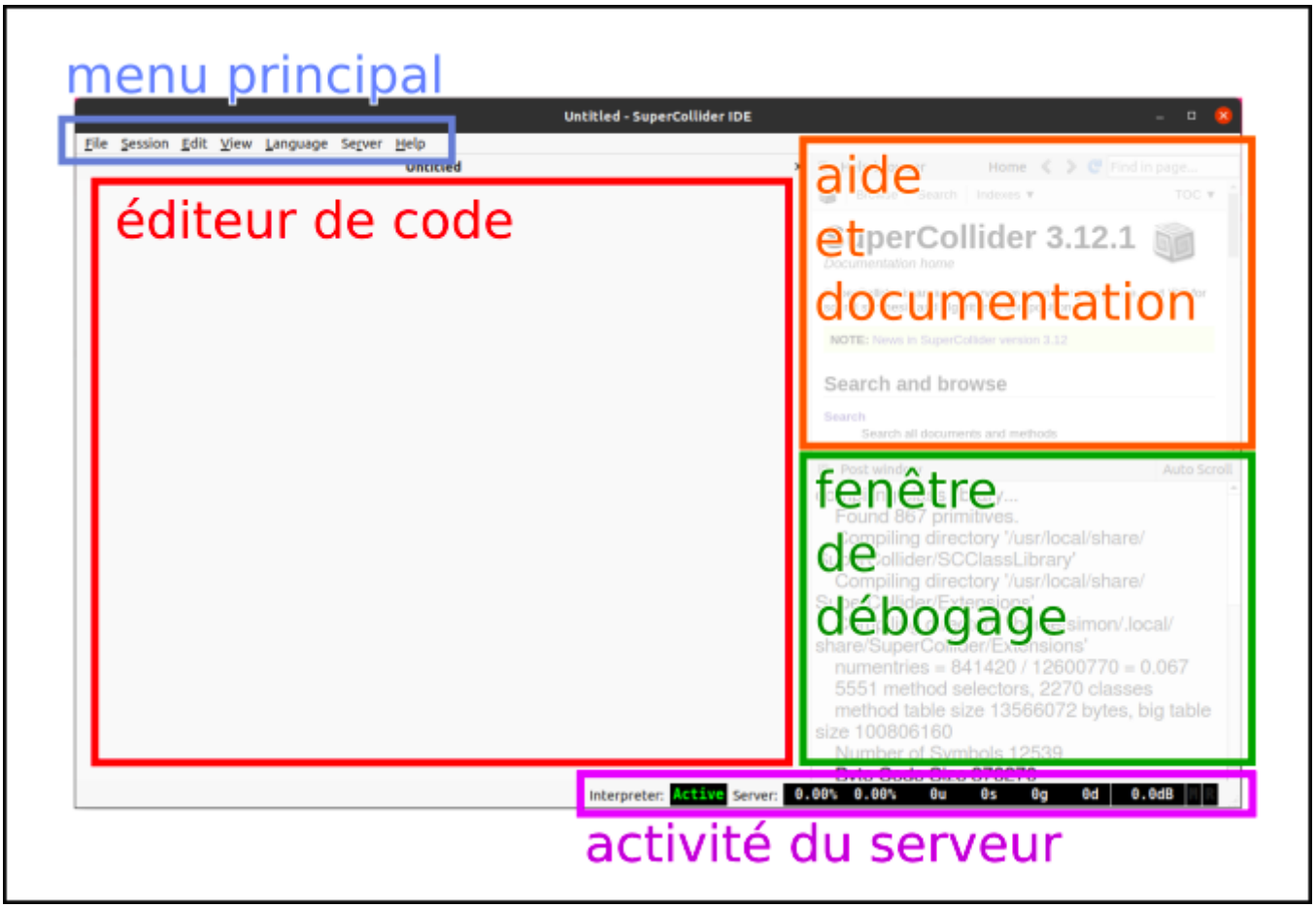
Si vous êtes d'attaque, il faut d'abord commencer par l'installer !

Pour les utilisateurs de Linux, vous pouvez le trouver depuis le gestionnaire de paquets :

```
sudo apt-get install scide
```

Lançons maintenant le logiciel !

Voici l'interface qui apparaît au lancement :



Avant de détailler les différents espaces, revenons sur un point assez important. **SuperCollider n'est pas un logiciel construit que d'un seul bloc.** Vous pouvez l'utiliser sans ouvrir la fenêtre ci-dessus. Pour simplifier, il y a **deux briques fondamentales** : le **serveur**, qu'on appelle communément *SuperCollider*, et l'**IDE**, ici *scide*, qui correspond à la fenêtre. Le serveur s'occupe de générer la musique, et l'IDE permet de piloter le serveur. Vous pourriez en fait piloter le serveur depuis n'importe quel langage de programmation, simplement en utilisant les bonnes commandes. De manière plus approfondie, le serveur s'appelle *scsynth* et le langage de l'IDE s'appelle *sclang*. Mais cela n'a pas d'importance pour débuter. Ce qu'il faut retenir pour l'instant, c'est surtout la présence de ce fameux **serveur**.

L'IDE de base, et le langage de programmation associé, est développé spécialement pour manipuler le serveur, ce qui en fait un environnement idéal pour débuter.

Voyons rapidement les différents espaces de l'IDE, qui apparaissent au lancement, et que l'on peut voir sur l'image ci-dessus :

- le *menu principal* vous permet les opérations communes : manipulation de fichiers, paramétrages, infos...
- l'*éditeur de code* vous permet... d'éditer du code !
- la *fenêtre d'aide et de documentation* intègre la documentation de SuperCollider.

- la *fenêtre de débogage* est un terminal qui affiche les messages internes, comme les erreurs, et les sorties que vous souhaitez afficher.
- la *fenêtre d'activité du serveur* indique un certain nombre d'information à propos du serveur.

Nous allons maintenant **cheminer tranquillement à travers les opérations basiques que vous pouvez effectuer avec l'IDE**. La plupart d'entre elles sont liées à **des raccourcis claviers**, que vous devrez sans doute mémoriser tant vous aurez à vous en servir.

Premièrement, vous serez souvent amené à **vous servir de la *fenêtre de débogage***, qui vous permettra d'obtenir des informations intéressantes sur ce qui se passe, notamment sur ce qui se passe mal...

Il y a déjà des informations qui y sont affichées : **lorsque l'IDE démarre, il vous donne des informations sur la manière dont il a chargé SuperCollider**. Par exemple la dernière ligne :

```
SCDoc: Indexed 1925 documents in 1.21 seconds
```

m'indique qu'il a trouvé 1925 pages de documentation, et qu'il les a chargées en 1 secondes 21.

Pour **remettre à zéro la *fenêtre de débogage***, on utilise la commande **CTRL + MAJ + P** . C'est pratique pour éviter que les informations s'accumulent !

Voyons maintenant **comment afficher des informations dans la fenêtre de débogage**.

Pour ce faire, il vous faudra vous servir de l'*éditeur de code*. Commençons par écrire un programme "Hello World", dont le but est simplement d'afficher "Hello World" dans la fenêtre de débogage. Voici la commande qui le permet, qu'il faudra taper dans l'*éditeur de code* :

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

Ensuite, pour lancer le code, il faudra utiliser la commande **CTRL + ENTRÉE** .

Si vous avez eu de la chance, le texte s'est mis temporairement en surbrillance, et "Hello World" s'est affiché deux fois dans la *fenêtre de* débogage. Sinon...

Il est nécessaire de donner quelques explications sur ce que nous venons de faire.

Premièrement, **le langage que nous utilisons est un langage dit "orienté objet"**. Cela indique qu'**il est constitué uniquement d'*objets* qui répondent à des *messages*.**

Dans notre exemple, l'objet "Hello World", qui est de type *String*, c'est-à-dire une suite de caractères, peut répondre au message "postln" en s'affichant dans la fenêtre de débogage.

À noter, deux syntaxes sont équivalentes dans SuperCollider pour envoyer un message à un objet :

`objet.message();`

`message( objet );`

Autrement dit, nous aurions également pu écrire

`postln( "Hello World" );`

Comme vous l'aurez remarqué, **on doit mettre des ; à la fin des phrases pour indiquer à SuperCollider la fin d'une commande**. Si cela fait sens pour vous, vous pouvez noter qu'il est possible d'omettre le dernier **;** d'un bloc.

Voyons maintenant pourquoi il est possible que "Hello World" ne se soit pas affiché lorsque que vous avez appuyé sur **CTRL + ENTRÉE**. Ce sera notamment le cas si vous êtes revenu à la ligne après la dernière commande.

Contrairement à d'autres langages de programmation, **SuperCollider ne va pas lire l'ensemble du code écrit dans l'éditeur** lorsque que vous utilisez **CTRL + ENTRÉE**. Par défaut, **il ne lira que le code situé sur la même ligne que votre curseur de texte.** C'est parce qu'il s'agît d'un langage dit interprété.

Considérons le code suivant :

```
"Je suis la première ligne".postln;
"Je suis la deuxième ligne".postln;
```

On pourrait s'attendre qu'en appuyant sur **CTRL + ENTRÉE** s'affichent successivement "Je suis la première ligne", puis "Je suis la deuxième ligne".

Il n'en est rien. Si votre curseur de texte est à la deuxième ligne, alors seule la deuxième ligne sera affichée dans *la fenêtre de débogage*.

Quel intérêt me direz-vous ? Eh bien dans le contexte musical, c'est assez pratique ma foi ! Nous pouvons avoir une ligne par instrument et les faire jouer indépendamment les uns des autres !

Mais **comment faire pour exécuter plusieurs lignes à la fois ?**

Deux solutions sont possibles : soit **sélectionner nos deux lignes et appuyer sur CTRL + ENTRÉE**, ce qui implique qu'elles soit adjacentes.

Soit, **nous utilisons des parenthèses pour regrouper du code en un seul bloc**. Dans ce cas là, **CTRL + ENTRÉE** exécutera l'ensemble du code entre parenthèse si le curseur de texte est à

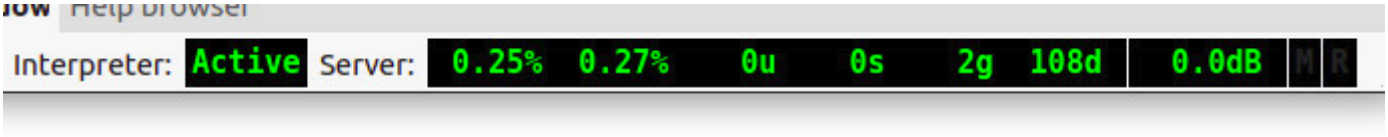
l'intérieur :

```
(
"Je suis la première ligne".postln;
"Je suis la deuxième ligne".postln;
)
```

Il est temps de faire du son ! Presque !

D'abord, il faut que **le serveur soit allumé** pour qu'il puisse produire du son. La commande qui permet de l'allumer est **CTRL + B** .

Cela est visible dans la fenêtre d'activité du serveur, qui affichera ses informations en vert lorsque le serveur est allumé :



Maintenant... **Quelques règles de sécurité. Comme tous les logiciels audio, l'utilisation de SuperCollider peut être dangereuse.** C'est principalement le cas lorsque que le volume est trop fort. Cela peut endommager votre matériel, et surtout, vos oreilles... Vous apprendrez avec le temps quels réglages sont les plus adaptés.

Pour l'instant, je vous conseille deux choses :

Premièrement, utiliser **CTRL + M** pour afficher les niveaux de sortie de SuperCollider. Cela permet d'**avoir un indice visuel de la puissance sonore, qui permet de vérifier si le son est trop fort sans risquer de s'abîmer les oreilles.**

Deuxièmement, tant que vous n'êtes pas à l'aise avec les outils de production de son de SuperCollider : **si vous n'êtes pas sûr du son qui va sortir de vos enceintes ou de votre casque : baissez le volume au maximum, c'est-à-dire jusqu'au silence. Lancez votre son. Puis augmentez graduellement le volume. Répétez pour chaque nouveau son.**

Maintenant que les règles de sécurité sont posées, voici donc enfin la première ligne de code qui va nous permettre de faire du son :

```
{ SinOsc.ar( 440, mul: 0.1 ) }.play
```

Pour **arrêter tous les son du serveur**, vous pouvez utiliser la commande clavier ***ctrl + maj + .*** .

Cette ligne de code est en fait le raccourci d'une méthode plus complexe. Nous allons voir ses composants dans le détail.

Dans SC, **les éléments à l'intérieur d'accolades { } sont une fonction**. La classe *Function* a une méthode particulière, *play*, qui permet de l'exécuter et d'en produire le son si elle contient des éléments musicaux.

Dans notre cas, nous avons en effet utilisé une des briques de bases du son dans SC : SinOsc.

SinOsc est la classe permettant de générer des signaux sonores sinusoïdaux.

Voici une version plus modulable de la ligne précédente :

```
(
{
	SinOsc.ar(
		freq: [ 440, 440 ],
		phase: 0,
		mul: 0.1,
		add: 0
	)
}.play
)
```

SinOsc prend en compte quatre paramètres : **la fréquence d'oscillation**, **le décalage initial de la phase** (normalement compris entre 0 et 2pi), **l'amplitude** (désignée comme *mul*), et un paramètre nommé *add* qui s'additionne à *mul* .

En comparant les deux exemples, vous noterez qu'il est possible d'omettre le nom des paramètres si ceux-ci sont spécifiées dans le bon ordre.

Dans ce cas particulier, j'ai utilisé une array de deux valeurs pour la fréquence. Lorsque que nous utilisons ce type d'argument, **SC créé automatiquement un nombre de canaux sonores égal à la taille de notre array** : ici, nous obtenons donc une sortie stéréo. Cette méthodologie, très utilisée dans SC, s'appelle *Multichannel Expansion*.

Dans notre cas, les paramètres de *phase* et d'*add* sont superflus, je les omettrai pour l'instant.

Dans SC, **tous les objets répondent aux méthodes mathématiques**, ce qui est extrêmement pratique pour la synthèse sonore :

```
(
{
SinOsc.ar( [ 440, 440 ], mul: 0.1 ) +
SinOsc.ar( [ 440, 440 ] * 5 / 4, mul: 0.1 )
}.play
)
```

Vous remarquerez qu'il est possible d'appliquer directement une opération mathématique à une array par ailleurs.

Pour ceux qui auraient remarqué le mystérieux *.ar* qui suit l'appel à la classe : nous y reviendrons.

Cette manière de faire est très pratique pour prototyper ou créer des drones, mais elle est loin d'exploiter la manière dont SC a été pensé.

C'est maintenant que nous allons voir comment se servir du serveur de SC.

L'idée est de créer une sorte de moule pour notre instrument, dont nous pourrons nous servir ensuite pour créer autant d'instruments que nous souhaitons.

Dans SC, cela correspond à créer une ***SynthDef*** :

```
(
SynthDef( \sineLabomedia, {
| out = 0, freq = 440, amp = 0.1 |

var snd = SinOsc.ar( freq, mul: amp );

Out.ar( out, [ snd, snd ] );

} ).add
)
```

Ok ! Premièrement, si vous êtes perdu, ***scide* vous permet d'accéder à la documentation du mot sur lequel est positionné le curseur d'écriture en utilisant le raccourci *ctrl* + *d*** . C'est notamment pratique pour se documenter sur les classes et comprendre leurs arguments, ou pour le débogage.

La *SynthDef* commence par le nom du synthé, dans un format particulier : il s'agit d'un *symbole*. Celui-ci se repère car il commence par un anti-slash (\ ) . Vous pouvez spécifier celui que vous souhaitez, mais il devrait en général correspondre à un nom clair pour l'instrument que vous

créez.

Ensuite vient la fonction qui correspond au son de notre instrument. Vous pouvez voir que j'ai indiqué des arguments à celle-ci : vous pouvez les omettre, mais c'est en général une bonne idée d'en avoir car ce sont ces arguments auxquels vous accéderez lorsque vous créerez votre instrument pour spécifier ses particularités.

Dans notre cas, cette fonction accomplit deux actions :

Une variable qui stocke un son est créée, à l'aide *SinOsc.*

Le son est poussé vers la sortie de la carte son grâce à *Out* . Vous pourrez noter qu'ici, je créé la stéréo non pas dans *SinOsc*, mais dans *Out*.

Enfin, **j'utilise la méthode *.add* sur ma *SynthDef***. C'est là l'élément important : en faisant cela, j'ajoute un nouveau modèle d'instrument au sein du serveur, que je pourrai réutiliser à l'envie *a posteriori*.

Une fois cela fait, je peux utiliser la classe *Synth* pour appeler mon synthé, en indiquant son nom :

```
Synth( \sineLabomedia )
```

Il utilisera les arguments par défaut que j'ai assigné, mais il m'est également possible d'en spécifier d'autres :

```
Synth( \sineLabomedia, [ freq: 660 ] )
```

Pour le modifier après l'avoir créé, il faudra **référencer l'instrument dans une variable**.

L'exemple commun est celui-ci, nous lançons le synthé et le référençons dans la variable *x* :

```
x = Synth( \sineLabomedia, [ freq: 440 * 3 / 2 ] )
```

Il est ensuite possible de modifier ses paramètres. Attention, il faut accéder aux arguments de la fonction en les indiquant en tant que symboles, c'est-à-dire en leur ajoutant un anti-slash :

```
x.set( \freq, 330, \amp, 0.05 )
```

Pour l'arrêter :

```
x.free
```

Il est étrange que nous puissions effectuer ces actions dans l'ordre de notre choix. C'est parce que la variable *x* est une variable *globale*.

Les *variables globales* restent en mémoire à la fin de l'exécution d'un bloc. Elles sont communes à tous les fichiers SC actuellement ouverts. Sans ce mécanisme, le code ne pourrait pas être interprété mais seulement exécuté.

Par défaut, **chacune des lettres de l'alphabet est une *variable globale* dans SuperCollider**, qui vous permet de référencer des objets constamment modifiables.

Je vous déconseille néanmoins l'usage des lettres de l'alphabet comme variables globales. Premièrement, car la variable *s* référence par défaut le serveur de SC, et qu'il est commun de réserver la variable *t* à un objet particulier. Ensuite parce qu'une lettre unique n'indique pas vraiment ce que la variable contient.

Pour créer vos *variables globales*, il suffira d'indiquer un nom précédé d'un ~ :

```
~monInstrument = Synth( \sineLabomedia, [ freq: 660 ] )
```

[EN CONSTRUCTION]

Révision #1
Créé 3 août 2023 14:31:54 par Rachelle
Mis à jour 3 août 2023 14:44:31 par Rachelle