Spécification de l'extension LibAudioStream

Stéphane Letz

ANR-12-CORD-0009
INEDIT Project
Programme ContInt
http://inedit.ircam.fr/

Spécification de l'extension LibAudioStream

Stéphane Letz
GRAME, Centre National de Création Musicale
letz@grame.fr

Abstract

LibAudioStream est un moteur de rendu audionumérique disponible sous forme de librairie, permettant de manipuler des ressources audio à travers le concept de flux. Ce moteur est facilement intégrable dans des applications qui nécessitent de jouer des fichiers son, des montages audio, et d'appliquer des effets et des traitements DSP en temps-réel. Une algèbre de description, de composition et de transformation des flux audio permet de construire des expressions complexes, que l'on va pouvoir ordonnancer à des dates précises dans le futur, et dont le rendu sera ensuite exécuté en temps réel le moment venu. L'application a un contrôle à l'échantillon près de ce qui est fait et à quel moment, tout en étant déchargée des calculs audio temps réel proprement dit. Ce rapport donne une description formelle du langage d'expressions et des fonctionnalités de *LibAudioStream*.

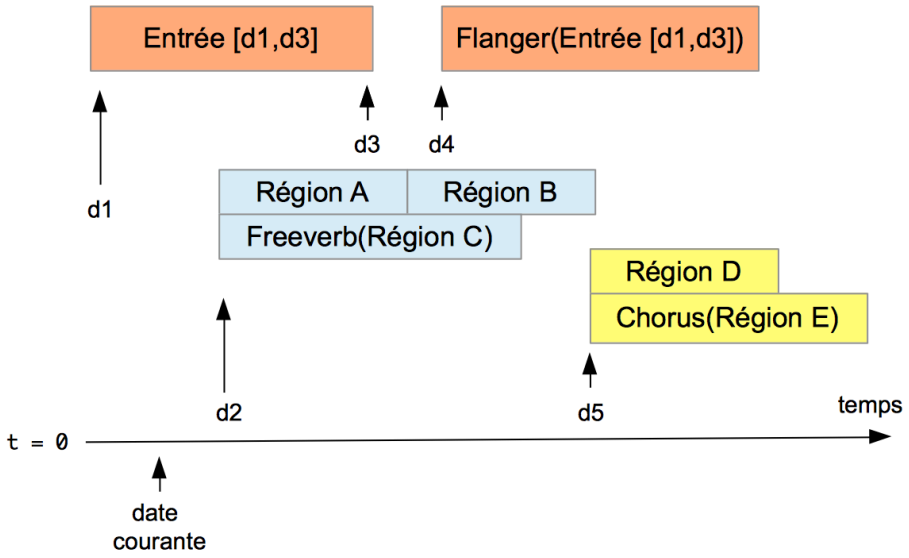

Figure 1: Exemple de timeline contenant plusieurs expressions flux construites à partir de régions de fichiers audio, des expressions faisant référence au flux temps-réel, et insérées à des dates différentes

1 Introduction

LibAudioStream permet la manipulation de fichiers et ressources audio via le concept de flux. Dans la métaphore bien connue de bancs de montage audionumérique de type ProTools, cette librairie permet de construire de manière algorithmique des expressions audio multi-canaux combinant des flux (fichiers ou régions de fichiers audio, entrée temps-réel) à l'aide de différentes opérations de montage, mixage, découpage, sélections de canaux, applications d'effets...

Ces flux seront ensuite insérés à des instants précis dans une *timeline* [1] globale (Figure 1), grâce à un ordonnancement à l'échantillon. Leur évolution dans le temps (changement des valeurs de paramètres de contrôle des effets par exemple) peut être contrôlée. Enfin le rendu temps réel sera assuré par un composant en interaction avec le système audio bas niveau (CoreAudio, PortAudio, JACK...).

Dans cette approche flux, l'entrée temps réel est considérée comme un flux multi-canal (typiquement correspondant au n entrées physiques de la carte audio utilisée), dont la date de début correspondra au début du rendu de la timeline globale. Bien que non encore connu dans sa totalité au moment du démarrage du rendu, ce flux peut malgré tout être manipulé *comme un flux temps différé déjà connu* (donc par exemple découpé, monté en séquence ou mixé) dans la mesure ou les contraintes de causalité permettant son rendu sont respectées. Cette approche originale permet alors de considérer le flux temps réel comme un cas particulier de flux temps différé, et qui pourra alors être manipulé le plus souvent avec les mêmes opérations.

1. Une *timeline* est ici définie comme une «partition dynamique» dont on peut constamment modifier le contenu à venir, en ajoutant et en supprimant des éléments à jouer, ou en spécifiant la valeur réelle de « dates symboliques» lorsqu'elles seront connues.

2 L'entrée temps réel

Les environnements audio classiques opèrent bien souvent une distinction entre les opérations et constructions qui relèvent du temps différé (ou "hors temps") de celles qui agissent "en temps réel". L'approche explorée dans la *LibAudioStream* est d'autoriser autant que possible la manipulation de l'entrée temps réel avec les mêmes opérations, algorithmes de manipulation, ou transformations que ceux utilisées sur des flux temps différés déjà connus.

Ainsi il est possible par exemple de définir le concept *d'écho audio*, vu comme le mixage de plusieurs instances d'un même flux légèrement décalées dans le temps et dont chaque instance serait atténuée. Cette fonction de transformation pourra alors être utilisée aussi bien sur un flux temps différé (par exemple créé à partir d'un fichier audio) que directement appliqué sur l'entrée temps réel. Pour une même description de haut niveau, c'est l'étape de rendu qui s'occupera de réaliser les opérations adaptées, soit sur le flux fichier, soit sur le flux temps réel.

L'entrée temps-réel, dont le début sera l'instant de démarrage du rendu de la timeline globale, sera enregistrée en mémoire, et accessible dès cet instant aux opérations de montage, découpe, mixage, transformation... La contrainte de causalité exprime alors le fait qu'il ne sera pas possible de faire le rendu d'une expression complexe qui aurait besoin de *lire des échantillons du flux temps-réel qui sont dans le futur de la date de rendu courante*, en revanche il sera possible d'exprimer des opérations de composition de flux temps réel ou il sera *lu à nouveau plus tard dans le temps de la timeline globale* : par exemple décaler le flux temps réel d'une seconde permettre d'exprimer le concept de délai d'une seconde sur celui-ci.

3 Gestion des effets

Un flux peut être transformé par un effet décrit dans le langage Faust [1] . La compilation dynamique de code DSP Faust est réalisée grâce à *libfaust* [2] , la version embarquable et dynamique du compilateur Faust associée à LLVM [3] qui permet de traduire les traitements synchrones, écrites en Faust, en code binaire exécutable fonctionnant à vitesse native.

On suppose que le nombre de canaux du flux est "compatible" avec le nombre d'entrées de effet, c'est à dire est un sous-multiple ou multiple du nombre des canaux de l'effet. L'effet sera potentiellement dupliqué sous la forme plusieurs instances contrôlées par les mêmes noms de paramètres. Le contrôle sera asynchrone au sens ou les valeurs des paramètres seront échantillonnées par le rendu audio en temps-réel, dans la fenêtre temporelle d'activation de l'effet.

4 Rendu

On appelle *rendu sonore* le calcul global des flux résultants et leur jeu sur une interface audio multi-canaux. Le rendu effectif est typiquement cadencé de manière périodique par une interruption audio, et le calcul effectué sur des blocs d'échantillons consécutifs de taille donnée. Les entrées audio sont donc reçues et traitées, les sorties sont calculées et produites par blocs. Par ailleurs *les informations asynchrones de contrôle* (signaux reçus de capteurs, interactions avec l'interface utilisateur...) produisent des événements discrets qui influencent et s'entrelacent avec le *flux d'échantillons synchrone* du rendu.

4.1 Prise en compte des paramètres de contrôle

Dans une approche discrète du traitement des événements de contrôle, on considère classiquement que la valeur du signal de contrôle est constante pendant l'intervalle du buffer audio sur lequel il s'applique.

Comme les paramètres de contrôle des effets sont datés à l'échantillon, les valeurs vont alors changer potentiellement à l'intérieur d'un bloc audio. La fonction de rendu va donc découper le bloc audio en "tranches" correspondant à des valeurs constantes des paramètres de contrôle.

4.2 Dates symboliques

Pour le contrôle du rendu, certaines opérations (démarrage/arrêt des expressions flux, changement des paramètres des effets...) vont être spécifiées à l'aide de date connues. D'autres en revanche vont dépendre d'événements asynchrones, dont les dates ne seront précisément connues qu'au moment de leur occurrence effective.

Il sera malgré tout possible d'ordonnancer des démarrage/arrêts d'expressions ou modifications de paramètres des effets *à l'avance* grâce à la notion de *date symbolique*. Une *date symbolique* pour être créée et utilisée dans une expression, sera instanciée avec une date réelle lorsque celle ci sera effectivement connue, et toutes les expressions qui y font références sont alors effectivement insérés dans l'ordonnanceur global.

5 Formalisation

Dans cette section nous donnons une description précise des expressions que l'on peut jouer avec la *LibAudioStream*. Comme indiqué en introduction, le paradigme général est celui du montage audionumérique type Protools. Une expression E décrit comment découper, mettre en séquence, monter, mixer et transformer des matériaux audio. Toute expression bien formée, aussi complexe soit elle, dénote donc l'équivalent d'un fichier audio multi-canal, c'est à dire un ou plusieurs flux d'échantillons de même longueur.

Une particularité de la *LibAudioStream* est d'aborder le temps-réel dans le cadre conceptuel du montage temps-différé, grâce à l'expression `input`. On peut considérer `input` comme un fichier audio en devenir, correspondant à l'enregistrement des entrées audio depuis l'instant 0 (fonction `start()` de la librairie). Même si `input` n'est jamais connu en totalité, il peut être découpé, monté, transformé comme un matériau temps-différé ordinaire, à condition que le rendu de l'expression résultante respecte les contraintes de causalité.

Dans la suite de cette section nous allons définir la syntaxe abstraite des expressions de la *LibAudioStream* (5.1), le sous-ensemble des expressions bien formées (5.2), et plusieurs fonctions sémantiques :

- $\mathcal{D}[\![.]\!]$ la durée, exprimée en nombre d'échantillons d'une expression (5.4)
- $\mathcal{C}[\![.]\!]$ le nombre de canaux que comporte une expression (5.3)
- $\mathcal{A}[\![.]\!]$ le flux d'échantillons multi-canal qu'une expression dénote (5.5.3)
- $\mathcal{T}[\![.]\!]$ la date au plus tôt à laquelle une expression peut être jouée sans violer la causalité (5.6).

Les transformations sont exprimées en langage Faust. Pour un programme Faust F donné, on notera $\mathcal{I}[\![F]\!]$ son nombre d'entrées et $\mathcal{O}[\![F]\!]$ son nombre de sorties.

5.1 Syntaxe abstraite

Nous donnons ici la syntaxe abstraite des expressions. La correspondance entre les nom des opérations utilisées ici et les fonction de l'API de la *LibAudioStream* sera donnée plus loin.

E	::=	`input`	entrées temps-réel
	|	`0`	silence élémentaire de 1 échantillon
	|	`file(`*pathname*`)`	fichier audio
	|	`begin(`E`,`d`)`	début d'une expression sur une durée d
	|	`end(`E`,`d`)`	fin d'une expression privée de son début sur une durée d
	|	`seq(`$E1$`,`$E2$`)`	mise en séquence de deux expressions
	|	`mix(`$E1$`,`$E2$`)`	mixage de deux expressions
	|	`par(`$E1$`,`$E2$`)`	mise en parallèle de deux expressions
	|	`loop(`E`,`n`)`	mise en boucle de E répétée n fois
	|	`apply(`F`,`E`)`	application du traitement Faust F sur l'expression E
	|	E`[`c`]`	extraction du canal c de l'expression E

5.2 Expressions bien formées

On dira d'une expression E qu'elle est bien formée ($E \in$ EBF) si et seulement si elle respecte les règles suivantes :

`input`	$\in$ EBF	
`0`	$\in$ EBF	
`file(`*pathname*`)`	$\in$ EBF	si *pathname* designe un fichier audio valide
`begin(`E`,`d`)`	$\in$ EBF	si $E \in$ EBF et $0 \leq d \leq \mathcal{D}[\![E]\!]$
`end(`E`,`d`)`	$\in$ EBF	si $E \in$ EBF et $0 \leq d \leq \mathcal{D}[\![E]\!]$
`seq(`$E1$`,`$E2$`)`	$\in$ EBF	si $E1 \in$ EBF et $E2 \in$ EBF et $\mathcal{C}[\![E1]\!]=\mathcal{C}[\![E2]\!]$
`mix(`$E1$`,`$E2$`)`	$\in$ EBF	si $E1 \in$ EBF et $E2 \in$ EBF et $\mathcal{D}[\![E1]\!]=\mathcal{D}[\![E2]\!]$ et $\mathcal{C}[\![E1]\!]=\mathcal{C}[\![E2]\!]$
`par(`$E1$`,`$E2$`)`	$\in$ EBF	si $E1 \in$ EBF et $E2 \in$ EBF et $\mathcal{D}[\![E1]\!]=\mathcal{D}[\![E2]\!]$
`loop(`E`,`n`)`	$\in$ EBF	si $E \in$ EBF et $n \geq 1$
`apply(`F`,`E`)`	$\in$ EBF	si $E \in$ EBF et F désigne un programme Faust et $\mathcal{I}[\![F]\!] = \mathcal{C}[\![E]\!]$
E`[`c`]`	$\in$ EBF	si $E \in$ EBF et $0 \leq c < \mathcal{C}[\![E]\!]$

Ces règles sont volontairement très restrictives de façon à alléger la description de la sémantique des expressions bien formées. Dans la pratique on peut considérer que s'il manque des canaux ou de la durée, ceci soit compensé automatiquement par du silence.

5.3 Nombre de canaux d'une expression bien formée

Toute expression bien formée E a un nombre fini de canaux $\mathcal{C}[\![E]\!]$ donné par les règles suivantes :

$$
\begin{array}{rcl}
\mathcal{C}[\![\texttt{input}]\!] & = & \text{nombre d'entrées audio de la machine} \\
\mathcal{C}[\![\texttt{0}]\!] & = & 1 \\
\mathcal{C}[\![\texttt{file(}\mathit{pathname}\texttt{)}]\!] & = & \text{nombre de canaux audio du fichier} \\
\mathcal{C}[\![\texttt{begin(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{C}[\![E]\!] \\
\mathcal{C}[\![\texttt{end(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{C}[\![E]\!] \\
\mathcal{C}[\![\texttt{seq(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{C}[\![E1]\!] = \mathcal{C}[\![E2]\!] \\
\mathcal{C}[\![\texttt{mix(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{C}[\![E1]\!] = \mathcal{C}[\![E2]\!] \\
\mathcal{C}[\![\texttt{par(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{C}[\![E1]\!] + \mathcal{C}[\![E2]\!] \\
\mathcal{C}[\![\texttt{loop(}E\texttt{,}n\texttt{)}]\!] & = & \mathcal{C}[\![E]\!] \\
\mathcal{C}[\![\texttt{apply(}F\texttt{,}E\texttt{)}]\!] & = & \mathcal{O}[\![F]\!] \\
\mathcal{C}[\![E\texttt{[}c\texttt{]}]\!] & = & 1
\end{array}
$$

5.4 Durée d'une expression bien formée

Toute expression bien formée E a une durée $\mathcal{D}[\![E]\!]$, éventuellement infinie, exprimée en nombre d'échantillons :

$$
\begin{array}{rcl}
\mathcal{D}[\![\texttt{input}]\!] & = & \infty \\
\mathcal{D}[\![\texttt{0}]\!] & = & 1 \\
\mathcal{D}[\![\texttt{file(}\mathit{pathname}\texttt{)}]\!] & = & \text{durée du fichier audio} \\
\mathcal{D}[\![\texttt{begin(}E\texttt{,}d\texttt{)}]\!] & = & d \\
\mathcal{D}[\![\texttt{end(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{D}[\![E]\!] - d \\
\mathcal{D}[\![\texttt{seq(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{D}[\![E1]\!] + \mathcal{D}[\![E2]\!] \\
\mathcal{D}[\![\texttt{mix(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{D}[\![E1]\!] = \mathcal{D}[\![E2]\!] \\
\mathcal{D}[\![\texttt{par(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{D}[\![E1]\!] = \mathcal{D}[\![E2]\!] \\
\mathcal{D}[\![\texttt{loop(}E\texttt{,}n\texttt{)}]\!] & = & n.\mathcal{D}[\![E]\!] \\
\mathcal{D}[\![\texttt{apply(}F\texttt{,}E\texttt{)}]\!] & = & \mathcal{D}[\![E]\!] \\
\mathcal{D}[\![E\texttt{[}c\texttt{]}]\!] & = & \mathcal{D}[\![E]\!]
\end{array}
$$

5.5 Sémantique audio d'une expression bien formée

Toute expression bien formée E a une sémantique audio, notée $\mathcal{A}[\![E]\!]$, représentée par un flux multi-canal d'échantillons.

5.5.1 Flux d'échantillons

On notera $s^n = s_0, s_1, \ldots, s_{n-1}$ un flux de n échantillons audio. Le flux vide est noté $\emptyset$.

Trois opérations sur les flux d'échantillons sont définies : addition de deux flux de même longueur, concaténation de deux flux, sélection d'un segment de flux entre deux indices.

$$
\begin{array}{rcl}
\textit{addition} \quad u^n + v^n & = & u_0^n + v_0^n, u_1^n + v_1^n, \ldots, u_{n-1}^n + v_{n-1}^n \\
\textit{concaténation} \quad u^n v^m & = & u_0, u_1, \ldots, u_{n-1}, v_0, v_1, \ldots, s_{m-1} \\
\textit{sélection} \quad s^n[i,j[& = & e_i, e_{i+1}, \ldots, e_{j-1}, \; s^n[i,i[= \emptyset
\end{array}
$$

5.5.2 Flux multi-canal d'échantillons

Un flux multi-canal est un groupe de c flux d'échantillons de même longueur n : $m^{c,n} = \langle s_0^n; s_1^n; \ldots; s_{c-1}^n \rangle$. Un flux multi-canal peut être vue comme analogue à un fichier audio multi-canal.

De manière similaire aux flux simples, quatre opérations sur les flux multi-canaux sont définies, addition de deux flux multi-canaux de même longueur et de même nombre de canaux, concaténation de deux flux multi-canaux de même nombre de canaux, mise en parallèle de deux flux multi-canaux de même longueur, sélection d'une partie entre deux indices.

$$\begin{array}{rrcl}
\textit{addition} & \langle u_0^n; u_1^n; \ldots; u_{c-1}^n\rangle + \langle v_0^m; v_1^m; \ldots; v_{c-1}^m\rangle & = & \langle u_0^n + v_0^m; u_1^n + v_1^m; \ldots; u_{c-1}^n + v_{c-1}^m\rangle \\
\textit{concatenation} & \langle u_0^n; u_1^n; \ldots; u_{c-1}^n\rangle\langle v_0^m; v_1^m; \ldots; v_{c-1}^m\rangle & = & \langle u_0^n v_0^m; u_1^n v_1^m; \ldots; u_{c-1}^n v_{c-1}^m\rangle \\
\textit{parallelisation} & \langle u_0^n; u_1^n; \ldots; u_{c-1}^n\rangle|\langle v_0^n; v_1^n; \ldots; v_{p-1}^n\rangle & = & \langle u_0^n; u_1^n; \ldots; u_{c-1}^n; v_0^n; v_1^n; \ldots; v_{p-1}^n\rangle \\
\textit{selection} & \langle s_0^n; s_1^n; \ldots; s_{c-1}^n\rangle[i,j[& = & \langle s_0^n[i,j[; s_1^n[i,j[; \ldots; s_{c-1}^n[i,j[\rangle
\end{array}$$

Par ailleurs nous introduisons la possibilité de transformer un flux multi-canal en appliquant un traitement Faust F comportant c entrées et p sorties, ce que l'on notera : $F(\langle u_0^n; u_1^n; \ldots; u_{c-1}^n\rangle) = \langle v_0^n; v_1^n; \ldots; v_{p-1}^n\rangle$.

5.5.3 Sémantique audio d'une expression

Munis de ces opérations nous pouvons définir la sémantique audio $\mathcal{A}[\![E]\!]$ d'une expression E bien formée comme un flux multi-canal d'échantillons :

$$\begin{array}{rcll}
\mathcal{A}[\![\texttt{input}]\!] & = & \langle s_1^\infty; s_2^\infty; \ldots; s_c^\infty\rangle & \text{les échantillons audios reçu sur les entrées} \\
\mathcal{A}[\![\texttt{0}]\!] & = & \langle 0\rangle & \\
\mathcal{A}[\![\texttt{file(}\textit{pathname}\texttt{)}]\!] & = & \langle s_0^n; s_1^n; \ldots; s_{c-1}^n\rangle & \text{les échantillons du fichier audio} \\
\mathcal{A}[\![\texttt{begin(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{A}[\![E]\!][0,d[& \\
\mathcal{A}[\![\texttt{end(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{A}[\![E]\!][d,\mathcal{D}[\![E]\!][& \\
\mathcal{A}[\![\texttt{seq(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{A}[\![E1]\!]\mathcal{A}[\![E2]\!] & \\
\mathcal{A}[\![\texttt{mix(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{A}[\![E1]\!]+\mathcal{A}[\![E2]\!] & \\
\mathcal{A}[\![\texttt{par(}E1\texttt{,}E2\texttt{)}]\!] & = & \mathcal{A}[\![E1]\!]\,|\,\mathcal{A}[\![E2]\!] & \\
\mathcal{A}[\![\texttt{loop(}E\texttt{,1)}]\!] & = & \mathcal{A}[\![E]\!] & \\
\mathcal{A}[\![\texttt{loop(}E\texttt{,}n\texttt{)}]\!] & = & \mathcal{A}[\![E]\!]\mathcal{A}[\![\texttt{loop(}E\texttt{,}n\texttt{-1)}]\!] & \\
\mathcal{A}[\![\texttt{apply(}F\texttt{,}E\texttt{)}]\!] & = & F(\mathcal{A}[\![E]\!]) & \\
\mathcal{A}[\![E\texttt{[}c\texttt{]}]\!] & = & \langle s_c^n\rangle & \text{avec } \mathcal{A}[\![E]\!] = \langle s_0^n \ldots; s_c^n; \ldots; s_{p-1}^n\rangle
\end{array}$$

5.6 Causalité d'une expression

Pour une expression E, on notera $\mathcal{T}[\![E]\!]$, la date au plus tôt à laquelle cette expression peut être jouée sans violée la causalité. Si E est une expression purement temps différée, c'est à dire ne faisant pas appel à `input`, alors il n'y a pas de contraintes, l'expression peut être jouée à tout moment et donc $\mathcal{T}[\![E]\!] = -\infty$.

Toute expression E qui fait appel à `input` est temps-réel, ce qui se traduit par $\mathcal{T}[\![E]\!] > -\infty$.

$$\begin{array}{rcl}
\mathcal{T}[\![\texttt{input}]\!] & = & 0 \\
\mathcal{T}[\![\texttt{0}]\!] & = & -\infty \\
\mathcal{T}[\![\texttt{file(}\textit{pathname}\texttt{)}]\!] & = & -\infty \\
\mathcal{T}[\![\texttt{begin(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{T}[\![E]\!] \\
\mathcal{T}[\![\texttt{end(}E\texttt{,}d\texttt{)}]\!] & = & \mathcal{T}[\![E]\!]+d \\
\mathcal{T}[\![\texttt{seq(}E1\texttt{,}E2\texttt{)}]\!] & = & \max(\mathcal{T}[\![E1]\!],\mathcal{T}[\![E2]\!] - \mathcal{D}[\![E1]\!]) \\
\mathcal{T}[\![\texttt{mix(}E1\texttt{,}E2\texttt{)}]\!] & = & \max(\mathcal{T}[\![E1]\!],\mathcal{T}[\![E2]\!]) \\
\mathcal{T}[\![\texttt{par(}E1\texttt{,}E2\texttt{)}]\!] & = & \max(\mathcal{T}[\![E1]\!],\mathcal{T}[\![E2]\!]) \\
\mathcal{T}[\![\texttt{loop(}E\texttt{,}n\texttt{)}]\!] & = & \mathcal{T}[\![E]\!] \\
\mathcal{T}[\![\texttt{apply(}F\texttt{,}E\texttt{)}]\!] & = & \mathcal{T}[\![E]\!] \\
\mathcal{T}[\![E\texttt{[}c\texttt{]}]\!] & = & \mathcal{T}[\![E]\!]
\end{array}$$

Ainsi par exemple $\mathcal{T}[\![\texttt{end(input,100)}]\!] = 100$ indiquant que l'on ne peut pas écouter l'entrée temps-réel privée de ces 100 premiers instants avant la date 100.

6 Structure de la librairie

La description formelle présentée précédemment est une simplification de l'API réelle qui est implémentée. Voici une description rapide de son utilisation comme elle serait utilisée dans une application réelle. [2]

6.1 Mise en route, arrêt de la librairie

Les fonctions suivantes sont utilisée pour démarrer et arrêter le moteur audio :

```
AudioPlayerPtr OpenAudioPlayer(long inChan,
                               long outChan,
                               long sample_rate,
                               long buffer_size,
                               long stream_buffer_size,
                               long rtstream_duration,
                               long renderer,
                               long thread_num);

long StartAudioPlayer(AudioPlayerPtr player);

long StopAudioPlayer(AudioPlayerPtr player);

void CloseAudioPlayer(AudioPlayerPtr player);
```

6.2 Création et manipulation des flux

Les flux sont soit des flux primitifs (créés à partir d'un fichier audio, ou des entrées temps réel), soit construits à partir de combinaison de flux. Les opérations classiques de montage, mixage, découpage, transformations de flux sont disponibles.

```
AudioStream MakeRegionSound(const char* name, long begin, long end);

AudioStream MakeCutSound(AudioStream sound, long begin, long end);

AudioStream MakeSeqSound(AudioStream s1, AudioStream s2, long xFade);

AudioStream MakeMixSound(AudioStream s1, AudioStream s2);

......
```

2. La description complète de l'API est donnée dans le document annexe [4]

6.3 Création et manipulation des effets

Un flux peut être transformé par un effet décrit dans le langage Faust. On suppose que le nombre de canaux du flux est "compatible" avec le nombre d'entrées de effet, c'est à dire est un sous-multiple ou multiple du nombre des canaux de l'effet.

L'effet sera potentiellement dupliqué sous la forme plusieurs instances contrôlées par les mêmes nom de paramètres. Le contrôle sera asynchrone au sens ou les valeurs des paramètres seront échantillonnées par le rendu audio en temps-réel, dans la fenêtre temporelle d'activation de l'effet.

```
AudioEffect MakeFaustAudioEffect(const char* name, ......);

const char* GetNameEffect(AudioEffect effect);

long GetControlCountEffect(AudioEffect effect);

void GetControlParamEffect(AudioEffect effect,
                           long control,
                           char* label,
                           float* min,
                           float* max,
                           float* init);

AudioStream MakeEffectSound(AudioStream sound,
                            AudioEffect effect,
                            long fadeIn,
                            long fadeOut);

.....
```

6.4 Gestion des dates symboliques

Les dates symboliques sont créées et utilisées comme des "identifiants" pour démarrer/arrêter des expressions flux, ou changer les valeurs de paramètres. Une date symbolique donnée sera alors instanciée avec une date réelle, lorsque celle-ci sera connue.

```
SymbolicDate GenSymbolicDate(AudioPlayerPtr player);

SymbolicDate GenRealDate(AudioPlayerPtr player, audio_frames_t date);

long SetSymbolicDate(AudioPlayerPtr player,
                     SymbolicDate symb_date,
                     audio_frames_t real_date);

audio_frames_t GetSymbolicDate(AudioPlayerPtr player,
                               SymbolicDate symb_date);

.....
```

6.5 Insertion des expressions flux, ordonnancement et rendu

Le temps démarre au moment du démarrage du rendu de la timeline. Les expressions créées par les opérations de combinaison de flux peuvent alors être insérées dans la timeline à des dates, connues et précises à l'échantillon, ou symbolique et instanciées plus tard, de même pour les paramètres de contrôle des effets. Toutes ces opérations sont insérées dans un ordonnanceur global qui assure le tri temporel des actions, et les effectue à leur dates d'échéance.

Les expressions vont alors être démarrées, arrêtées, et leur paramètres de contrôle (typiquement des effets Faust utilisés) modifiés pendant le rendu de la timeline globale.

```
long StartSound(AudioPlayerPtr player,
                AudioStream sound,
                SymbolicDate date);

long StopSound(AudioPlayerPtr player,
               AudioStream sound,
               SymbolicDate date);

long SetTimedControlValueEffect(AudioPlayerPtr player,
                                const char* effect,
                                const char* path,
                                float value,
                                SymbolicDate date);
```

7 Conclusion

Nous avons présenté *LibAudioStream*, un moteur audio audionumérique disponible sous forme de librairie et permettant de manipuler des expressions audio complexes, mêlant des éléments temps différé et temps réel. L'originalité de cette approche est de traiter les entrées temps-réel comme un flux en devenir, qui peut être manipulé comme un flux déjà connu. Grace à l'utilisation de libfaust, la version dynamique et embarquable du compilateur Faust, des effets audio peuvent être décrits et activés. Enfin l'étape de rendu de ces expressions synchrones peut être contrôlée dans le temps grâce à la notion de date symbolique, instanciées à la réception d'événements de contrôle asynchrones.

Remerciements

Cette recherche est menée dans le cadre du projet INEDIT soutenu par l'Agence Nationale pour la Recherche [ANR-12-CORD-0009] que nous tenons à remercier ici.

References

[1] Y. Orlarey and D. Fober and S. Letz "FAUST : an Efficient Functional Approach to DSP Programming", *New Computational Paradigms for Computer Music*, Editions DELATOUR FRANCE, 2009.

[2] Y. Orlarey "Version librairie du compilateur Faust", *INEDIT Livrable 3.4.a*

[3] Low Level Virtual Machine "Language Reference Manual", *llvm.org/docs/LangRef.html*

[4] LibAudioStream 2.00 API reference, *INEDIT Livrable 3.2.b*