DJ'Oz : Rapport de projet

Philippe Verbist Antoine Paris
3521-13-00 3158-13-00

4 décembre 2014

1 Structure du programme

Nous avons divisé, comme demandé, le programme en trois parties :

— Une fonction Interprete, qui prends une partition en paramètre et qui renvoie une liste d'échantillons ;
— une fonction Mix, qui prend (1) une fonction qui permet d'interpréter une partition et (2) une musique et qui renvoie un vecteur audio, sous la forme d'une liste de flottants compris dans l'intervalle [-1.0 ;1.0] ;
— un fichier example.dj.oz, qui contient une musique.

Les deux fonctions que nous avons créées sont écrites entièrement en programmation déclarative. Par ailleurs, nous avons rendu toutes nos sous-fonctions récursives terminales.

1.1 Fonction Interprete

1.1.1 Sous-fonction principale

Cette fonction est codée autour de la sous-fonction fun InterpreteFlattened FlattenedPartition, qui parcourt la partition passée en paramètre et qui, à chaque itération, prend le premier élément de la liste, l'analyse (dans l'ordre : s'agit-il d'une transformation, d'un silence ou d'une note ?) et crée un échantillon. Cet échantillon est soit obtenu immédiatement s'il s'agit d'une note ou d'un silence (cas simple), soit après l'appel d'une autre sous-fonction s'il s'agit d'une transformation - chaque transformation ayant une fonction qui lui correspond (cas complexe). Par exemple, la sous-fonctioN Etirer Facteur Part est appelée pour la transformation etirer(facteur :F P). Ceci est vrai pour toutes les transformations, sauf muet(P), pour laquelle il n'existe pas de sous-fonction Muet Part.

1.1.2 Sous-fonctions de transformation

Comme dit ci-dessus, il y a autant de fonctions de transformation que de transformations (sauf pour muet(P)). On retrouve donc :

— `{WantedDuration Part}`
— `{Etirer Facteur Part}`
— `{Bourdon Note Part}`
— `{Transpose Demitons Part}`
— `{Instrument InstrumentAtom Part}`

Le fonctionnement de chacune de ces sous-fonctions suit toujours le même schéma :

— la partition passée en paramètre `Part` est transformée en une liste d'échantillon par un appel à
`{InterpreteFlattend Partition}`
(la sous-fonction principale)
— cette liste d'échantillons est ensuite parcourue par une sous-sous-fonction, qui applique la transformation demandée sur chacun des échantillons.

Mettre un exemple ?

1.1.3 Sous-fonctions complémentaires

Pour alléger certaines tâches de nos sous-fonctions de transformation, nous avons créé 4 sous-fonctions complémentaires, à savoir :

— VoiceDuration ListEchantillon
— NumberOfSemiTones Note
— NameToNumber Name
— ToNote Note [1]

La seule fonction qui présente ici un réel intérêt est la fonction NumberOfSemiTones Note. Nous l'expliquerons un peu plus loin.

1.2 Fonction Mix

1.2.1 Sous-fonction principale

A l'instar de la fonction Interprete, Mix est codée autour d'une sous-fonction principale : fun MixMusic Music, qui parcourt la musique (une liste de morceaux) passée en paramètre et qui, à chaque itération, prend le premier élément de la liste, l'analyse d'après toutes les possibilités que peut être un morceau et crée un vecteur audio. De nouveau, deux cas se distinguent :

— un cas simple : le vecteur audio peut être obtenu directement à partir d'une voix, d'une partition ou d'un vecteur audio déjà existant dans un fichier .wav.
— un cas plus complexe, à savoir un filtre ou un merge (jouer deux musiques en même temps), qui s'applique sur un vecteur audio déjà existant. Nous appelons ces cas 'complexes' car ils peuvent chacun s'appliquer sur un nouveau morceau, qui peut lui-même être soit simple soit complexe. Il est en effet permis d'appliquer un filtre sur un filtre sur un filtre... A noter que l'extrémité d'une telle chaine est toujours un cas simple.

1.2.2 Sous-fonctions de création de vecteur audio (cas simples)

Les 3 cas simples cités ci-dessus (voix, partition, wave) ne nécessitent que l'utilisation de deux sous-fonctions (voix et partition reviennent au même, puisqu'il suffit d'appliquer Interprete sur partition()) :

— MixVoice Voice (dont la méthode est donnée dans le rapport)
— Projet.readFile File (fonction donnée)

1.2.3 Sous-fonctions de filtre et merge (cas complexes)

Il y a autant de sous-fonction de filtre qu'il y a de filtres possibles (sauf pour renverser) :

1. Cette fonction nous a été donnée dans l'énonce du projet. Elle a été reprise sans être modifiée.

— RepetitionNB NB AV
— RepetitionDuree Duree AV
— Clip Bas Haut AV
— ... (la suite de la liste se déduit assez facilement, et n'est pas particulièrement intéressante)

2 Complexité calculatoire

Dans cette section, nous donnons les complexité calculatoire propre de chaque fonction, c'est à dire sans tenir des sous-fonctions utilisées.

Fonctions	**Temporelle**	**Spatiale**
InterpreteFlattened	n	1
DureeTrans WantedDuration Part	n, taille de Part	1
Etirer Facteur Part	n, taille de Part	1
Bourdon Note Part	n, taille de Part	1
Transpose Demitons Part	n, taille de Part	1
Instrument InstrumentAtom Part	n, taille de Part	1
VoiceDuration ListEchantillon	n, taille de ListEchantillon	1
NumberOfSemiTones Note	1	1
NameToNumber Name	1	1
ToNote Note	1	1
MixMusic Music	n, taille de Music	1
MixVoice Voice	n, taille de Voice	1
Fill F Duree	n, valeur de Duree	1
Merge MusicsWithIntensity	n, taille de MusicsWithIntensity	1
Combine L1 L2	n, taille de la plus grande des listes	1
Lissage AV Duree	n, taille de AV	1
HauteurToNote	1	1
NumberToNote	1	1

TABLE 1 – Analyse de la complexité calculatoire de nos fonctions.

3 Décisions prises et astuces de programmation

Programmation déclarative Comme dit ci-dessus, aucune structure non-déclarative n'a été nécessaire pour écrire notre programme. Nous n'en avons donc pas utilisé.

Récursion terminale Pour rendre notre code plus rapide, nous nous sommes arrangés pour que toutes nos sous-fonctions soient récursives terminales. Nous avons également utilisé des accumulateurs.

Astuce lors des récursions A de nombreux endroits du code, nous devions construire progressivement une liste (en utilisant la récursion terminale). Une technique consiste à utiliser la fonction Append Accumulateur NouvelElement. Le problème est que Append va d'abord parcourir l'accumulateur en entier avant "d'ajouter" le nouvel élément... Or, très souvent, l'accumulateur est très grand (en particulier si la musique à mixer est grande) et le nouvel élément très petit (il s'agit, très souvent, d'un seul élément). Pour accélérer notre code, nous avons donc inversé les arguments dans le Append, et à la fin de la fonction, nous renvoyons Reverse Acc. Ceci permet de ne parcourir qu'une seule fois la liste en entier, et donc de réduire la complexité.

Astuce lors de Reverse

4 Extensions

5 Pistes d'améliorations