VisualImpro : Documentation

Jérémy Lixandre
mail jeremy.lixandre@enseirb-matmeca.fr

Juillet 2017

Table des matières

1 Introduction

Ce document vise à présenter et documenter le projet d'outil d'improvisation musicale intégré à la plateforme Bela dont le développement a commencé en Juin 2017. Nous décrirons son fonctionnement en détail aussi bien pour l'utilisateur que le programmeur, ainsi que les possibilités d'amélioration pour un développement ultérieur.

Avant de continuer, il est conseillé de s'informer sur les généralités concernant le Bela (son fonctionnement, ses entrées/sorties, etc) ainsi que le capelet additionnel pour les entrées audio sur le wiki de bela.io .

2 Présentation générale du projet

Le projet consistait en la création d'un outil pour l'improvisation musicale, permettant à des musiciens jouant en même temps d'avoir une retrospective visuelle et en temps réel de leur improvisation. Cette représentation permet notamment de visualiser des corrélations entre les musiciens deux à deux. Formellement, il se résume à la chose suivante :

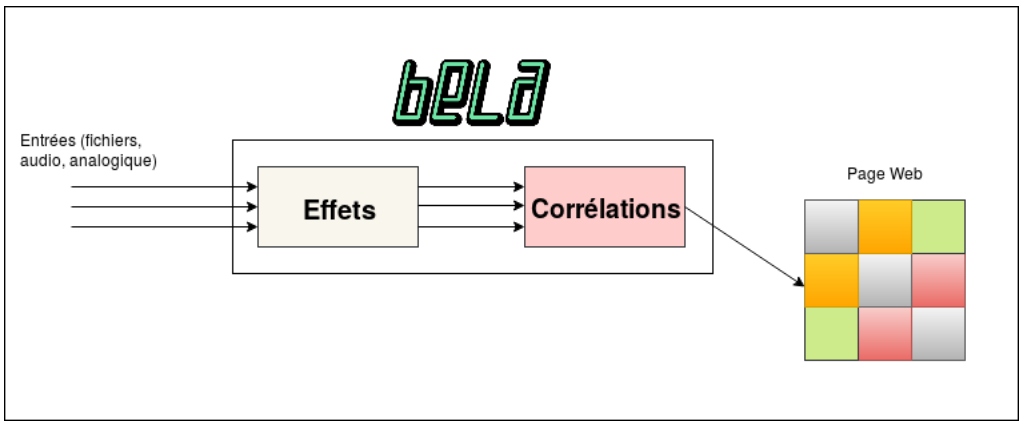
```
Entrée : N pistes musicales (musiciens et/ou pistes enregistrées)
Sortie : une matrice M de taille NxN,
    où M(i,j)=M(j,i) est la corrélation des pistes i et j.
```

Cette matrice doit évidemment se mettre à jour régulièrement et en temps réel. De plus, la notion de "corrélation" étant large et redéfinissable, on souhaitait pouvoir changer de fonction de corrélation facilement selon les besoins.

Il a été de plus décidé d'utiliser comme support la plateforme Bela, contenant un grand nombre d'entrées/sorties audio et idéale pour ce genre de projets.

Finalement les objectifs visés étaient les suivants :

— répondre au problème exposé ci dessus
— proposer une visualisation simple de la matrice (sur une page web)
— rendre simple le changement d'algorithmes de calcul de la matrice
— permettre la possibilité d'ajouter des effets audio avant calcul



3 Utilisation

Cette partie explique comment utiliser le logiciel. Il existe deux manières de le lancer, toutes deux en ligne de commande, l'une depuis la session utilisateur et l'autre en ssh depuis Bela.

3.1 Depuis l'ordinateur

Cette méthode est la plus simple pour le néophyte car elle est opaque. Pour utiliser cette méthode, il suffit d'avoir un PC sous Linux avec le logiciel nodejs installé.

Tout d'abord, dézippez le fichier VisualImproExe.zip . Vous allez vous retrouver avec un dossier VisualImproExe sur votre ordinateur, contenant tout le nécessaire pour lancer le projet. Rendez vous dans ce dossier en ligne de commande.

```
jeremy@PC:~/Enseirb/stage2A/bela/VisualImproExe$ ls
config.cfg  index.html  Makefile  node_modules  server.js  src  tracks  VisualImpro
```

Ce dossier contient deux fichiers importants pour l'utilisateur : le fichier de configuration `config.cfg`, et l'exécutable `VisualImpro`.

Le fichier `config.cfg` sert à configurer le programme avant son execution. Vous pouvez ici choisir les fichiers que vous voulez ajouter en spécifiant leur chemin, ainsi que le nombre de pistes audio (2 pistes max, qui correspondent aux entrées audio de l'étage du bas), de pistes analogiques (8 entrées sur l'étage supérieur du Bela), ainsi que les 3 méthodes de calcul utilisées (ces méthodes seront détaillées plus tard). Vous pouvez enfin choisir la taille des buffers de calcul et choisir d'activer les effets ou non.

Tous les paramètres sont expliqués en commentaire dans le fichier.

Configurez le fichier comme nécessaire. Vous pouvez ajouter les fichiers wav que vous souhaitez utiliser dans le dossier `tracks` pour plus de simplicité. Les fichiers wav que vous utilisez devront être échantillonnés à 44100 Hz (sinon la vitesse de lecture sera inadaptée).

Une fois le fichier configuré et sauvegardé, vous pouvez lancer le programme. Lancez en ligne de commande `./VisualImpro` depuis le répertoire `VisualImproExe/`. Cette commande démarrera automatiquement le serveur, le programme sur Bela et la page web.

```
jeremy@PC:~/VisualImpro/VisualImproExe$ ./VisualImpro
scp ./configtmp.cfg root@192.168.7.2:/root/Bela/projects/VisualImpro/config/
configtmp.cfg                                 100% 1176     1.2KB/s   00:00
VISUALIMPRO SERVER:
Server listening on 192.168.7.1:12345
ssh root@192.168.7.2 'cd /root/Bela/projects/VisualImpro && ./VisualImpro config/./conf
igtmp.cfg' &
************* VisualImpro ************
CONNECTED: 192.168.7.2:56376


------ Settings ------
* Files : 0
* Standard Audio Tracks : 2
* Additionnal Audio Tracks : 2
* Sample Rate : 44100
* Buffer Processing Length : 32768
* Processing Speed (bufflen/samplerate) : 0.743039 s
* Effects disabled
```

Vous pouvez ensuite, si ce n'est déjà fait, brancher vos entrées monopistes sur le Bela. Les entrées disponibles seront toujours dans l'ordre croissant. Par exemple, si vous avez décidé d'activer 5 entrées analogiques, seules les entrées numérotées de 0 à 4 seront disponibles. Les autres sont désactivées.

Vous pourrez ensuite observer sur la page web qui s'est ouverte, la matrice de corrélation des signaux que vous avez passés en entrée.

Lorsque vous voudrez arrêter le programme, appuyer sur Ctrl+C dans le terminal ouvert. Cette commande stoppe le programme et supprime les fichiers wav du Bela. Vous aurez ensuite accès, dans le dossier `logs`, à un enregistrement en texte des matrices de corrélation successives. Ces fichiers sont nommés par la date et l'heure de fin du programme.

3.2 En ssh depuis Bela

Cette utilisation est plus compliquée mais il est utile de la connaitre pour un programmeur.
Vous devrez d'abord ouvrir 2 terminaux sur lesquels vous vous connecterez en ssh sur Bela.

```
jeremy@PC:~$ ssh root@192.168.7.2
Linux bela 3.8.13xenomai-bone41 #2 Wed Mar 26 11:41:31 GMT 2014 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

 ____  _____ _        _
| __ )| ____| |      / \
|  _ \|  _| | |     / _ \
| |_) | |___| |___ / ___ \
|____/|_____|_____/_/   \_\

The platform for ultra-low latency audio and sensor processing

http://bela.io

Bela release image, v0.2.0b, 15 December 2016

More info at https://github.com/BelaPlatform/bela-image/releases

root@bela ~$
```

Dans le premier terminal, rendez vous dans le repertoire `~/Bela/IDE/public/VisualImpro`. Tapez ensuite la commande `node server.js`. Cette commande démarre le serveur communiquant entre la page web et le programme.

Ouvrez ensuite une page web à l'adresse 192.168.7.2:8080. Ceci correspond à l'adresse IP de Bela et le port du programme utilisé.

Enfin, dans le second terminal, rendez vous dans le dossier `~/Bela/projects/VisualImpro`. Modifiez ensuite le fichier `settings.cfg`, de la même manière que le fichier `config.cfg` de la première méthode. Vous devrez importer vos fichier wav manuellement, ou via l'IDE Bela à l'adresse 192.168.7.2 (visitez le wiki de bela.io pour plus d'informations). Enfin, lancez la commande `./VisualImpro`. Le programme se lance.

Pour arrêter le programme vous devrez tout fermer à la main. A la fin du programme, les résultats sont stockés dans le fichier `log/log`, cependant on ne conserve pas de trace des précédents résultats.

Cette méthode est plus compliquée mais utile pour un développeur, car plus rapide et plus transparente que la première (il n'y a pas de transfert de fichiers wav donc plus rapide). Il est possible de laisser tourner le serveur et la page web et de ne relancer que le programme à chaque fois.

Il est aussi possible de lancer le programme via l'IDE plutôt qu'en ligne de commande, en spécifiant les bons flags au compilateur si vous recompilez.

3.3 Ajouter des plug-in de traitement de la matrice

Le calcul de la matrice peut se configurer à l'execution comme nous l'avons vu. Il est de plus possible de rajouter des plug-in de calcul au programme. Ces fonctions se répartissent en trois catégories, et correspondent au trois étapes de calcul de la matrice :

— les fonctions de "pre-processing", traitement du signal d'entrée en amont
— les fonctions de calcul de coefficient de correlation
— les fonctions associant un coefficient à une couleur.

Pour ajouter une fonction, votre fichier devra respecter les critères suivants :

— il devra s'appeler Preproc*.cpp, Coeff*.cpp ou Color*.cpp, selon que vous ajoutez une fonction de preprocessing, de correlation ou de couleur.
— les fonctions ajoutées devront respecter les prototypes standards exposés plus bas.

Enfin, le fichier placé dans le répertoire `process` devra respecter le squelette suivant :

```
#include "../utilities.cpp" //pour inclure la structure Triplet

// code de fonctions auxiliaires

extern "C"{
// code de ma fonction Preproc, Color ou Coeff
}
```

De plus, pour tester votre fonction avant de l'ajouter au programme, vous aurez besoin de la classe Triplet :

```
class Triplet{
public:
  int one, two, three;
  Triplet(int _one, int _two, int _three) : one(_one), two(_two), three(_three){}
};
```

Ce triplet correspond à un triplet RGB. Les entiers contenus sont donc entre 0 et 255, `one` étant le rouge, `two` le vert et `three` le bleu.

3.3.1 Preproc

Ces fonctions ont le prototype suivant :

```
std::vector<std::vector<float> > PreprocX (std::vector<std::vector<float> > buff);
```

L'argument en entrée est une matrice de vecteurs représentatant les signaux d'entrée. Ici $buff.size()$ est égal au nombre de pistes, et la longueur de tous les élements du tableau (un vecteur) est identique et correspond à la

longueur du signal à traiter en temps réel. La matrice de sortie sera donc de taille $buff.size() * x$, où x peut être différent de la longueur des signaux initiaux.

Cette fonction peut correspondre par exemple à :
— un calcul d'enveloppe du signal
— un filtrage
— une transformée de fourier
— un rééchantillonnage
— etc

Par exemple, la fonction `PreprocEnergy` calcule l'energie moyenne du signal sur des petits intervalles et les vecteurs retournés sont des enveloppes d'energie, de taille bien inférieure à la taille du signal. Il est intéressant de réduire la taille des signaux d'entrée grâce à ces fonctions, car cela améliore nettement les performances du programme. Il peut par exemple être judicieux de créer une fonction qui garde 1 échantillon sur 2,3 ou 4 de manière à alléger les calculs de corrélations en suite de chaine.

3.3.2 Coeff

Ces fonctions ont le prototype suivant :

```
float CoeffX (std::vector<float> s1, std::vector<float> s2);
```

Les arguments en entrée sont 2 vecteurs, éléments de la matrice retournée par la fonction `Preproc` précédente. Le flottant en sortie correspond à la correlation de ces deux signaux.

Cette fonction sera appelée $\frac{n(n-1)}{2}$ fois, où n est le nombre de pistes. La corrélation est symétrique, entre 0 et 1, et vaut toujours 1 lorsqu'un signal est corrélé à lui même.

La corrélation classique correspond au produit scalaire euclidien.

3.3.3 Color

Ces fonctions ont le prototype suivant :

```
Triplet ColorX (float coeff);
```

Elle convertit un coefficient de correlation en une couleur, représentée par son triplet RGB. Voir les exemples d'echelles `ColorBlackToWhite` et `ColorGreenToRed` déjà implémentés.

3.3.4 Ajouter les fonctions

Une fois que votre fonction est testée, placez la dans le repertoire `process` du projet. Dans ce repertoire, compilez avec `make`. Vous n'avez alors plus qu'à changer les paramètres du fichier de configuration que vous utilisez. Si vous avez créé une fonction `ColorBlueToYellow`, dans un fichier `ColorBlueToYellow.cpp`, changez le fichier de configuration en specifiant `COLOR ColorBlueToYellow` à la place de ce qui y est écrit actuellement.

4 Généralités sur Bela

4.1 Caractéristiques techniques

La plateforme embarquée Bela est idéale pour réaliser ce genre de projets. En effet, elle dispose de nombreuses entrées audio, analogiques et digitales pour capter les signaux sonores d'entrée, ainsi que de nombreuses sorties, et se connecte à un PC sous Linux en branchant un simple cable USB. Il est dès lors très simple de coder sur Bela, en ssh depuis un terminal par la commande `ssh root@192.168.7.2`, ou depuis l'IDE depuis un navigateur à l'adresse 192.168.7.2 (accessible sans connection internet). D'autres informations sont disponibles sur le wiki de Bela.

4.2 Structure d'un projet Bela en C++

Voir la documentation de bela.io .

Le projet d'improvisation musicale a été implémenté en C++. Un projet se crée via l'IDE ("new project"). Il correspond à un dossier contenant au moins un fichier, le fichier `render.cpp`.

Ce fichier contient lui même 3 fonctions :

— `setup()` appelée avant le démarrage de l'audio. Elle initialise et prépare les ressources
— `render()` est appelée lors du processus audio. Elle est appelée régulièrement, à chaque fois qu'un nouveau bloc audio est disponible, avec une priorité supérieure à toutes les autres taches sur le processeur. Chaque fois qu'elle est appelée, elle dispose en argument de buffers contenant les echantillons à traiter.
— `cleanup()` est appelée à la fin du processus et libère eventuellement les ressources allouées, termine des taches, etc.

Ces fonctions prennent toutes les mêmes arguments :

— `BelaContext * context`, une structure contenant tous les paramètres du programme, dont les buffers audio et analogiques (voir la doc).
— `void *userData`, un pointeur laissé à disposition du programmeur pour communiquer des données entre les différentes fonctions. Pour le projet, nous l'utiliserons notamment pour transferer des paramètres de configuration du `main` à la fonction `setup`.

Compte tenu de la régularité des appels de render, il est important de veiller à ce que seuls les tâches les plus importantes soient exécutées dans cette fonction. En effet, si une tâche trop coûteuse en temps est présente dans la fonction, `render` risque de ne pas finir à temps avant l'arrivée des prochains échantillons, certains blocs seront donc manqués.

Ainsi lorsqu'on aura besoin d'exécuter des fonctions couteuses, on les exécutera dans des tâches auxiliaires. Il est possible d'en créer grâce à la structure `AuxiliaryTask` et aux fonctions associées. Voir la doc ici ainsi que les programmes d'exemple Bela et le code du projet.

Enfin, le programme contient un fichier `main.cpp`. Si aucun fichier main n'est créé, alors le Makefile spécifie lui même un main par défaut. Pour notre part, nous avons eu besoin de créer un main plus complexe que l'original.

4.3 Compiler et executer un programme Bela

Pour compiler un projet Bela depuis un terminal, se rendre dans le repertoire `~/Bela`, et utiliser
`make PROJECT=VisualImpro`. De plus il est nécessaire de spécifier certains flags et librairies, par exemple la librairie dl pour `dlopen()`. On compile donc par la ligne `make PROJECT=VisualImpro CPPFLAGS=-g LDLIBS=-ldl` . Il est aussi possible de compiler depuis l'IDE en spécifiant les flags dans les paramètres.

5 Structure globale du projet

Nous allons présenter dans cette section, le fonctionnement global du programme, sans rentrer dans les détails. Ce fonctionnement se repose principalement sur les fichiers `render.cpp` et `main.cpp`.

5.1 Le fichier main.cpp

5.1.1 Récupération du fichier de configuration

On commence d'abord par parser le fichier de configuration, contenant divers paramètres tels que les noms des fichiers audio à charger, le nombre d'entrées analogiques, etc. Si aucun argument n'est spécifié au programme, on charge le fichier `settings.cfg` sur le Bela. Sinon, l'argument fixe le fichier à charger. On récupère les données de ce fichier grâce à la classe `Parser`, et on stocke ces paramètres dans une structure `ChSettings`. Cette structure sera ensuite communiquée au thread audio.

On récupère notamment les fonctions de la libraire `libprocess.so` créée dans le dossier `process`, grâce à `dlopen`.

5.1.2 Initialisation des paramètres audio

Cette partie du main est semblable à celle du main par défaut. Elle sert à initialiser le thread audio via les fonctions `Bela_initAudio()`, `Bela_defaultSettings()`, `Bela_startAudio()` et tous les traitements faits autour de ces fonctions. Voir un exemple de main classique dans un autre projet ici. On commence par initialiser la variable `settings` qui contient les paramètres audio (nombre de canaux analogiques, etc). On l'initialise avec les paramètres par défaut avec `Bela_defaultSettings()`, puis on règle le nombre de canaux analogiques comme définis dans le fichier de configuration.

Dans le main par défaut, `settings` est modifié grâce aux paramètres en ligne de commande. Pour éviter de modifier en dur cette variable, et risquer d'oublier un paramètre, nous avons créé des variables `argc` et `argv` similaires aux arguments du main, avec les paramètres à modifier. ces variables sont ensuite passées en arguments

de `Bela_getopt_long()`, qui parse ces variables et modifie `settings` en conséquence. En l'occurence, nous spécifions le nombre de pistes analogiques à utiliser (-C 4 ou 8), ainsi que celles à activer (-Y).

Enfin, la fonction `Bela_initAudio()` récupère les paramètres via la variable `settings`. Ces paramètres seront utilisables dans les fonctions de `render.cpp` via l'argument `userData`.

5.2 Le fichier render.cpp

5.2.1 La fonction setup()

La fonction récupère les paramètres passés par le main via l'argument `userData`, et les stocke dans des variables globales. Elle initialise notamment les structures permettant de lire les fichiers wav, ainsi que les buffers pour stocker les echantillons et les traiter ensuite.

5.2.2 La fonction render()

La fonction `render` suit globalement le schéma suivant :

```
pour chaque frame de données :
    avancer le curseur des fichiers de 1
    lancer le calcul de la matrice si les buffers sont remplis
    si effets activés:
        boucle effets
    sinon
        boucle normale
    ecrire dans la sortie audio
```

Les boucles d'effets et normale seront expliquées plus en détail ensuite, mais consistent grossièrement à lire dans les entrées standards (fichiers, analogiques et audio) les échantillons arrivant, et les stocker dans une matrice de taille $nb\_pistes * taille\_buffer$, représentant les pistes à traiter. La différence entre les deux boucles est que la boucle d'effets contient un traitement supplémentaire pour calculer les effets, tandis que la boucle normale stocke directement les échantillons.

6 Boucle de traitement de render.cpp

Après avoir expliqué le fonctionnement global de cette boucle, nous allons nous concentrer sur les détails, et expliquer chaque ligne de l'algorithme décrit en 5.2.2.

Tout d'abord, on répète la boucle "pour chaque bloc de données". En fait, à chaque appel de render, l'argument `context` contient des nouveaux buffers `audioIn` et `analogIn`, contenant les nouveaux échantillons. Ces buffers sont constitués de `frames`. Une frame correspond à un nombre d'échantillons égal au nombre de pistes, et représente chaque signal à un instant t. Les buffers sont constitués de plusieurs frames, il faut donc répéter le traitement de `render` autant de fois qu'il y a de frames.

6.1 Curseur des fichiers

Bien que les buffers analogiques et audio se mettent à jour automatiquement, ce n'est pas le cas des structures permettant de lire les fichiers, c'est à dire la classe `SampleStream`. Cette classe a été créée dans le programme d'exemple `sample-streamer-multi` et réutilisée dans ce projet. Elle dispose notamment d'une méthode `processFrame()`, permettant simplement d'avancer le curseur de lecture des fichiers. Il est nécessaire d'adapter la vitesse de lecture à celle des entrées physiques. La boucle dispose donc d'une instruction appelant la méthode `processFrame()` pour chaque piste.

6.2 Boucle normale

Cette partie correspond à la ligne "boucle normale". Elle consiste à récupérer les échantillons des différents canaux pour les stocker dans des buffers. Ces buffers sont stockés dans la structure suivante :

```
vector<vector<float> > gProcessBuffer;
```

La première dimension de cette matrice est le nombre de pistes, et chaque `gProcessBuffer[i]` est un vecteur représentant le signal audio, de taille `gUserSet.buffer_len` (initialisé dans `setup()` et correspondant à un paramètre du fichier de configuration). Les signaux sont stockés de la manière suivante :

— de 0 à $gNumStreams - 1$: fichiers wav
— de $gNumStreams$ à $gNumStreams + gNumAnalog - 1$: entrées analogiques
— de $gNumStream + gNumAnalog$ à la fin : entrées audio

6.3 Boucle d'effets

Le début de cette boucle est le même que la boucle normale : les échantillons sont stockés dans une matrice appelé `gEffectBufferIn`, de taille fixée par l'utilisateur (`EFFECT_BUFFER_LEN`). De plus, on a initialisé une autre matrice, `gEffectBufferOut`, de deuxième dimension deux fois plus grande. Ce sont les valeurs de cette matrice qui seront utilisés pour le calcul de la matrice de corrélation et la sortie audio. Ces buffers suivent le fonctionnement d'un buffer circulaire.

Les échantillons sont donc transferés de premier buffer vers le second, avec au milieu l'application de l'effet voulu.

On initialise aussi des curseurs :

— `gReadPointer` : entier pour savoir où mettre le prochain echantillon dans `gEffectBufferIn`
— `gWritePointer` : entier pour savoir quel est le prochain échantillon à lire de `gEffectBufferOut`
— `gLastSample` : dernier échantillon valide de `gEffectBufferOut`
— `gIndIn` : indice de `gEffectBufferIn` à partir duquel le signal doit être traité. Vaut généralement 0.
— `gIndOut` : premier indice de `gEffectBufferOut` non utiisé. On commencera à copier le signal traité ici.

Au premier transfert, l'algorithme est le suivant :

```
si gEffectBufferIn est plein :
    gEffectBufferIn.swap(gEffectBufferInCopy) //on libère BufferIn pour que les autres samples arrivent
    gWritePointer = 0; //en attendant le calcul, il faut lire les échantillons à partir de 0
    (on entend du silence pour l'instant)
    gIndOut = gEffSize; //on va ecrire le résultat du premier signal traité à cet indice.
    Bela_scheduleAuxiliaireTask(gEffectTask);
```

De cette manière, render va lire `gEffSize` échantillons vides. Pendant ce temps, la tâche sera executée, et se terminera avant que render ait lu `gEffSize` échantillons. Il lira donc les premiers signaux traités normalement.

Par la suite, l'algorithme fait en sorte de copier le résultat au début du buffer et à la fin alternativement. Pendant ce temps, le buffer est lu de manière circulaire, c'est à dire que lorsque le curseur de lecture arrive à la fin du buffer, il retourne au début, et ainsi de suite. Ceci permet de ne pas causer d'interruption ou de corruption du signal.

6.4 Calcul de la matrice

Cette partie de l'algorithme se rapporte au calcul et à l'envoi de la matrice. Elle s'explique comme ceci :

```
si gProcessBuffer est plein :
    copier(gProcessBuffer, gProcessBufferCopy);
    reset(gProcessBuffer);
    lancerTacheAuxiliaire(gProcessBufferTask);
```

L'instruction de copie est en fait un `swap` de deux vecteurs, qui echange les adresses et se fait en temps constant. Reset revient simplement à réinitialiser le curseur de position du buffer. Enfin, la tâche auxiliaire est lancée via `Bela_scheduleAuxiliaryTask(gProcessBufferTask)`, où `gProcessBufferTask` lance une fonction `processBuffer()` définie préalablement, et appelant les fonctions de calcul de matrice dont nous parlerons dans la partie sur les classes `ProcessMulti`.

6.5 Sortie audio

Pour entendre le résultat final, il est nécessaire d'écrire la somme de ces signaux dans une sortie audio. Pour ce faire, il existe dans la boucle normale et la boucle d'effet une variable flottante `out`. Cette variable sert à stocker la somme des signaux à mesure qu'on les récupère.

A la fin de la boucle, on écrit la valeur de `out` dans les 2 sorties audio, pour avoir une sortie stéréo. On utilise pour cela la fonction `audioWrite` détaillée dans la documentation.

6.6 Adaptation du taux d'échantillonnage

Enfin, vous pourrez remarquer dans le code, la présence de certaines instructions `if (gSampleFactor == STANDARD_SAMPLE_RATE)` . Ces instructions correspondent à de petits changements nécessaires lorsque le taux d'échantillonnage des entrées analogiques varie.

En effet, Bela impose d'utiliser soit 4 sorties analogiques à 44100 Hz, soit 8 sorties analogiques à 22050 Hz. Dans le premier cas, il n'y a pas de problème. Mais dans le deuxième cas, des modifications sont nécessaires. En effet, les entrées audio classiques ainsi que les fichiers ne peuvent pas choisir leur fréquence d'échantillonnage, celle ci étant fixée à 44100 Hz. Il est donc nécessaire d'adapter artificiellement les entrées audio et fichiers. Dans les deux cas, cela revient à récupérer un échantillon sur deux.

Pour gérer les fichiers, il est nécessaire de faire défiler le curseur 2 fois plus vite : ceci explique la présence d'une deuxième instruction `processFrame()` dans le cas ou on est à 22050 Hz, qui bouge le curseur une fois supplémentaire.

Pour les entrées audio, il faut simplement lire une fois sur deux, d'où le `audioRead(context, 2*n, a)` à 22050 Hz.

Enfin, la sortie audio est elle-même échantillonnée à 44100 Hz. Lorsqu'on choisit un taux à 22050 Hz, il faut écrire deux fois plus d'échantillons que ce qu'on a dans les buffers, de manière à rééchantillonner artificiellement. On écrit donc 2 fois de suite la même valeur en sortie.

Ces modifications permettent donc à l'utilisateur de choisir le nombre de pistes analogiques dont il a besoin sans se préoccuper du traitement. Lorsqu'il choisit entre 0 et 4 entrées analogiques, `gSampleFactor` vaut 2, ce qui correspond à 44100 Hz. Si il choisit 5 entrées ou plus, `gSampleFactor` vaut 1 pour 22050 Hz.

6.7 Autres remarques

Il existe aussi avant la boucle principale, une instruction `Bela_scheduleAuxiliaryTask(gFillBuffersTask)`. Celle ci permet de mettre à jour les buffers liés aux fichiers si besoin. Pour tenter de gagner de la rapidité, nous avons rajouté un compteur fixé à 10, permettant d'éviter d'exécuter cette tâche trop souvent. Cependant l'instruction `if` entourant cette tâche peut être supprimée sans problème.

7 Fichier de configuration

Cette section détaille le fonctionnement du fichier de configuration, ou plus justement du parseur de ce fichier. Comme dit dans la partie sur le fonctionnement du main, ce parseur est appelé à son début.

7.1 Fichier parse.cpp

Nous avons tout d'abord créé un certain nombre de fonctions de parsing réutilisables dans d'autres cas. Ces fonctions sont placées dans le fichier `parse.cpp` :

- des fonctions `is_number(), is_ip(), is_name()` ... permettant de tester un `string`.
- des fonctions `get_next_*`, prenant en paramètre en itérateur de `string`, le modifiant et retournant une `string`.

On utilisera plus tard la fonction `get_next_word()` fonctionnant comme ceci :

```
string str = "hello world";
string::iterator it = str.begin();
cout << get_next_word(&it); //retourne "hello"
cout << get_next(&it); //retourne " " (espace)
cout << get_next_word(&it); //retourne "world"
```

7.2 Fichier Parser.cpp

Cette classe possède des attributs privés correspondant aux paramètres à récupérer dans le fichier, des getters, ainsi qu'une méthode `get_word()` et un constructeur.

`get_word()` prend en paramètre un `string` et renvoie le deuxième mot.

Enfin, le constructeur ouvre le fichier de configuration, le parcourt ligne par ligne, puis, à l'aide de la méthode précédente, stocke les paramètres de configuration.

7.3 Configuration à l'exécution dans le main

Comme expliqué dans la partie sur le main, le parseur récupère les paramètres du fichier de configuration à l'exécution. Ces paramètres sont ensuite stockés et interprétés dans le main puis utilisés dans `render.cpp`.

8 Classes ProcessMulti

Cette section détaille le fonctionnement des classes permettant le calcul de matrices, et plus généralement le traitement des buffers.

8.1 Interface

Lorsqu'on ordonnance la tâche de calcul de la matrice comme expliqué dans la partie sur `render.cpp`, la fonction exécutée par celle ci est la suivante :

```
ProcessMulti * p;
void processBuffer(){
    if (gBufferProcessed == 0){
        p->process(gProcessBufferCopy, gUserSet.conn);
        gBufferProcessed = 1;
    }
}
```

Tout le calcul se fait donc par cette méthode `process()` de la classe `ProcessMulti`. Cette classe est en fait une interface contenant uniquement cette méthode et un destructeur virtuel.

Les classes implémentant cette interface seront passées par polymorphisme au programme.

La méthode `process()` prend en argument la matrice contenant les signaux, ainsi qu'un attribut de la classe `Connection`, permettant de communiquer avec le page web. Cette classe sera détaillée dans la section à son nom. Cependant cet attribut n'est pas toujours nécessaire : par exemple, la classe `ProcessMultiWriteWav` qui implémente l'interface n'utilise pas cet attribut : elle se contente de copier les signaux dans un fichier wav.

Cependant la classe fille la plus utile est la classe `ProcessMultiCorrel`.

8.2 ProcessMultiCorrel

Cette classe permet véritablement le calcul de la matrice. Son constructeur prend en paramètre 3 pointeurs de fonctions, qui correspondent aux fonctions que l'on peut ajouter dans la librairie `process/libprocess.so` :

— une fonction de preprocessing
— une fonction de calcul de correlation
— une fonction associant un coefficient à une couleur.

La méthode `process()` execute alors ces fonctions à la suite, afin d'obtenir une matrice de triplets correspondant à des couleurs. Cette matrice est alors envoyée via une socket au serveur web (nous détaillerons ce point dans les sections dédiées).

8.2.1 Fichier log

De plus, la méthode enregistre les valeurs des coefficients de correlation au fur et à mesure dans le fichier `log/log`.

Un fichier log se présente sous cette forme :

```
BUFFERSIZE : X SAMPLE_RATE : Y
m[1][0] m[2][0] m[2][1] m[3][0] m[3][1] m[3][2] ... m[n][n-1]
m[1][0] m[2][0] m[2][1] m[3][0] m[3][1] m[3][2] ... m[n][n-1]
m[1][0] m[2][0] m[2][1] m[3][0] m[3][1] m[3][2] ... m[n][n-1]
```

Chaque ligne de coefficient représente une matrice de corrélation. Pour éviter de stocker l'information redondante, on ne stocke que la partie basse de la matrice, sans la diagonale.

9 Choix des process à l'execution

Nous avons de plus ajouté la possibilité de choisir les fonctions précédentes à l'exécution.

9.1 Création de la libraire libprocess.so

La librairie `libprocess.so` est une bibliothèque dynamique stockant les fonctions créées par l'utilisateur. Sous réserve que ces fonctions soient correctes et correspondent aux standards exigés dans le tutoriel du début de cette documentation, cette librairie se compile via un simple Makefile présent dans le dossier. Il compile l'ensemble des fichiers concernés :

```
g++ -shared -O3  -fPIC Preproc*.cpp Coeff*.cpp Color*.cpp -o libprocess.so
```

Les options `fPIC` et `shared` permettent de créer le .so, et l'option -O3 permet de compiler avec le plus d'optimisations possibles. Sans cette option, le programme est beaucoup plus lent.

9.2 Linkage dynamique avec dlopen

Les fonctions définies dans le fichier de configuration sont ensuite chargées dans le main avec `dlopen`. Cette librairie très simple d'utilisation permet de charger dynamiquement les fonctions d'une librairie.

Pour que les noms de symboles correspondent aux noms de fonctions, il est nécessaire que les fonctions soient implémentées avec `extern "C"` comme expliqué dans le tutoriel.

10 Serveur web

Cette section détaille le fonctionnement de la communication entre le programme et la page web.

10.1 Classe Connection

Comme expliqué plus haut, les méthodes `process()` prennent en argument une instance `Connection`. Cette classe permet la communication avec le serveur web via une socket TCP.

```
class Connection{}
private :
  int sockfd;
  bool _isConnected;
  int _port;
  std::string _addr;
public :
  Connection() : _port(12345), _addr("192.168.7.1") {}
  Connection(int port, std::string addr) : _port(port), _addr(addr){}
  bool isConnected();
  int init();
  int send(const std::string& msg);
  int end();
};
```

`sockfd` désigne la socket TCP, `_port` est le numéro de port et `_addr` l'IP. La connection est initialisée dans `init()` et terminée dans `end()`. Enfin, `send()` envoie un message à la socket. Le code de ce module est un code classique permettant de créer un client TCP en C. Ce client se connecte à l'adresse IP et au port précisé dans le fichier de configuration, à laquelle se connectera également le serveur TCP (voir serveur nodejs)

`Connection.cpp` contient aussi un code commenté pour établir une connection UDP avec le serveur. Cependant ceci risque d'être peu utile.

10.2 Serveur nodejs

Pour communiquer entre la page web et le programme, nous avons par la suite mis en place un serveur. Celui ci a été codé en javascript grâce à la plateforme `nodejs` et le module `socket.io`, permettant de créer des serveurs très facilement. Ceci a été implémenté dans le fichier `server.js`, présent sur Bela dans le dossier `~/Bela/IDE/public/VisualImproo`, et dans le dossier à placer sur l'ordinateur, selon la manière dont on veut lancer l'application. Pour gagner en performance, il est plus judicieux de lancer le serveur depuis l'ordinateur. En effet, Bela ne possédant qu'un seul processeur, le thread audio tend à accaparer toutes les ressources lorsqu'il y a beaucoup de pistes à traiter, diminuant la réactivité du serveur si celui-ci tourne sur le Bela.

Ce fichier contient en fait 2 serveurs :

— un serveur TCP, permettant la communication avec le client TCP du programme (voir le sous-chapitre précédent). Celui-ci se connecte sur le port 12345, à l'adresse `192.168.7.1` si il est sur l'ordinateur, et `127.0.0.1` (localhost) si il est sur le Bela

— un serveur HTTP, permettant la communication avec la page web. Celui-ci se connecte sur le port 12345, à l'adresse `192.168.7.1` si il est sur l'ordinateur, et `192.168.7.2` si il est sur le Bela

Lorsque le serveur TCP reçoit un message d'un de ses clients TCP, c'est à dire une chaine de caractère représentant une matrice, il la retransmet à tous les cients HTTP connectés.

Le fichier contient aussi l'implémentation d'un serveur UDP, qui a été commentée car inutile pour l'instant.

10.3 Page web

La page web contient des fonctions de création et de destruction de matrices en javascript. Ces "matrices" sont en fait des assemblages de balises `<tr>` et `<td>`, dont on a rempli le fond. La fonction javascript `createTable()` crée cette matrice en se basant sur les id `#rows` et `#columns` fixant les dimensions de la matrice. Cette matrice est créée sous la balise `<div id="matrixTableId">`. Il existe égalemnt une fonction permettant de la détruire.

Enfin, nous avons créé une fonction `updateMatrixFromSocket()`, prenant en argument un message représentant les couleurs de la matrice (décrit dans le sous chapitre suivant), et mettant à jour la matrice.

Enfin, la page web possède le script suivant :

```
<script>
  var socket = io.connect('http://192.168.7.1:8080');
  socket.on('message', function(message) {
  updateMatrixFromSocket(message);
  })
</script>
```

Ceci permet de se connecter au serveur web et de traiter chaque message arrivant.

10.4 Communication

A la fin de la méthode `process()` de `ProcessMultiCorrel`, la matrice de triplets doit être convertie en chaine de caractères, pour être communiquée au serveur. Ceci est fait via la fonction `matrixtostring()` implémentée dans `utilities.cpp`. Elle convertit la matrice comme ceci :

— un triplet (x,y,z) est converti en une chaine $\#xxyyzz$ de 7 caractères, où xx est la représentation hexadécimale de x, et ainsi de suite.

— la chaine finale est de la forme $m - nc_{11}c_{12}...c_{1n}c_{21}c_{22}...c_{2n}.....c_{m1}c_{m2}...c_{mn}$ où c_{ij} est la couleur ligne i colonne j, de la forme décrite au dessus. Vous pouvez visualiser ces chaines en décommentant la ligne `console.log('DATA ' + sock.remoteAddress + ': ' + data)` dans le serveur TCP.

11 Effets

Les effets sont très peu développés en l'état du programme, car si l'on veut les modifier il faut passer par le code et recompiler. Cependant il a été créé une interface `Effect`. Cette interface possède une méthode virtuelle pure `apply` ainsi qu'un destructeur virtuel.

Les effets sont passés en paramètres dans la structure `ChSettings` depuis le main, et récupérés comme les autres paramètres. La structure `ChSettings` contient trois `vector<Effect*>`, un pour les pistes analogiques, un pour les fichiers et un pour les pistes audio. Ils permettent d'associer à chaque piste un effet par polymorphisme.

12 Exécutable à distance

Enfin, nous avons créé un exécutable permettant de lancer le programme depuis l'ordinateur. Ce fichier suit la séquence suivante :

— d'abord, on récupère le fichier de configuration, qu'on modifie dans un fichier temporaire en modifiant le chemin de fichiers wav. Par exemple, si on a l'instruction `FILE test.wav`, on la remplace par `FILE ~/Bela/projects/VisualImpro/wavfiles/test.wav`, car ce sera le futur chemin de notre fichier wav. Ceci est fait par la classe `ExecParser`.
— on copie tous les fichiers wav nécessaires de l'ordinateur au Bela, au chemin indiqué précedemment
— on lance le serveur en tache de fond
— on ouvre une page web avec la matrice
— on lance le programme en ssh sur Bela

Enfin on souhaitait pouvoir tout stopper (le serveur et le programme Bela) en envoyant le signal `CTRL+C`. Nous avons donc mis au point un gestionnaire de signaux, qui stoppe le serveur, le programme Bela et supprime tous les fichiers wav du Bela ainsi que les fichiers temporaires créés. Enfin, celui ci récupère le `log` sur Bela et le sauve dans le dossier `logs` au nom correspondant à la date et l'heure de fin du programme.

13 Améliorations possibles

Pour terminer, nous allons évoquer quelques pistes d'amélioration du programme, ainsi que des idées pour les mettre en place.

13.1 Effets

La première possibilité serait de permettre de choisir ses effets à l'exécution, de la même manière qu'on choisit les fonctions de calcul matriciel. Pour cela, il faudrait utiliser les mêmes idées que pour la librairie `libprocess` :

— définir un prototype ou une classe standard d'effets. Cette étape devra être étudiée avec soin : il faudra penser notamment à la possibilité de règler les paramètres d'un effet (exemple : pour un effet de delay, on pourra régler l'amplitude et le retard) tout en gardant un standard
— créer un dossier pour les effets et un makefile pour créer la librairie
— modifier le fichier de configuration pour qu'il prenne en compte le choix des effets pour chaque piste
— utiliser `dlopen` dans le main de la même façon que pour `ProcessMulti`

13.2 Autres corrélations

Un autre champ d'étude est la création de nouvelles corrélations et fonctions de calcul. Voici quelques idées.

Pour le preprocessing :

— des fonctions permettant de ne garder qu'un sample sur 2,3,4... Cela réduira considérablement le temps de calcul de corrélation par la suite
— des fonctions de transformation de Fourier
— divers calcul d'enveloppes

Pour les corrélations :

— reconnaissance des notes et calcul de corrélations basées dessus
— reconnaissance du rythme

13.3 Autre sortie audio

Nous avons également abordé l'idée d'utiliser une autre sortie audio, dans le but d'emettre des informations selon le résultat des corrélations. Une sorte de "visualisation sonore" des corrélations.

Pour cela, il peut être judicieux d'utiliser un système de buffers circulaires, à la manière du traitement des effets, et de rajouter des fonctions aux classes `ProcessMulti` permettant de remplir ces buffers après le calcul de corrélations. Ensuite, ces buffers seront transmis à une sortie audio comme à la toute fin de la fonction `render()`.

13.4 Interface web

Enfin, il est possible de permettre une meilleure gestion des paramètres via une interface web plus poussée qu'actuellement, avec la possibilité de chosir la configuration au début, voire en cours de route. Pour cela, il

sera nécessaire de modifier la classe `Connection` en ajoutant une fonction `receive()`, ainsi que d'ajouter des fonction de parsing permettant d'associer un message reçu à une action. De plus, il faudrait complexifier le serveur nodejs ainsi que la page web pour remplacer le fichier de configuration par ceci. Pour la gestion en temps réel des paramètres, il faudra certainement ajouter une instruction `if (message reçu)` dans `render()`. Cette question nécessite évidemment un temps de reflexion important.