La programmation JS FX sous Reaper

par F. Loyer

1 août 2017

1 Introduction

Un caractéristique de Reaper est la capacité à écrire des scripts pour obtenir un niveau de personnalisation très avancé.

Cette programmation présente deux formes :

— JS FX : il s'agit de définir des effets analogues aux plugins VST, mais plus faciles à développer. Ces scripts interagissent avec le flux d'événements MIDI ou avec les flux audio ;

— ReaScript : il s'agit de définir des actions en interaction avec l'utilisateur pour personnaliser l'interface de Reaper, et peuvent être comparée à des macros d'une suite bureautique.

Les JS FX peuvent s'intégrer à n'importe quel DAW à j'aide du VST ReaJS (`https://www.reaper.fm/reaplugs/`). En revanche, les ReaScript restent une spécificité Reaper.

Par ailleurs, JS FX impose un langage spécifique propre à Reaper (EEL), alors que les ReaScript peuvent être développés en EEL, Lua ou Python.

Cet ouvrage ne s'intéressera qu'à la programmation de JS FX. Par ailleurs, la démarche retenue ici consiste à prendre des petite réalisation pratique afin d'introduire les notions au fur et à mesure. Cette démarche sera moins exhaustive que le site de référence, qu'il faudra consulter pour obtenir une liste exhaustive de fonction opérateur, etc. : `https://www.reaper.fm/sdk/js/js.php`.

2 Notre premier script JS FX

Pour commencer notre premier effet, cliquons sur le bouton FX d'une piste, puis le menu « FX / Add FX », puis, dans la nouvelle boîte de dialogue « FX / Create JS FX », et enfin saisissons le nom de notre script.

En double-cliquant sur le JS FX, nous pouvons afficher son interface (avec 5 curseurs graphique). Un bouton « Edit... » qui permet de le modifier avec l'éditeur intégré de Reaper. Même si l'on dispose d'un éditeur de fichier texte performant, l'éditeur intégré offre plusieurs avantages : dès la sauvegarde (Ctrl-S), le script est recompilé ce qui modifie l'exécution immédiatement ou affiche un message d'erreur en cas d'erreur empêchant la compilation. De plus, l'éditeur intégré reconnait les mots-clés du langage et les met en évidence avec un code de couleur. Enfin, les variables du programme sont affichée en vis-à-vis du code et permet de lire leurs valeurs.

3 Les sections des fichiers script

Le fichier script créé, nous observons une entête qui décrit les caractéristiques du script, puis des instructions définies dans un langage dit EEL dans différentes sections qui indiquent quant ces instructions sont exécutées.

Les déclarations les plus usuelles que l'on peut lire sont :

desc : Description Le nom du script ;

slider1 : 5<0,10,1> Description décrit un curseur graphique n°1 avec 5 comme valeur par défaut, et des valeurs admises entre 0 et 10 de 1 en 1 ;

slider1 : 1<1,3,1{menu A,Menu B,menu C}> Description décrit un menu déroulant n°1 avec 1 comme valeur par défaut. Les valeurs de 1 à 3 sont choisies d'après un menu déroulant ;

in_pin : Description Cela permet de déclarer une voie audio (mono) en entrée et de lui donner un nom dans la matrice des flux de l'effet.

out_pin : Description Cela permet de déclarer une voie audio (mono) en sortie et de lui donner un nom dans la matrice des flux de l'effet.

Notes : Voir `https://www.reaper.fm/sdk/js/js.php` pour en découvrir de plus avancées).

Lorsque Reaper crée un script, il demande un nom qui sert de nom de fichier (dans le répertoire Effects de Reaper), et est utilisé lorsque l'on choisi un script parmi ceux existant. En revance, lorsque l'on récupère un script nouveau, où lorsque l'on utilise le menu « FX / Scan for new plugins », c'est la ligne `desc:` qui est utilisée. On a donc intérêt à changer cette ligne pour la rendre cohérente avec le nom choisi, et non « new effect » comme proposé par défaut par Reaper.

Alors que la section déclarative suit une syntaxe propre, le reste du fichier est structuré sous forme de sections introduites par un mot clé comme « `@init` ». Chaque section comprends des expressions de même nature (que l'on verra plus tard), mais exécutées à différent moment.

@init Le code est exécuté une fois pour toute,

@slider Le code est exécuté à chaque déplacement de curseur graphique,

@block Le code est exécuté à chaque bloc d'échantillons (en fonction du paramétrage de la carte son,

@sample Le code est exécuté à chaque échantillon (et utilise des variables `spl0`, `spl1`, etc. pour accéder ou générer des échantillons.

@serialize Le code est exécuté en vue de lire/sauvegarder son état (lecture/sauvegarde du projet),

@gfx Le code est exécuté environ 30 fois par seconde en vue de rafraîchir une interface graphique.

4 Calcul des échantillons, cas d'un amplificateur avec distorsions

Les instructions les plus simples à placer dans une sections sont les assignations de valeurs sous la forme suivante.

```
variable = expression;
```

Les expressions sont séparées par des points-virgules (`;`). Elles peuvent utiliser les opérateurs usuels (`+`, `-`, `*`, `/`) [1] Les variables sont soit les variables spéciales (`spl0`, `spl1`, `slider1`, `slider2`...), soit des variables à vous permettant de calculer des variables intermédiaires. L'éditeur intégrer met en évidence les variable spéciales (en rouge). La convention pour valeurs d'échnatillons `spl0`, `spl1`, etc. est une valeur 1 au maximum (0 dB FS). Reaper supporte des valeurs au delà, mais elles satureraient si envoyées directement à la carte son.

Ainsi, pour réaliser un amplificateur avec un facteur de distorsion appliqué après amplification, il est possible de saisir (à copier-coller à la place de votre script). Ainsi dans le script « side chain » en annexe B,

```
// un premier essai

desc:Mon premier JSFX

slider1: 0<-10,10,0.25> Gain (dB)
slider2: 0<0,100,0.1> Distorsion (%)
in_pin: input L
in_pin: input R
out_pin: output L
out_pin: output R

@slider

g = pow(10,slider1/20); // le gain
d = slider2/100;        // un facteur de distorsion

@sample

s0 = g * spl0;
s1 = g * spl0;

spl0 = s0 * (1-d) + s0*s0*s0*d;
spl1 = s1 * (1-d) + s1*s1*s1*d;
```

Toutes les fins de lignes après `//` sont ignorées, ce qui permet de saisir des commentaires.

On remarque ici, l'usage de deux sections, `@slider` et `@sample`, ainsi, la variable « d » n'est calculée qu'une fois à chaque changement. Cela est surtout utile pour des calculs de filtre qui peuvent être complexe au point de préférer ne pas les recalculer à chaque échantillon. La fonction `pow` sert ici à calculer une puissance. On trouve d'autres fonctions à l'adresse `https://www.reaper.fm/sdk/js/basiccode.php`.

5 Expressions avec condition

Le script précédent réalise une moyenne entre la valeur d'un signal amplifié et ce dernier après avoir subit une distorsion. En prenant le signal au cube, nous avons bien un signal avec distorsion et qui reste de même signe. Si nous voulons plutôt un carré, il nous faut rétablir le signe du signal d'origine.

1. L'annexe A définit tous les opérateurs.

Pour cela, il suffit d'utiliser la formulation :

```
test ? valeur si vrai  : valeur si faux
```

Ainsi, nous pouvons réécrire les deux dernières ligne de notre amplificateur avec distorsion ainsi

```
spl0 = s0 * (1-d) + s0*s0*((s0>0)?+1:-1) * d;
spl1 = s1 * (1-d) + s1*s1*((s1>0)?+1:-1) * d;
```

Si besoin, on peut utiliser les opérateurs `!` (non logique), `&&` (ET logique) et `||` (OU logique). Les comparaison sont usuelles, mais avec l'égalité écrite `==`, et l'inégalité, `!=`.

Cet opérateur peut aussi être utilisé pour exécuter des instructions de façon conditionnelle. Ainsi, nous avons dans le script en annexe B, la construction suivante qui indique : si on est dans la phase « Stand by » (0), et si le niveau Root-Mean-Square (RMS) est supérieur au seuil (`tresh`), on initialise la variable `t` à 0 en vue d'un chronométrage, on passe à la phase d'après (1). Sinon, le gain doit être à 1.

```
// Standby phase phase
phase == 0 ? (
  rms > tresh ? (
  t = 0;
  phase = 1) : gain = 1;
);
```

6 Les tableaux

Dans certains cas, des tableaux peuvent être utiles. Ainsi, dans le cas du script en annexe B, on mémorise les derniers échantillons afin de pouvoir garder une somme des puissances instantanées pour en calculer la moyenne.

La gestion des tableaux est assez limitée dans Reaper : ce dernier considère un tableau de 8 millions de valeurs que l'on peut accéder à l'aide de l'expression :

```
tableau[index]
```

Ici, `tableau` pointe dans la mémoire de 8 millions de valeurs sur le début de tableau, et `index` pointe à l'intérieur. (Il s'agit de conventions usuelles : rien n'empêche d'inverser les rôles, car la première chose que Reaper fait avec ces nombres est de les additionner !) Il convient alors d'initialiser autant de variables que de tableaux requis, en évitant des tableaux qui se superposent.

7 La réception et l'émission des événements MIDI

La norme MIDI est une norme qui permet les échanges d'information musicale entre contrôleur, synthétiseur et logiciel DAW (comme Reaper). Elle permet d'échanger des informations telles que les notes jouées, des états de contrôleurs (potentiomètre, pédales, contrôleurs à vents), etc.

On distingue deux types d'événements MIDI : les événements courts qui prennent 2 ou 3 octets, et les événements longs (typiquement les événements SysEx) qui prennent une taille quelconque.

Pour lire un événement MIDI court, il suffit d'appeler la fonction :

```
midirecv(offset,msg1,msg2,msg3)
```

Cette fonction retourne une valeur non nulle si elle reçoit un événement court, et 0 sinon. Dans le cas où cette fonction reçoit un événement, l'événement est retiré, et les variables proposées contiennent alors les valeurs correspondantes (offset est un nombre d'échantillons depuis un début de bloc, `msg1`, `msg2` et `msg3` contiennent les valeurs portées par l'événement MIDI).

`msg1` dit aussi « statut » contient souvent un type de message et le numéro de **canal numéroté à partir de 0**. Ainsi, il convient de séparer ces deux informations.

Réciproquement, l'émission utilise son homologue :

```
midisend(offset,msg1,msg2,msg3)
```

Le script ce dessous propose une transposition et utilise quelques notions supplémentaires : `while` qui permet évalue une condition est interprète une autre tant que la première est vraie et la notation `$x90` qui permet d'écrire un nombre en hexadécimal (issue ici de la norme MIDI). Le `&` est un ET binaire qui est calculé indépendamment avec chacun des bits des opérandes. (Il diffère du ET logique, `&&` qui fonctionne de façon globale) .

```
desc: Transposition

slider1:0<-12,+12,1>Transposition

@block

MIDI_NOTE_OFF = $x80;
MIDI_NOTE_ON  = $x90;

// On boucle tant que l'on reçoit un événement court
while(midirecv(offset,msg1,msg2,msg3)) (
   type = msg1 & $xF0;
   canal  = msg1 & $x0F;

   (type == MIDI_NOTE_OFF  ||  type == MIDI_NOTE_ON) ?
       midisend(offset, msg1, msg2 + slider1, msg3)
   :
       midisend(offset, msg1, msg2, msg3);

);
```

Attention, lorsque le type est `MIDI_NOTE_ON`, `msg2` est le numéro de la note concernée, et `msg3` la nuance demandée, mais si elle est nulle, l'événement doit être considéré comme synonyme d'un `MIDI_NOTE_OFF`.

8 L'affichage graphique

Avant d'appeler des commandes graphiques, il convient de définir la géométrie de la surface graphique placée sous les paramètres du script. C'est l'objet de la définition du block `@gfx` qui s'écrit :

```
@gfx taille_horizontale taille_verticale
```

Les valeurs nulles sont permises et signifient que la taille importe peu.

Une fois la section définie, on définit une séquence de rafraîchissement qui sera appelée environ 30 fois par seconde. Pour cela, il convient de considérer des variables spéciales (affichées en rouge dans l'éditeur) qui sont nécessaires pour paramétrer les fonctions graphiques :

gfx_x abscisse de la position courante (en pixel),

gfx_y ordonnée de la position courante (en pixel, le 0 étant en haut),

gfx_r composante rouge de la couleur courante (entre 0 et 1),

gfx_g composante verte de la couleur courante (entre 0 et 1),

gfx_b composante bleue de la couleur courante (entre 0 et 1),

gfx_a indice de transparence (entre 0 — complètement transparent — et 1 — complètement opaque).

Avec quelques commandes, on peut afficher l'écart en demi-tons entre la note la plus élevée d'un accord et la moins élevée. (noter le `midisend` qui émet à nouveau la note quelle puisse être jouée).

```
desc:Ambitus

@init
table_note = 0;
memset(table_note,0,128);

MIDI_NOTE_OFF = $x80;
MIDI_NOTE_ON  = $x90;

@block
// On boucle tant que l'on reçoit un événement court
while(midirecv(offset,msg1,msg2,msg3)) (
toto=msg1;
   type = msg1 & $xF0;
   canal = msg1 & $x0F;

   (type == MIDI_NOTE_OFF) ? (table_note[msg2] = 0);
   (type == MIDI_NOTE_ON)  ? (table_note[msg2] = msg3);

   midisend(offset, msg1, msg2, msg3);
);

@gfx 0 32

a=table_note[48];
min_note = 128;
max_note = -1;
i=0;
loop(128,
    (table_note[i]>0) ? (
       min_note = min(min_note,i);
       max_note = max(max_note,i);
    );
```

```
    i += 1;
);
gfx_x = 0;
gfx_y = 0;

gfx_r = 0;
gfx_g = 0;
gfx_b = 1;
gfx_a = 1;

gfx_setfont(1,"Arial", 32, "");
gfx_rect(gfx_x,gfx_y,100,32);


(max_note >= 0) ?
(
  gfx_r = 1;
  gfx_g = 1;
  gfx_b = 1;
  gfx_a = 1;
  gfx_drawnumber(max_note-min_note,0);
);
```

Les fonctions disponibles sont assez nombreuses. Il convient donc de consulter celles qui existent sur le site de l'éditeur (`https://www.reaper.fm/sdk/js/gfx.php`).

A Opérateurs

Les opérateurs sont définit par la page `https://www.reaper.fm/sdk/js/basiccode.php#js_ops`.

A.1 Opérateurs numériques

-valeur	opposé de la valeur
+valeur	valeur inchangée
v1 + v2	somme
v1 - v2	différence
v1 * v2	produit
v1 / v2	division
v1 % v2	modulo
v1 ^ v2	puissance

A.2 Opérateurs binaires

Les opérations suivantes portent sur des valeurs binaires. Chaque bit du résultat est calculé d'après chacun des bits des opérandes initiales.

v1 | v2	OU binaire
v1 & v2	ET binaire
v1 ~ v2	XOR (ou exclusif) binaire
v1 << n	décalage à gauche de n bit. (Multiplication par 2^n).
v1 >> n	décalage à droite de n bit. (Division par 2^n).

A.3 Comparaisons

v1 == v2	comparaison approchée (égalité)
v1 === v2	comparaison exacte (égalité)
v1 != v2	comparaison approchée (différence)
v1 !== v2	comparaison exacte (différence)
v1 < v2	comparaison : inférieur
v1 <= v2	comparaison : inférieur ou égal
v1 > v2	comparaison : supérieur
v1 >= v2	comparaison : supérieur ou égal

A.4 Opérations logiques

! valeur	NON logique
v1 || v2	OU logique
v1 && v2	ET logique

A.5 Test conditionnel

L'opérateur ? s'utilise d'une des façon suivantes :

```
condition ? si\_vrai : si\_faux
condition ? si\_vrai
```

La condition est évaluer est la deuxième ou troisième expression est évaluée selon le cas et sa valeur est retournée. Cet opérateur est souvent utilisé avec des opérations impliquant des effets de bord (modification de variable, envoi de commandes MIDI, affichage graphique, etc.) Lorsque la troisième partie est manquante, la valeur de l'expression vaut 0 dans le cas où la condition est fausse.

A.6 Modification de variable

La modification de variable présente les formes suivantes :

y = z	affecte z à y
y *= z	multiplie y par z et affecte le résultat à y
y /= diviseur	divise y par le diviseur et affecte le résultat à y
y %= diviseur	divise y par le diviseur et afecte le reste (modulo) à y
base ^ = exposant	affecte à base, sa puissance à l'exposant exposant
y += z	ajoute à y, z
y -= z	ôte à y, z
y |= z	assigne à y un OU avec les bits de z
y &= z	assigne à y un ET avec les bits de z
y = z	assigne à y un XOR (OU exclusif) avec les bits de z

A.7 Accès aux tableaux

L'usage des tableaux s'utilise de la façon suivante :

```
z=x[y];
x[y]=z;
```

Dans ce cas, la cellule x+y est lue où écrite. 8 388 608 cellule sont ainsi disponibles.

L'utilisation du mot-clé `gmem` permet d'accéder à un espace mutualisé entre tous les scripts. Seuls 1 048 576 valeurs sont ainsi dispo :

```
z=gmem[y];
gmem[y]=z;
```

Une introduction en amont du script permet de limiter la porter des espaces `gmem` à ceux d'une même famille.

```
options:gmem=famille_de_plugins
```

A.8 Séquence d'opérations

Une séquence d'opération s'écrit :

```
opération1 ; opération2 ; opération 3 ; ... ;
```

Le résultat retourné est simplement la dernière opération évaluée.

B Sidechain ducker

```
// (C) 2017, Frédéric Loyer
// Creative Common, CC-BY-SA

desc:FLR Ducker
//tags: dynamics ducker
```

```
//author: Frédéric Loyer

slider1:-20<-60,0,.1>Threshold (dB)
slider2:0<0,1000,1>RMS Size (ms)
slider3:20<0,500,1>Attack (ms)
slider4:.5<0,500,.1>Hold (ms)
slider5:200<0,1000,1>Release (ms)
slider6:100<0,100,1>Wet/dry (%)
slider9:0<-120,60,1>Output (dB)

in_pin:left input
in_pin:right input
in_pin:sidechain left input
in_pin:sidechain right input
out_pin:left output
out_pin:right output

@init
gain = 1;
c_ampdB = 8.65617025;

@slider
treshdB = min(slider1,-.1);
tresh = exp(treshdB/c_ampdB);
attack = slider3/1000*srate;
hold = slider4/1000*srate;
release = slider5*srate/1000;

rms_size_1 = rms_size;
rms_size = min( max(slider2/1000*srate,1) , 1000000);
rms_size_1 != rms_size ? (
    rms_sqr_sum = rms_bpos = rms_tbl = 0;
    memset(rms_tbl,0,rms_size);
);

mode_make_up = ratio < .5 ? 1:slider8;

volume = exp(slider9/c_ampdB);

phase = 0

@sample
sidechain_signal = max(abs(spl2),abs(spl3));

rms_sqr_sum = max(rms_sqr_sum - rms_tbl[rms_bpos],0) + (rms_tbl[rms_bpos]
      = sqr(sidechain_signal));
(rms_bpos+=1) >= rms_size ? rms_bpos=0;
rms = sqrt(rms_sqr_sum/rms_size);

// Standby phase phase
```

```
phase == 0 ? (
  rms > tresh ? (
  t = 0;
  phase = 1) : gain = 1;
);

// Attack phase
phase == 1 ? (
  t >= attack ? (
     t=0;
     phase = 2) :
  (
    gain = 1- (t/attack)
  )
);

// Hold phase
phase == 2 ? (
  t >= hold ? (
    t=0;
    phase = 3) :
  (
    gain = 0
  )
);

//Release phase
phase == 3 ? (
  t >= release ? (
     t=0;
     phase = 0) :
  (
     gain = t/release
  )
);

t += 1;

gain = (gain*slider6 + 100-slider6)/100;

spl0 = (spl0 *= gain * volume) ;
spl1 = (spl1 *= gain * volume) ;
```