CRÉATION DYNAMIQUE DE PLUGICIELS AVEC LE LANGAGE JS DE REAPER ET LA LIBRAIRIE COOKDSP

Olivier Bélanger
iACT (institut Arts Cultures et Technologies)
Université de Montréal, Faculté de Musique

RÉSUMÉ

Les compositeurs de musique électroacoustique sont régulièrement confrontés aux problématiques reliées à l'exploration du traitement de signal et au développement de modules d'effets originaux à l'intérieur d'une station audionumérique. Les plugiciels existant ne répondant pas toujours exactement aux besoins des compositeurs, il devrait être possible pour ceux-ci de créer leurs propres outils de manipulations sonores sans avoir à quitter le séquenceur pour travailler avec des logiciels externes. Les solutions présentées dans cet article visent principalement les créateurs sonores possédant déjà des notions de base en traitement de signal, sans pour autant être des programmeurs chevronnés. Après un tour d'horizon des alternatives disponibles, une revue détaillée du langage de script audio JS (Jesusonic), intégré au logiciel Reaper, sera présentée. Ensuite, nous introduirons la librairie CookDSP, développée par l'auteur pour le langage JS, qui est une collection de fonctions offrant accès à plusieurs algorithmes essentiels en traitement de signal. CookDSP constitue une extension du langage JS permettant de simplifier et d'accélérer le développement des plugiciels. Finalement, des exemples de plugiciels JSFX, créés avec la librairie CookDSP, seront présentés.

1. INTRODUCTION

Le flux de production du créateur sonore utilisant l'ordinateur comme instrument principal peut prendre plusieurs formes. Quantité d'outils de traitements audionumériques et de mise en place de la musique existent et le choix des uns ou des autres dans le travail de création dépend principalement des besoins de l'œuvre, ainsi que des habitudes et compétences du créateur. La nature de l'œuvre (composition pour support fixe, musique mixte interprétée en concert, musique d'application, etc.) favorise déjà certains environnements de travail plutôt que d'autres. Ainsi, une pièce destinée à être jouée en concert, utilisant des traitements en temps réel, sera généralement composée dans un environnement offrant les outils nécessaires à la création et au contrôle en direct de chaînes de traitements audio originales. Ces environnements, pour la plupart des langages de programmation (MaxMSP, Puredata, SuperCollider, Chuck, pyo), implique déjà, de la part du créateur, une connaissance de base de la programmation musicale et du traitement de signal. À l'intérieur de ces environnements, le compositeur peut créer des chaînes de traitements originales en toute liberté. Mais qu'en est-il des compositeurs utilisant principalement le séquenceur comme outil de mise en forme de la musique ? Le traitement du son à l'intérieur du séquenceur est limité aux plugiciels (gratuits ou payants) disponibles ainsi qu'aux contrôles que leurs concepteurs ont bien voulu rendre accessibles. Cet état de fait force généralement le compositeur soit à modifier son idée originale pour se plier aux contraintes des outils disponibles, soit à quitter momentanément la station audionumérique pour travailler ses sources sonores avec des outils dédiés au traitement de signal qui répondent mieux à ses besoins. Ces allers-retours entre les différents environnements ralentissent le travail et compliquent les manipulations sur les sources lorsque celles-ci doivent présenter une forte corrélation au mixage final. En effet, travailler une source sonore à l'aide d'un logiciel externe équivaut à travailler à l'aveuglette puisque l'on n'entend pas l'effet des manipulations sur l'ensemble des signaux en interaction. C'est pourquoi les compositeurs apprécient généralement les outils qui s'intègrent aisément à l'environnement de travail principal, c'est-à-dire le séquenceur. Quelles sont donc les possibilités qui s'offrent aux compositeurs désirant travailler dans un environnement unique, mais qui, sans nécessairement être des programmeurs expérimentés, possèdent suffisamment de connaissances en mathématique du traitement de signal pour être en mesure de créer leurs propres outils de traitement du son ? Dans cet article, les problématiques reliées au traitement de signal à l'intérieur même de la station audionumérique seront discutées. Ensuite, après un tour d'horizon des alternatives les plus communes, le langage JS [1], intégré au logiciel Reaper, et la librairie CookDSP seront détaillés. Finalement, des exemples de plugiciels utilisant les outils précédemment décrits seront présentés.

1 . Acronyme pour *Jesusonic*, à ne pas confondre avec le *JavaScript*.

2. PROBLÉMATIQUE

Les rôles principaux du séquenceur consistent à effectuer la mise en forme temporelle de la musique, c'est-à-dire placer avec précision les événements sonores dans le temps, ainsi qu'à faire la gestion de la polyphonie, c'est-à-dire le mixage des différentes sources en interaction. Plusieurs plugiciels existent sur le marché pour effectuer les opérations propres au *mastering* [2] (gestion de la dynamique, filtrage, spatialisation, etc.) mais lorsque l'on parle de traitement du son dans le but de créer de nouvelles sources sonores, l'éventail des possibilités de traitement est beaucoup trop vaste pour être entièrement couvert par les plugiciels existants. C'est principalement dans ce contexte que le compositeur a besoin d'un espace, à l'intérieur même du séquenceur, dans lequel il peut définir la nature des modifications à apporter au signal audio. De cette façon, il est possible d'effectuer toutes les opérations nécessaires au processus de création à l'intérieur d'un environnement unique. Il existe, bien sûr, une quantité importante de plugiciels effectuant des traitements originaux sur le signal audio. Ces outils, souvent très intéressants, ont le désavantage d'être, par nature, très hermétiques. Un plugiciel (vst, audioUnit, ladspa, etc.) compilé en format binaire est une boîte noire offerte à l'utilisateur avec très peu de possibilité de personnalisation. Les contrôles accessibles via l'interface sont ceux que les concepteurs ont jugés pertinents mais il arrive que le compositeur pense autrement. Si un plugiciel ne produit pas tout à fait le résultat sonore souhaité par le créateur, ce dernier doit-il bifurquer de son plan original pour faire au mieux avec les outils disponibles ou doit-il quitter le séquenceur et ouvrir un autre logiciel pour y travailler ses sources sonores ? Dans un monde idéal, le compositeur devrait avoir la possibilité de modifier ses plugiciels afin qu'ils répondent le plus exactement possible à ses besoins. Ceci pose toutefois un autre problème. Écrire un plugiciel nécessite un bagage informatique que ne possède généralement pas les compositeurs de musique. Mises à part les notions de traitement de signal indispensables à la conception du plugiciel, il est impératif de maîtriser le langage de programmation orienté-objet C++, avec lequel la plupart des plugiciels sont écrits, et d'installer un environnement de développement de logiciels, afin de compiler le code source en format binaire. Chaque système d'exploitation possède ses propres particularités et le plugiciel doit être compilé indépendamment pour chacun. Tout ceci concerne l'ingénierie informatique, un domaine d'expertise en soi, et nous éloigne grandement du processus de création sonore. La section suivante présente un certain nombres d'alternatives permettant au compositeur de se créer un environnement de travail rapide, efficace et versatile, sans pour autant devenir expert en informatique. Ces solutions offrent des plates-formes de travail très différentes les unes des autres, chacune possédant ses forces et ses faiblesses. Nous élaborerons sur les raisons qui ont motivé le choix du langage JS de Reaper pour le présent projet.

[2]. Ensemble de traitements audio finalisant un morceau de musique, en vue de son exploitation ou de sa diffusion.

3. ALTERNATIVES

Les quatre plates-formes suivantes offrent la possibilité au compositeur d'explorer le traitement de signal et de développer ses propres processus audio, tout en conservant une relation plus ou moins étroite avec la station audionumérique.

3.1. MaxForLive

MaxForLive est une version du logiciel MaxMSP, un environnement de programmation par *patch*, intégré à la station audionumérique Live. Il permet d'insérer, et de modifier en temps réel, des modules développés avec MaxMSP dans la chaîne de production sonore de Live [6]. Très populaire auprès des communautés électroacoustique et électronique, ce couple de logiciels est couramment utilisé pour la composition ou les performances en temps réel. Ces logiciels ne sont malheureusement ni gratuits ni multi-plates-formes (ils ne fonctionnent pas sous linux).

3.2. Faust

Faust (acronyme pour *Functional Audio Stream*) est un langage fonctionnel permettant de définir des chaînes de traitement de signal pouvant ensuite être compilés pour différents environnements (plugiciel ladspa, lv2 ou vst, objet Puredata, Csound, SuperCollider, etc.). Faust est gratuit et supporte les plates-formes les plus populaires (OSX, Windows, linux, iOS, Android, Raspian). Une interface web est également disponible afin de développer la chaîne de traitement de signal. Une fois le code Faust écrit, l'interface fournit automatiquement le code source en C++ pour l'environnement demandé, un diagramme de la chaîne de traitement et une documentation en format pdf. L'interface permet aussi de télécharger un exécutable pré-compilé du traitement développé. Cette option est un énorme avantage sur l'écriture directe de plugiciel puisque la compilation pour un environnement donné nécessite généralement une grande part de configuration et des connaissances informatiques que ne devrait pas avoir à assumer un compositeur de musique. Un autre avantage de Faust est la parallélisation des processus audio à l'intérieur du code [4], permettant de tirer profit des architectures multi-cœurs (de plus en plus répandues) et d'obtenir du code très efficace en temps de calcul. Faust est un environnement d'exploration du traitement de signal très versatile et efficace pour les compositeurs, par contre, il faut quitter le séquenceur pour développer le plugiciel...

3.3. Blue Cat's plug'n script

La compagnie Blue Cat, qui offre plusieurs instruments et effets de grande qualité, a lancé en 2014 un nouveau plugiciel nommé *Plug'n Script* [5]. Il permet de développer des algorithmes de traitement de signal audio, ou des instruments MIDI, à même l'interface du plugiciel, sans quitter le séquenceur. Il utilise le langage hautement optimisé *AngelScript*, d'une syntaxe similaire au C++ et au Java. Le code est compilé « à la volée », ce qui permet de développer sans même arrêter la lecture de la séquence. Cet environnement permet le prototypage rapide d'effets audionumériques dans un langage similaire à celui d'un plugiciel autonome. Il est donc très simple de se servir du code *AngelScript* comme base à la programmation d'un plugiciel indépendant (vst ou audioUnit par exemple). Cependant, cette alternative nécessite des connaissances en programmation plus approfondies qu'une simple maîtrise du traitement de signal puisque le *AngelScript* n'est pas un langage particulièrement simple à manier. *Plug'n Script* supporte une grande quantité de formats de plugiciels mais ne fonctionne que sous OSX et Windows. Qui plus est, ce produit n'est pas gratuit.

3.4. Les plugiciels JSFX de Reaper

Le logiciel Reaper, développé par la compagnie Cockos, propose un environnement de développement de plugiciels complètement intégré à la station audionumérique [7]. Les plugiciels sont écrits à même l'interface avec le langage de script audio JS, compilé « à la volée » par Reaper. Le langage permet de développer des effets audio, des algorithmes MIDI ainsi que des interfaces graphiques à l'aide d'un moteur de dessin vectoriel optimisé. Ce système ne nécessite aucune configuration particulière, la compilation étant prise en charge par le logiciel, et très peu de connaissances en programmation, ce qui en fait un environnement idéal pour le compositeur désirant explorer les techniques de traitement de signal. Reaper fonctionne sur les principales plates-formes (OSX, Windows et linux) et est disponible en téléchargement gratuit sur le site de Cockos. La compagnie fait confiance à la bonne foi des utilisateurs pour acheter une licence, à un prix tout-à-fait raisonnable, si le logiciel est utilisé dans un contexte professionnel. La section suivante élabore sur les particularités du langage de script audio JS de Reaper. Ensuite sera présenté la librairie CookDSP, qui ajoute une couche de fonctionnalités au langage JS, facilitant ainsi la création d'effets audionumériques originaux.

4. LE LANGAGE DE SCRIPT AUDIO JS

JS est un langage de script audio propre à la station audionumérique Reaper. Il permet la création rapide de plugiciels par le biais d'une structure épurée ainsi que d'une intégration directe et efficace à l'architecture du logiciel. Un plugiciel JSFX consiste tout simplement en un fichier texte, compilé « à la volée » lorsque sauvegardé, contenant les directives nécessaires à l'exécution du processus sonore désiré. Les effets et instruments JSFX distribués avec le logiciel peuvent être consultés, modifiés ou utilisés comme point de départ pour la création de traitements originaux. Nous introduirons dans cette section les fonctionnalités de base du langage, la structure d'un fichier JSFX ainsi que les règles de syntaxe essentielles à l'écriture d'effets audio ou d'instrument MIDI.

4.1. Structure d'un fichier JSFX

Un fichier JSFX est un simple fichier texte, sans extension, contenant un certain nombre de lignes de description, suivi par une ou plusieurs sections de code. Les descriptions servent principalement à gérer le contenu de l'interface graphique tandis que les sections de code spécifient les actions à poser (modification des échantillons sonores) en fonction du processus développé et des manipulations effectuées par l'utilisateur.

4.1.1. Les descriptions

Une description est une commande, sous la forme d'un mot-clé, suivi du symbole deux-points (:), puis des arguments spécifiques à la commande. Les deux descriptions essentielles sont *desc*, qui permet de donner un titre au plugiciel, ainsi que *sliderX* (où *X* est remplacé par le numéro du potentiomètre), utilisé pour créer des contrôles graphiques de paramètres. Selon les paramètres donnés à la description *sliderX*, Reaper affichera soit un potentiomètre, soit un menu déroulant. Les descriptions permettent aussi d'établir des canaux de communication audio entre différents plugiciels ou de spécifier des fichiers audio nécessaires au bon fonctionnement du code. Le lecteur est invité à consulter la documentation du langage [2] pour de plus amples détails.

Le code suivant, bien que n'effectuant aucun traitement sur le signal audio, construira l'interface graphique illustrée à la figure 1.

```
desc:Un Plugiciel qui ne fait rien
slider1:0<0,10,1>Compteur d'entiers
slider2:0.5<0,1>Panoramisation
slider3:0<0,1,1{Off,On}>Commutateur On/Off
slider4:0<0,4,1{Rect,Triangle,Cloche}>Enveloppe
```

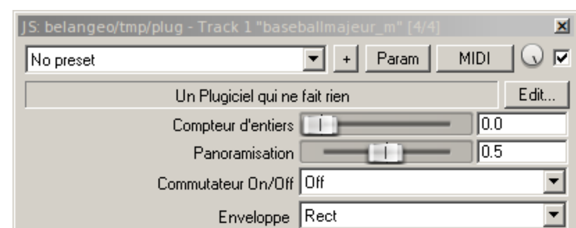

Figure 1. Interface graphique simple.

Le langage JS possède aussi plusieurs fonctions permettant d'effectuer du dessin vectoriel, dont le rafraîchissement a lieu à l'intérieur d'un processus parallèle au processus audio, offrant ainsi la possibilité de construire des interfaces graphiques plus évoluées.

4.1.2. Les sections de code

Les sections de code représentent des actions à poser, c'est-à-dire des manipulations à effectuer sur les signaux, en différents moments du processus. Par exemple, certaines opérations doivent être effectuées à l'initialisation du plugiciel, tandis que d'autres doivent être appliquées à chacun des échantillons, ou encore seulement quand un potentiomètre change de valeurs, etc. Une section est déclarée à l'aide d'une ligne unique, contenant le nom de la section, précédée du symbole arobas (@). Toutes les lignes suivantes appartiennent à cette section, jusqu'à la déclaration de la section suivante (ou jusqu'à la fin du fichier). Voici la liste des sections de code disponibles et leur contexte d'utilisation :

- **@init** : Les commandes contenues dans cette section seront exécutées au chargement du plugiciel ainsi qu'au lancement de la lecture de la séquence.
- **@slider** : Les commandes contenues de cette section ne seront exécutées que lors de la manipulation d'un élément graphique.
- **@block** : Cette section est exécutée une seule fois par bloc d'échantillons, tout juste avant le calcul des échantillons eux-mêmes.
- **@sample** : Le code de cette section est exécutée à chaque période d'échantillonnage, on y a donc accès à chacun des échantillons de façon indépendante.
- **@serialize** : Cette section est exécutée pour la sauvegarde ou la recharge d'un préréglage.
- **@gfx** : Section de code dédiée au dessin vectoriel. On y retrouve les commandes pour les affichages complexes (oscilloscope, spectrogramme, filtre paramétrique, etc.).

L'exemple suivant illustre l'organisation des opérations d'un plugiciel en sections de code. C'est un plugiciel qui met en place un contrôle du volume en décibels ainsi qu'une gestion de la balance des canaux gauche et droit. La section **@init** spécifie la valeur initiale des variables utilisées par le processus. La section **@slider** met à jour les variables en fonction des manipulations de potentiomètres effectuées à l'écran. Le code contenu dans la section **@sample** est exécuté à chaque période d'échantillonnage, ce qui permet d'appliquer les valeurs d'amplitude aux échantillons successifs du signal audio.

```
/* Amplification et Balance */
desc: Gain et Balance

slider1:0<-60,18>Gain (dB)
slider2:0<-100,100>Balance

@init
/* Code d'initialisation */
gain = amL = amR = 1.0;

@slider
/* Code des manipulations graphiques */
gain = 10 ^ (slider1 / 20);
slider2 < 0 ? amL = 1 : amL = 1 - slider2 * 0.01;
slider2 > 0 ? amR = 1 : amR = slider2 * -0.01;

@sample
/* Code affectant chacun des echantillons */
spl0 = spl0 * gain * amL;
spl1 = spl1 * gain * amR;
```

4.2. Les éléments de langage

Le corps d'un plugiciel JSFX est écrit avec le langage EEL2 (*Extensible Embeddable Language*) [3], dont la syntaxe est comparable à celle du C. Ce langage a été conçu expressément pour faciliter l'écriture en temps réel de scripts offrant une puissance de calcul comparable au langages compilés. Il est autonome et nécessite peu de dépendances, ce qui en fait un langage très facile à intégrer à différents environnements, tel que Reaper.

4.2.1. Syntaxe

Tout langage de programmation, aussi simple soit-il, implique un certain nombre de règles de syntaxe. Voici un petit condensé des notions essentielles concernant le langage JS :

- Les **variables** n'ont pas besoin d'être déclarées et elles sont « globales » par défaut (c'est-à-dire qu'elles sont accessibles dans toutes les sections de code). Toute valeur numérique est définie avec une double précision (64-bit).
- Un **commentaire** peut être spécifié à l'aide de la double barre oblique (//) ou entre les symboles « /* » et « */ ».
- Le symbole **point-virgule** (;) sert à séparer les déclarations (opérations mathématiques, assignation à une variable, etc.).
- Les **parenthèses** peuvent servir à clarifier l'ordre des priorités dans une expression mathématique ou à regrouper plusieurs opérations en un seul appel.
- Une mémoire virtuelle (environ 8 millions de mots) unique au plugiciel est accessible à l'aide des crochets ([et]).
- Il n'y a pas de mot-clé tel que *if* ou *else* dans le langage JS, les **expressions conditionnelles** utilisent la syntaxe :

condition ? code si vrai : code si faux

4.2.2. *Variables réservées*

Un certain nombre de variables réservés sont définies dans le langage afin d'assurer la communication entre le logiciel et le plugiciel. En voici un bref aperçu :

- **slider1, slider2, ..., slider64**, noms réservés des 64 potentiomètres possibles. Dans les sections de code, on récupère la valeur d'un potentiomètre via son nom de variable.
- **spl0, spl1, ..., spl63**, noms réservés des 64 canaux audio possibles. En stéréo, le signal de gauche est accessible via la variable **spl0** tandis que le signal de droite est accessible via la variable **spl1**.
- **srate**, variable donnant accès à la fréquence d'échantillonnage courante de Reaper. Cette variable peut être utilisée chaque fois qu'un calcul doit tenir en compte la fréquence d'échantillonnage.
- **play_state**, variable indiquant l'état courant de la lecture (en mode lecture, arrêt, enregistrement, etc.).
- **play_position**, variable donnant la position courante, en secondes, du pointeur de lecture dans la session.
- **tempo**, variable indiquant le tempo, en « bpm », du projet courant.

4.2.3. *Programmation orienté-objet*

Parmi toutes les fonctionnalités du langage JS, celle qui a permis le développement de la librairie CookDSP est sa capacité à gérer du code orienté-objet. Quoique très simple dans son implémentation, elle permet la persistance des variables internes d'une fonction d'un appel à l'autre, ce qui est amplement suffisant pour développer des processus audio indépendants et les contrôler au cours du temps. Seulement deux éléments de syntaxe sont nécessaires pour mettre en place une structure de type objet. Premièrement, à l'intérieur des fonctions, le nom des variables propres à l'objet doivent porter le préfixe *this*. Ensuite, on crée un objet en faisant précéder l'appel de fonction d'un nom de variable (ex. : *var.appel()*). Le nom de variable choisi devient la référence de l'objet créée. Par la suite, chaque appel de fonction qui sera précédé du nom de variable utilisera les variables implicites (et « persistantes ») de l'objet référencé.

L'exemple suivant illustre l'orientation objet du langage JS. Les fonctions *saw* et *saw_do* servent respectivement à initialiser et à calculer les échantillons d'une onde en dent de scie. Deux objets indépendants, oscillant à des fréquences différentes, sont ensuite créés et manipulés à l'aide de ces fonctions. Les deux signaux sont illustrés sur l'oscilloscope de la figure 2.

```
desc: Dent de scie

@init
/* Initialise une dent de scie. */
function saw(freq)
(
    this.inc = freq * 2 / srate;
    this.phs = 0;
);

/* Calcule les echantillons. */
function saw_do()
(
    val = this.phs;
    this.phs += this.inc;
    this.phs >= 1 ? this.phs -= 2;
    val;
);

/* Creation de deux dents de scie. */
r0.saw(172);
r1.saw(86);

@sample
/* Une dent de scie par canal. */
spl0 = r0.saw_do();
spl1 = r1.saw_do();
```

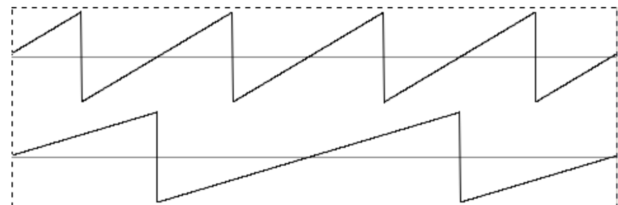

Figure 2. Deux dents de scie de fréquences différentes.

5. LA LIBRAIRIE COOKDSP

CookDSP est une boîte à outils de haut niveau, spécialement dédié au langage JS, facilitant l'élaboration de traitements audionumériques. Concrètement, tout ce que fournit le langage JS est un environnement de développement ainsi que des primitives indispensables au traitement de signal (opérateurs mathématiques, structures d'appels en boucle, instructions conditionnelles, etc.). Tout processus de contrôle ou de traitement audio doit être entièrement construit à partir de ces primitives. Cela peut s'avérer intimidant pour le compositeur ne possédant que des bases en traitement de signal. La librairie CookDSP fournit une collections de fonctions et d'objets prenant en charge certaines particularités propres à la programmation audio, notamment des algorithmes de conversion, un système de gestion de la mémoire, des algorithmes d'interpolation et des générateurs de formes d'onde. La librairie offre aussi plusieurs outils de traitement (filtres, délais, distorsions, réverbérations) et d'analyses (suivi d'enveloppe, suivi de hauteur, centre de gravité spectrale, etc.)

prêts à être utilisés. On y retrouve également tout le nécessaire pour créer des effets dans le domaine spectral, à l'aide de la transformée de Fourier et du vocodeur de phase. La documentation complète peut être consulté sur le site web de la librairie [1]. Pour avoir accès à la collection de fonctions, il faut d'abord importer CookDSP à l'aide de la ligne suivante :

```
import cookdsp.jsfx-inc
```

Les sections suivantes proposent un tour d'horizon détaillé des fonctionnalités offertes par la librairie CookDSP.

5.1. Gestion de l'espace-mémoire

Chaque plugiciel JSFX se voit automatiquement attribué un espace-mémoire privé d'environ 8 millions de mots (trois minutes de son monophonique pour une fréquence d'échantillonnage de 44100 Hz). Cet espace peut être utilisé en différents contextes. Par exemple, pour enregistrer du signal audio en vue d'une utilisation ultérieure, pour initialiser la mémoire d'une ligne de délai, pour mettre en banque des enveloppes et des formes d'onde ou pour conserver des valeurs temporaires dans un algorithme d'analyse. Pour que le plugiciel fonctionne correctement, il est primordial que les différentes utilisations de l'espace-mémoire n'interfère pas entre-elles. La mémoire allouée doit donc être séparée en blocs de différentes tailles selon les besoins du processus. Le code suivant initialise deux blocs de mémoire, le premier commençant à l'index 0 et le second à l'index 1024. Le premier bloc peut donc utiliser au plus 1024 échantillons (0 → 1023) avant de chevaucher le second. Une enveloppe *hanning* est mise en mémoire dans le bloc. La taille du second bloc n'est actuellement limité que par la taille totale de l'espace-mémoire.

```
@init
mem1 = 0;
mem2 = 1024;
k = 0;
loop(1024,
  mem1[k] = -0.5 * cos(2 * $pi * k / 1024) + 0.5;
  k += 1;
);
```

Dans un contexte comme celui-ci, si l'on veut augmenter la taille de l'enveloppe à 2048 échantillons, il faudra aussi augmenter l'index de départ de tous les blocs suivants afin d'éviter les chevauchements. Si le programme contient des dizaines de blocs-mémoire, jongler avec la taille des espaces-mémoire augmentera considérablement les risques d'erreur. La librairie CookDSP propose une gestion automatique de l'espace mémoire avec la fonction *memalloc*. Cette fonction alloue un bloc d'une taille spécifiée en argument et garde en mémoire l'index de départ suivant pour une allocation ultérieure. Chaque allocation de mémoire sera séparée de la suivante par huit espaces vides, qui seront utilisés, entre autre, par les algorithmes d'interpolation à l'écriture et à la lecture des lignes de délai). Tous les objets de la librairie faisant usage de la mémoire utilisent la fonction *memalloc* afin d'assurer une compatibilité avec le reste du code. L'exemple précédant pourrait donc s'écrire comme suit :

```
@init
size = 2048;
mem1 = memalloc(size);
mem2 = memalloc(srate); // une seconde de memoire
k = 0;
loop(size,
  mem1[k] = -0.5 * cos(2 * $pi * k / size) + 0.5;
  k += 1;
);
```

De cette façon, on peut modifier à volonté la valeur de la variable *size*, toutes les allocations de mémoire s'ajusteront automatiquement.

En plus de la fonction *memalloc*, la librairie offre deux objets forts pratiques en ce qui concerne la gestion de la mémoire. Le premier est l'objet *buffer*, qui contient plusieurs algorithmes de génération d'enveloppes et de formes d'onde ainsi que différentes méthodes de lecture. Le second est *delay*, qui permet de créer aisément des effets à base de lignes de délai récursives. En utilisant ces objets, la section d'initialisation de notre exemple se résumerait à ceci :

```
@init
buf.buffer(2048);
buf.buffer_window(1); // enveloppe hanning
del.delay(srate); // une seconde de memoire
```

5.2. Les outils de conversion

La librairie offre plusieurs fonctions permettant de convertir une valeur d'une échelle à une autre. La conversion d'une note midi en fréquence s'effectue avec la fonction *mtof*, l'inverse (Hz → midi) avec *ftom*. Les fonctions *dbtoa* et *atodb* convertissent des valeurs d'amplitude entre l'échelle des décibels (logarithmique) et l'échelle linéaire. La fonction *scale* offre le contrôle sur les ambitus d'entrée et de sortie de la conversion ainsi que sur le type de courbe utilisée (linéaire, exponentielle ou logarithmique). L'exemple suivant effectue deux conversions. Dans un premier temps, la valeur en décibels d'un potentiomètre est récupérée et convertie en amplitude linéaire. Ensuite, la valeur d'un second potentiomètre, normalisée entre 0 et 1, est convertie en fréquence à l'aide d'une courbe exponentielle.

```
import cookdsp.jsfx-inc

slider1:0<-60,18>Gain (dB)
slider2:0.1<0,1>Freq Norm

@slider
gain = dbtoa(slider1);
hz = scale(slider2, 0, 1, 20, 20000, 4);
```

5.3. Les objets de traitement

La librairie CookDSP fournit une vaste gamme de traitements audio, sous la forme de pseudo-classes, pouvant être assemblés afin de construire des chaînes d'effets complexes. Les traitements offerts couvrent le filtrage, les outils de gestion de la dynamique, les effets basés sur les délais, la granulation et les effets dans le domaine spectral. Des outils de génération aléatoire et d'analyse du signal audio accompagnent les objets de traitements et offrent la possibilité de construire de multiples trajectoires de contrôle de paramètre.

Tous les objets de la librairie utilisent la structure décrite ci-dessous.

1) Une fonction portant le nom de l'objet sert à initialiser le processus (une distorsion dans cet exemple).

```
@init
dis.disto(0.7, 5000);
```

2) Diverses fonctions *objet_xxx* servent à modifier le comportement de l'objet.

```
@slider
dis.disto_set_drive(slider1);
dis.disto_set_cutoff(slider2);
```

3) Une méthode *objet_do* exécute le processus audio sur l'échantillon donné en argument. Cette fonction est généralement appelée à chaque période d'échantillonnage, c'est-à-dire dans la section **@sample**.

```
@sample
// Ajustement de l'amplitude
amp = scale(slider1, 0, 1, 0.7, 0.4, 2);
// Applique la distorsion
spl0 = disL.disto_do(spl0) * amp;
```

Voici le code complet pour une distorsion stéréophonique :

```
desc: Simple Distortion

import cookdsp.jsfx-inc

slider1:0.7<0,1>Drive
slider2:5000<500,10000>Lowpass Cutoff

@init
disL.disto(slider1, slider2);
disR.disto(slider1, slider2);

@slider
disL.disto_set_drive(slider1);
disR.disto_set_drive(slider1);
disL.disto_set_cutoff(slider2);
disR.disto_set_cutoff(slider2);

@sample
amp = scale(slider1, 0, 1, 0.7, 0.4, 2);
spl0 = disL.disto_do(spl0) * amp;
spl1 = disR.disto_do(spl1) * amp;
```

6. EXEMPLES

Cette section présente deux exemples complets de plugiciel JSFX construits à l'aide de la librairie CookDSP.

6.1. Un plugiciel d'Auto-Wah

Le premier exemple de plugiciel, illustré à la figure 3, met en place un effet d'*Auto-Wah*, où la fréquence centrale d'un filtre passe-bande varie en fonction de l'amplitude du signal source.

```
/************************************************
Auto-Wah: Un filtre passe-bande dont la frequence
centrale varie en fonction d'un suivi d'amplitude
sur le signal source. (c) belangeo - 2014
************************************************/
desc: Auto-Wah

import cookdsp.jsfx-inc

slider1:5<1,10>Q des Filtres
slider2:5000<500,8000>Frequence Maximale
slider3:20<1,100>Reactivite du Suivi

@init
/* Initialisation du suiveur d'enveloppe. */
fol.follow(10);
/* Initialisation des filtres passe-bande. */
bpL.bp(1000, slider1);
bpR.bp(1000, slider1);

@slider
/* Ajustement de la frequence des filtres. */
bpL.bp_set_q(slider1);
bpR.bp_set_q(slider1);
/* Ajustement de la frequence du suiveur. */
fol.follow_set_freq(slider3);

@sample
/* Suivi d'enveloppe. */
input = (spl0 + spl1) * 0.707;
amp = fol.follow_do(input);
/* Conversion amplitude -> frequence */
freq = scale(amp, 0, 1, 50, slider2, 2);
/* Modifie la frequence centrale des filtres. */
bpL.bp_set_freq(freq);
bpR.bp_set_freq(freq);
/* Filtrage du signal d'entree. */
spl0 = bpL.bp_do(spl0);
spl1 = bpR.bp_do(spl1);
```

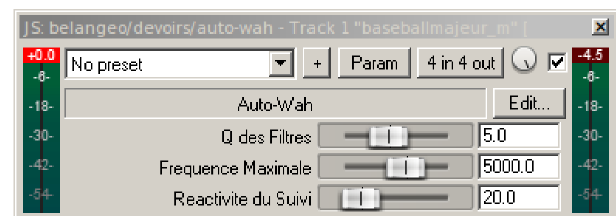

Figure 3. Interface graphique du plugiciel d'Auto-Wah.

6.2. Un plugiciel de délai récursif

Le second exemple de plugiciel, illustré à la figure 4, met en place un effet de délai récursif, où un signal stéréo est délayé puis réinjecté en entrée de la ligne de délai.

```
/*************************************************
Delai Stereo + Feedback: Un signal stereo est
delaye puis reinjecte en entree des lignes de
delai. (c) belangeo - 2014
*************************************************/
desc:Delai Stereo + Feedback

slider1:250<1,1000>Delay Time (ms)
slider2:0<-1,1>Feedback

import cookdsp.jsfx-inc

@init
// Deux lignes de delai de une seconde
d0.delay(srate);
d1.delay(srate);

@slider
// Conversion du temps de delai en echantillons
samps = slider1 * 0.001 * srate;

@sample
// Lecture avec interpolation cubique
val0 = d0.delay_read3(samps);
val1 = d1.delay_read3(samps);

// Ecriture dans les lignes de delai
d0.delay_write(spl0 + val0 * slider2);
d1.delay_write(spl1 + val1 * slider2);

// Mixage des signaux originaux et delayes
spl0 = (spl0 + val0) * 0.5;
spl1 = (spl1 + val1) * 0.5;
```

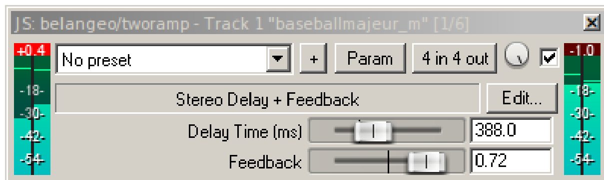

Figure 4. Interface graphique du plugiciel de délai récursif.

7. CONCLUSIONS

Dans cet article, nous avons exploré le langage de script audio JS, intégré à la station audionumérique Reaper, ainsi que la librairie CookDSP, une collection de fonctions facilitant l'exploration du traitement de signal et la création de plugiciels JSFX. Ces outils s'avèrent particulièrement utiles pour le compositeur désirant développer ses propres modules d'effets sonores sans quitter le logiciel de mise en forme de la musique. La décision de créer une librairie de processus sonores pour le langage JS plutôt que pour une plate-forme alternative a été motivé par la simplicité du langage lui-même ainsi que par l'épuration de l'environnement de développement (aucune installation ni compilation de librairies ne sont requises). La simplicité du langage et la prise en charge automatique de la compilation du plugiciel en font un environnement idéal pour initier les non-programmeurs au traitement de signal. Les compositeurs peuvent créer leurs propres banques d'effets audionumériques et les transporter facilement d'une plate-forme à l'autre puisque ceux-ci résident dans de simples fichiers texte. La librairie CookDSP représente un pas de plus vers la liberté d'exploration en fournissant aux créateurs une collection de fonctions automatisant certains aspects propres au traitement de signal (gestion de la mémoire, algorithmes d'interpolation, calcul des coefficients des filtres, etc.). À l'Université de Montréal, en classe de théorie électroacoustique, cette plate-forme a été utilisée avec succès pour illustrer et explorer les diverses techniques de traitement du son.

Beaucoup de travail a été accomplie afin d'optimiser au maximum les fonctionnalités déjà présentes dans la librairie CookDSP. Parmi les projets de développements futurs, en plus de continuer à agrémenter la librairie de nouveaux processus de traitement, une attention particulière sera accordée à la création d'instruments MIDI. Le langage JS prend déjà en charge les messages MIDI en entrée et en sortie mais très peu d'outils existent en ce qui concerne la gestion de la polyphonie. La polyphonie MIDI, ainsi que les éléments de base pour la création de sons de synthèse originaux (enveloppes, lecture de forme d'onde, anti-alias, etc.) seront au cœur des futurs développements de la librairie.

8. REFERENCES

[1] Bélanger, Olivier, « Documentation de la librairie CookDSP », *http://ajaxsoundstudio.com/cookdspdoc/*, 2014.

[2] Cockos Incorporated, « Documentation du langage de script audio JS », *http://www.reaper.fm/sdk/js/js.php*, (2005-2014).

[3] Olofson, David, « The Extensible Embeddable Language », *http://eelang.org/*, (2002-2014).

[4] Orlarey, Y., Fober, D., Letz, S., « Parallelization of Audio Applications with Faust », *6th Sound and Music Computing Conference* , Porto, Portugal, 2009.

[5] Press Release, « Make your own FX and VIs with Blue Cat's Plug'n Script », *Sound On Sound*, septembre, 2014.

[6] Rothwell, Nick, « Max for Live, Real-time Programming Toolkit For Ableton Live », *Sound On Sound*, february, 2010.

[7] Watson, Barry, « Jesusonic Plug-Ins, Reaper Tips and Techniques », *Sound On Sound*, february, 2013.