ENSEIRB-MATMECA

PFA - APPLICATION TACTILE POUR LES ARTS INTERACTIFS

Manuel d'Utilisation et de Maintenance

BEN ZINA Marwa
GAULIER Paul
GRATI Nour
MAURIN Julie
MERCIER Kevin
THIENOT Lucile
VIKSTROM Hugo

Responsable Pédagogique :
ROLLET Antoine
Client :
CELERIER Jean-Michaël

7 avril 2017

Table des matières

Première partie
Manuel d'utilisation

1 Plugin AppInteraction pour i-score

1.1 Lancer *i-score*

Pour lancer *i-score*, il faut utiliser la commande suivante dans le dossier contenant l'exécutable :

```
./i-score
```

La fenêtre suivante s'ouvre :

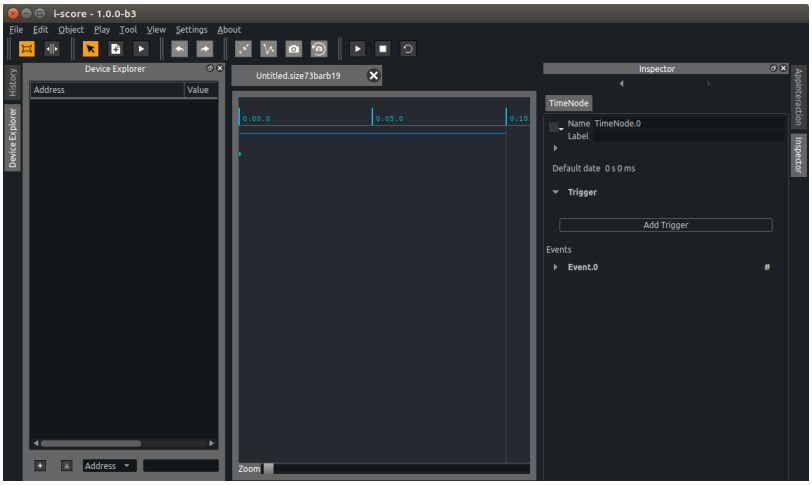

FIGURE 1 – Fenêtre principale

1.2 Utiliser *i-score*

Au sein du logiciel `i-score`, on peut créer différentes séquences d'interactions. `i-score` présente la possibilité de créer plusieurs types de `Process`, les lier à différents `Devices` et d'y appliquer toutes sortes de conditions.

Dans ce qui suit, on va s'intéresser à l'utilisation du `Process` développé au sein de notre PFA : `AppInteraction`. Ce `Process` est issu d'un plugin réalisé dans le but de créer la possibilité de réaliser une interaction via un appareil mobile.
Il est à noter que pour utiliser ce `Process` spécifique, il faut disposer de :

— Un ou plusieurs appareils mobiles (*Smartphone* ou *Tablette*)

— Un réseau Wi-Fi.

1.3 Ajouter une AppInteraction dans une partition

Pour l'ajout d'une `AppInteraction` dans un partition, on clique sur le point bleu dans la fenêtre centrale : une croix jaune apparaît. On clique dessus et on étire la ligne bleue créée : l'étirement de cette ligne définit la durée du `Process`. En sélectionnant cette ligne, une croix bleue apparaît au dessus. Il suffit alors de cliquer dessus pour sélectionner un `Process` à ajouter, comme le montre la figure 2.

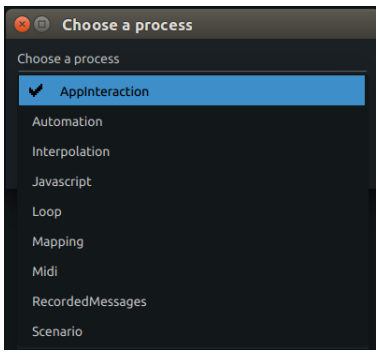

FIGURE 2 – Le menu pour faire le choix du `Process`

Si on sélectionne le premier choix, `AppInteraction`, le `Process` s'affichera comme suit :

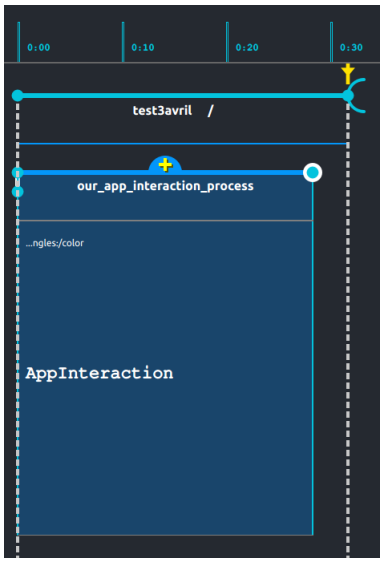

FIGURE 3 – Partition contenant uniquement un `Process AppInteraction`

1.4 Ajouter l'arborescence des `Devices`

On peut observer en bas du `Device Explorer`, sur le côté gauche du `panel` de *i-score*, une croix dans un carré (figure 4). Si on veut ajouter un `Device` ou une arborescence de `Devices`, il faut utiliser cette croix ou faire un clique droit dans le `Device Explorer`.

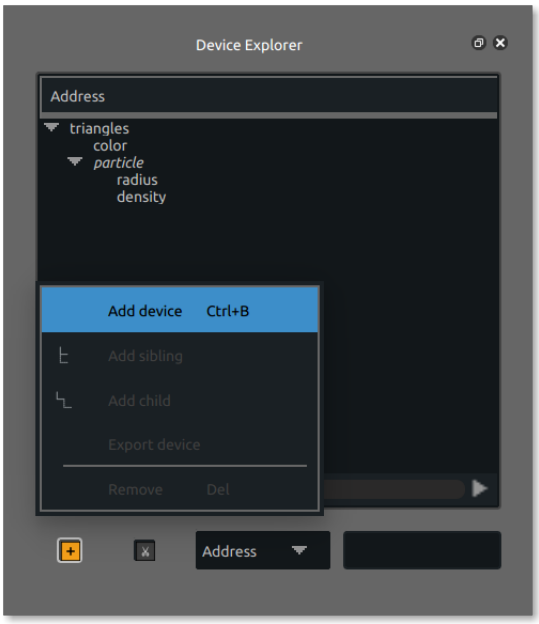

FIGURE 4 – Le `DeviceExplorer` de *i-score*

En appuyant ensuite sur `Add Device` (ou `Add Child` selon les cas), une fenêtre s'ouvre (voir figure 5). On peut alors procéder de deux manières pour ajouter le `Device`, en particulier s'il s'agit d'un OSCDevice :

— Manuellement en selectionnant les paramètres (protocole, ports,..) de ce `Device`. Voir figure 5

— Charger une arborescence en appuyant sur le bouton bleu `Load...` dans la fenêtre de la figure 5

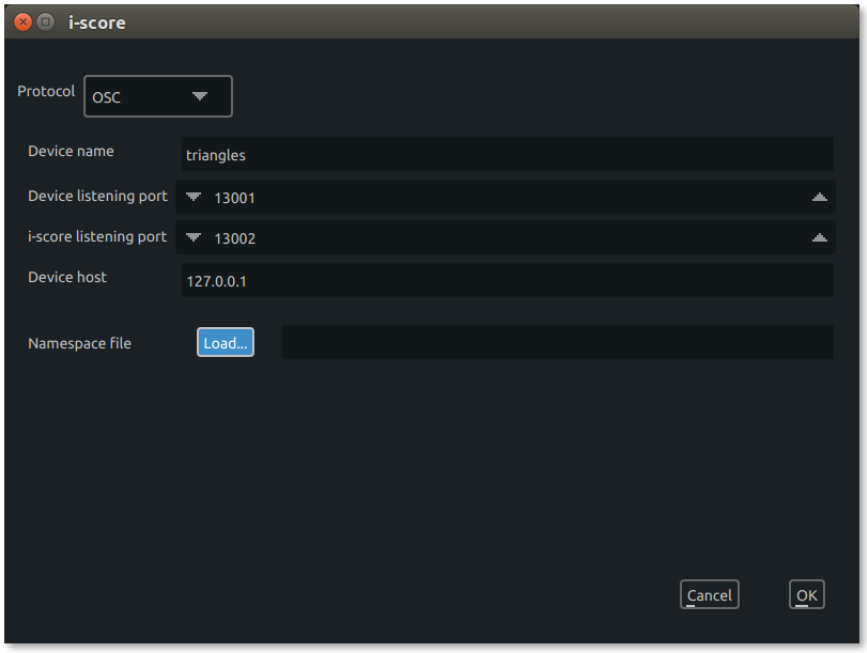

FIGURE 5 – Ajout d'un `OSCDevice`

1.5 Paramétrer l'AppInteraction

Le paramétrage de la fenêtre revient à la sélection des valeurs de certains champs dans le `panel` de l'inspecteur :

- `Interaction Type` : le type d'interaction à demander à l'application mobile ;
- `Mobile Device` : le nom de l'appareil mobile auquel est associé l'interaction ;
- `Address` : l'adresse du logiciel externe dont on veut changer des paramètres au cours de cette interaction ;
- `Min` et `Max` : définition de l' intervalle qui borne les valeurs envoyées au logiciel externe sélectionné.

La figure 6 présente un exemple de paramétrage d'une `AppInteraction`.

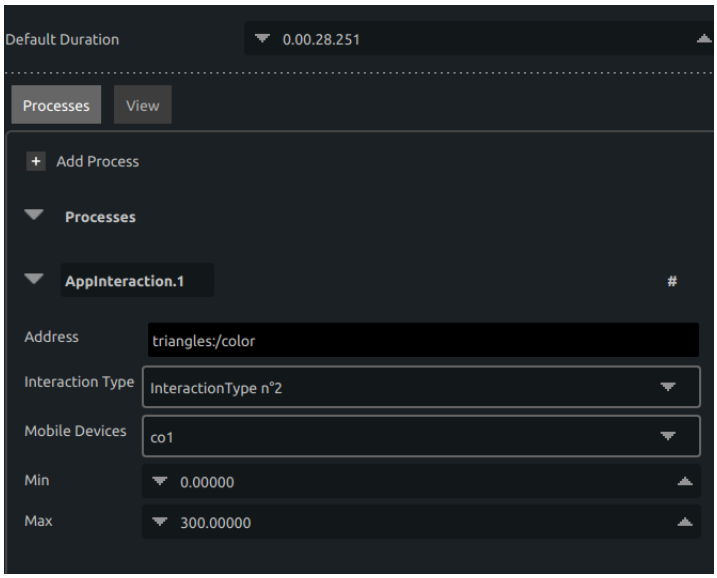

FIGURE 6 – Paramétrage d'une `AppInteraction` dans l'inspecteur

1.6 Jouer une partition contenant une AppInteraction

Pour jouer une partition contenant une `AppInteraction`, utilisons un fichier exemple de `Processing`, qui est un outil adapté à la création plastique et graphique interactive. Voici les étapes pas à pas :

— Ouvrir `Processing` et charger l'exemple fourni dans les sources de *i-score* plus précisement dans le répertoire *i-score/Documentation* sous le nom : `/Examples/Processing/Dataspace/dataspace/dataspace.pde` (voir figure 7)

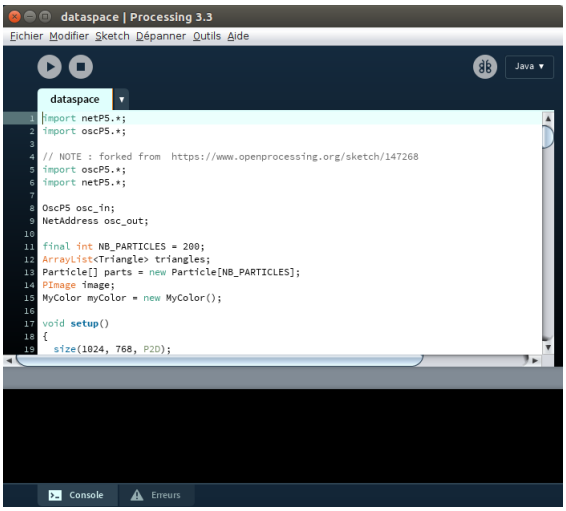

FIGURE 7 – Exemple chargé dans `Processing`

— S'assurer de la présence de la libraire `oscP5` dans `Sketch` puis `Importer une librairie`.

— Ouvrir *i-score* et charger la partition du fichier `test_dataspace_processing.scorejson`. Il est disponible sur le dépôt dans `addon/fichiers_tests/`.

— Appuyer sur le bouton `run` du `Processing`. La fenêtre *dataspace* s'ouvre comme on le voit en figure 8.

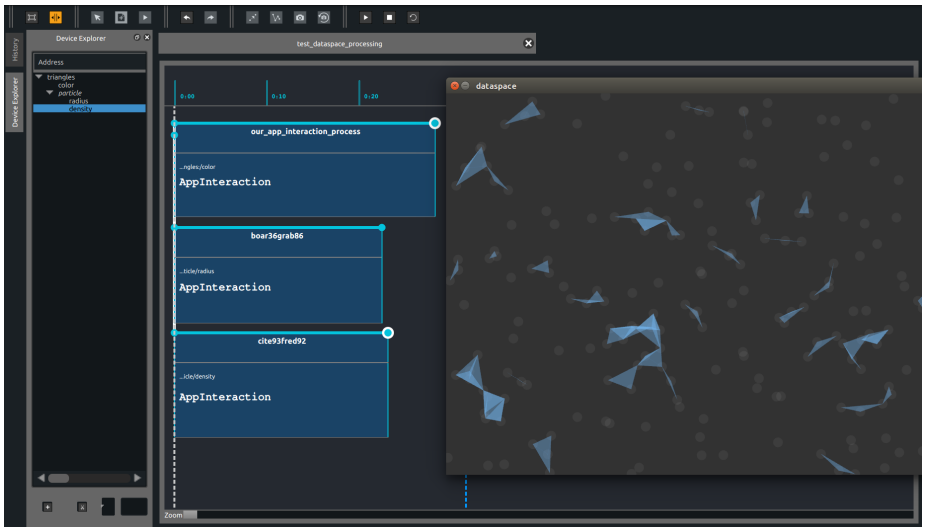

FIGURE 8 – Partition *i-score* et affichage de Processing avant lecture de la partition

— Appuyer sur le bouton `run` en haut à droite de *i-score*. On observe alors la couleur verte se propager sur la ligne temporelle au dessus des `Process` de la partition (voir figure 9). Sur la fenêtre de Processing, on observe suite au lancement de cette interaction une modification de la densité de population des triangles affichés, de leur couleur et des rayons des particules grises du fond.

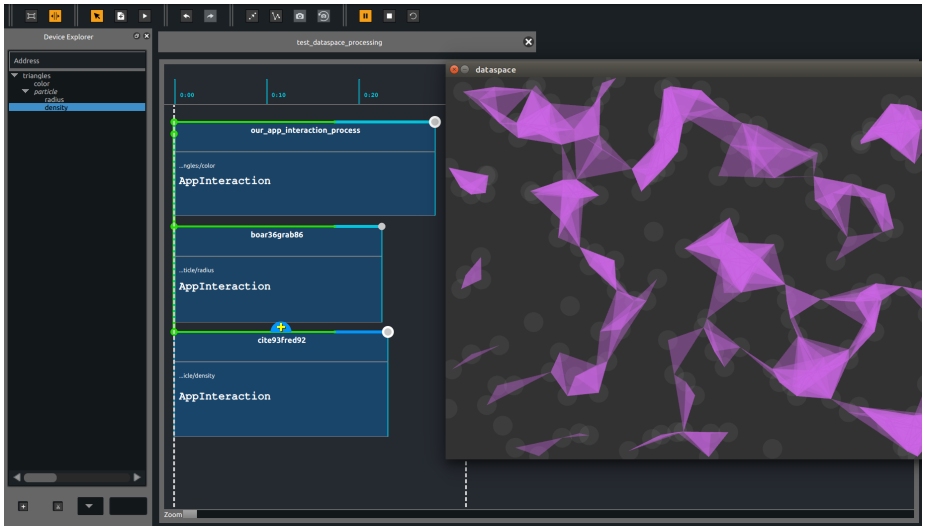

FIGURE 9 – Lecture de la partition et affichage de Processing modifié

Aujourd'hui, les faussaires permettent de créer de fausses connexions, et d'envoyer une valeur lorsqu'une partition est jouée. On peut ainsi modifier la couleur ou encore la densité des formes géométriques d'une exécution du logiciel `Processing`. On obtient comme ceci une simulation du comportement du plugin comme s'il disposait d'une connexion réelle correctement établie.

Cependant, ils ne permettent d'envoyer qu'une seule valeur par `Process`. En effet, la méthode `sendInteraction()` est appelée par `ProcessExecutor.start()`, or `sendInteraction()` ne retourne qu'*après* avoir transmis toutes les valeurs générées. Par conséquent, l'exécuteur reste « bloqué » dans `start()` jusqu'à ce que `sendInteraction()` retourne : l'exécuteur ne traitera alors que la dernière valeur reçue.

2 Application

2.1 Lancer l'application

Lors du lancement de l'application un écran de connexion s'affiche.

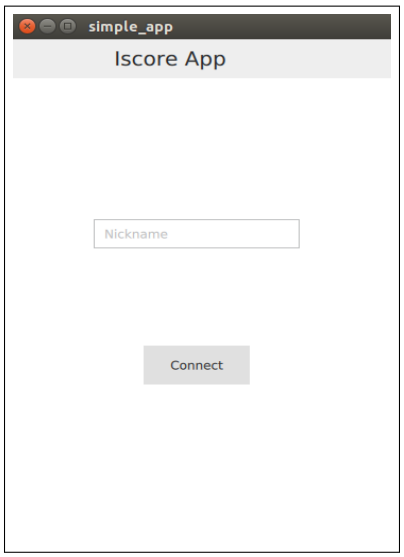

FIGURE 10 – Écran de connexion

Indiquez votre surnom et appuyez sur le bouton **connect**. Un menu s'affichera (figure 11) contenant les différentes interactions possibles dans l'application.

Sélectionnez l'icône d'une interaction pour lire sa description. Vous pouvez faire une recherche par nom aussi.

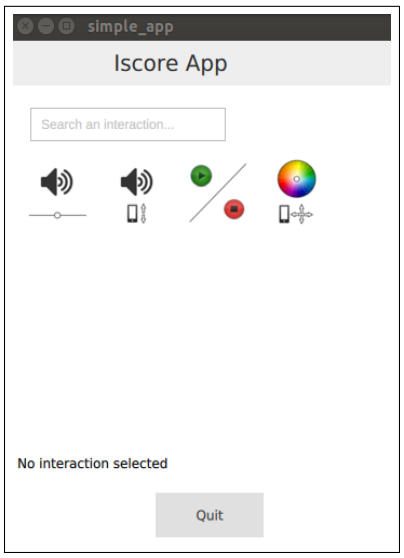

FIGURE 11 – Menu d'interactions

2.2 Effectuer une interaction

Au moment où il y a une interaction à effectuer, une fenêtre apparaît contenant la description de l'interaction et un compteur du temps restant avant que l'interaction commence.

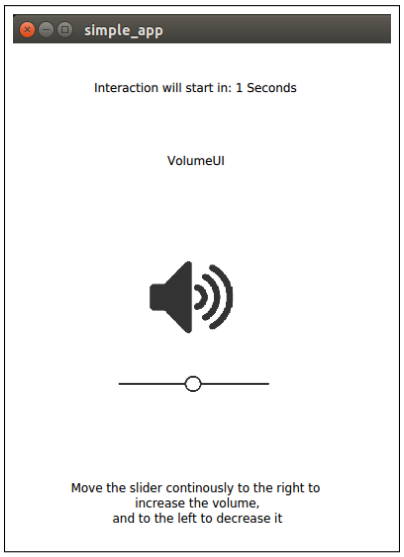

FIGURE 12 – Écran d'attente pour l'interaction VolumeUI

Lorsque le compteur s'annule, l'IHM de l'interaction apparaît automatiquement. Il suffit de suivre les instructions indiquées. Par exemple, dans le cas de l'interaction VolumeUI il suffit d'agir sur le slider en modifiant sa valeur de manière continue.

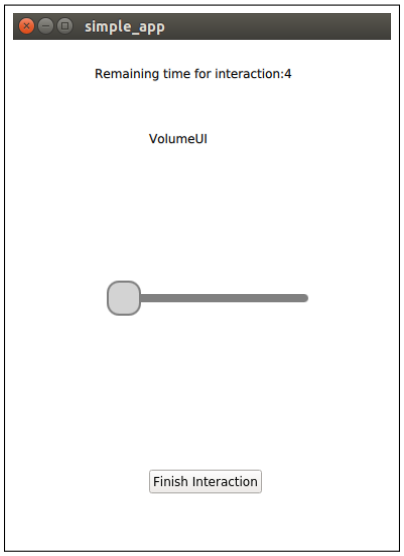

FIGURE 13 – IHM de l'interaction VolumeUI

Deuxième partie

Manuel de maintenance

Le manuel de maintenance se décompose en trois parties distinctes : le plugin, l'application, puis la connexion entre ces deux entités.

3 Plugin AppInteraction pour i-score

3.1 Compiler le plugin

3.1.1 Installer *i-score*

Le lien suivant contient les instructions à suivre pour installer et compiler *i-score* : `www.github.com/OSSIA/i-score/wiki/Build-and-install`.

3.1.2 Récupérer les sources

Pour intégrer un plugin à *i-score*, il faut positionner ses sources dans le répertoire `base/addons/`. Pour notre plugin, le chemin complet du fichier `CMakeLists.txt` est le suivant :
`i-score/base/addons/iscore-addon-app-interaction/CMakeLists.txt`

3.1.3 Recompiler *i-score*

Il suffit alors de recompiler *i-score* pour que le plugin y soit automatiquement intégré. On peut voir apparaître le nouveau `Process` appelé `AppInteraction` en exécutant *i-score*, comme mentionné dans le manuel d'utilisation (partie 1).

3.2 Architecture du plugin

3.2.1 Introduction au logiciel *i-score*

Le logiciel *i-score* est un séquenceur libre et open-source qui permet à l'utilisateur de créer des partitions musicales ou sonores par exemple. Il est écrit en C++ et utilise notamment l'API OSSIA et le framework Qt.

i-score permet à l'utilisateur de connecter d'autres logiciels au séquenceur afin de créer et modifier ses productions artistiques. Nous avons par exemple pu utiliser le logiciel Processing, qui est un outil adapté à la création plastique et graphique interactive. La figure 14 donne un aperçu du visuel obtenu lors du lancement du programme fourni comme exemple dans les sources d'*i-score*.

Ces logiciels annexes sont appelés *devices* dans le code d'*i-score*. Ils sont référencés sous la forme d'un arbre dont les feuilles sont les paramètres modifiables de ces *devices*. Cet arbre est affiché dans la fenêtre `Device Explorer`, automatiquement affichée sur la gauche au lancement d'*i-score*.

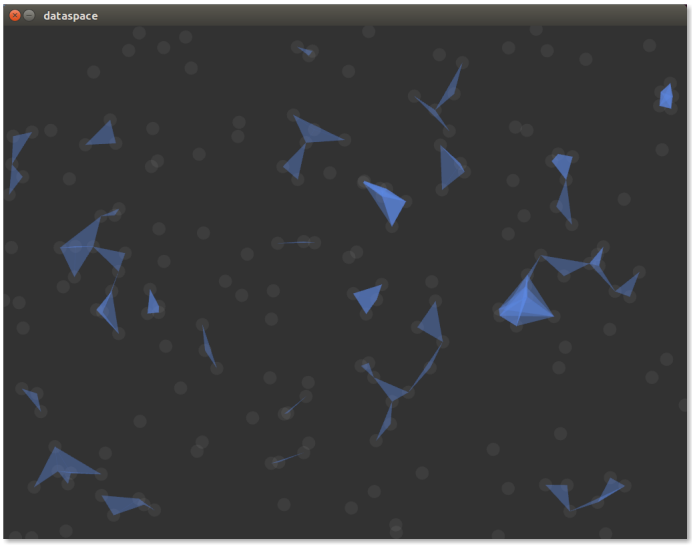

FIGURE 14 – Rendu visuel du programme `Documentation/Examples/Processing/Dataspace/dataspace/dataspace.pde`

Sont également implémentés dans *i-score* plusieurs `Process`, qui sont des éléments possédant notamment une date de début et de fin et qui permettent de modifier des paramètres d'un *device* donné. Tous les `Process` peuvent être ajoutés dans une partition, qui permet de les ordonner et d'y appliquer toutes sortes de conditions. La partition est éditée par le compositeur, qui peut également la lire, la mettre en pause ou la stopper, afin d'exécuter les `Process`.

L'`Automation` est un exemple de `Process` qui associe une courbe à un paramètre donné d'un *device*, courbe qui représente l'évolution de la valeur de ce paramètre pendant la lecture du `Process`. Le plugin qui fait l'objet de ce projet implémente un nouveau `Process`, qui peut modifier un *device* en fonction des valeurs reçues depuis notre application.

3.2.2 Structure générale

Arborescence La figure 15 donne une vue globale de l'architecture du plugin.

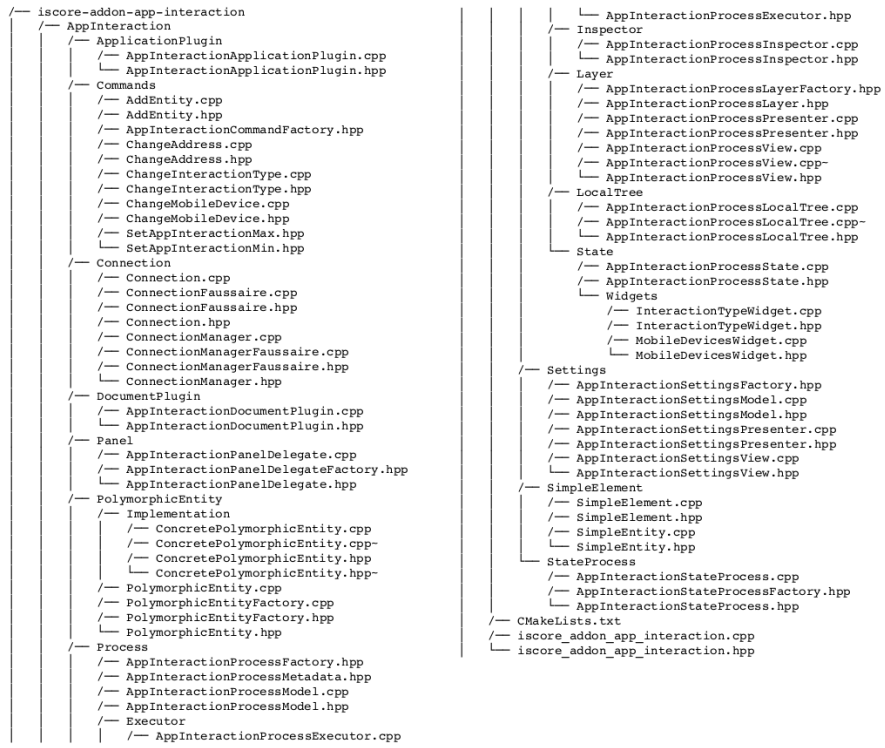

FIGURE 15 – Arborescence des fichiers sources plugin

CMake CMake est le moteur de production utilisé par *i-score*. Toutes les informations de configuration de la construction logicielle se trouvent dans les fichiers `CMakeLists.txt`, qui sont donc automatiquement détectés et utilisés par CMake.

Il existe une commande spécifique nommée `setup_iscore_plugin` à utiliser dans le fichier CMakeLists.txt de notre plugin pour s'assurer que ce dernier fait son build avec les mêmes options de compilation que *i-score* (static vs dynamic, precompiled headers, link-time optimization, etc.)

Qt plugin system Un plugin *i-score* est basé sur le système de plugin Qt. L'essentiel à retenir en ce qui concerne ce système :

— Le plugin Qt est généralement une classe nommée comme `i-score-addon--app-interaction`, à la racine du plugin. C'est le point d'entrée pour le charger.

— Cette classe ne fournit que des *factories* enregistrées dans le programme et peut instancier directement une instance accessible partout dans l'application via l'appel de la méthode `make_applicationPlugin()`.

— Les autres classes sont instanciées soit via `factoryFamilies` (s'il s'agit des abstract factories) soit par `factories` (s'il s'agit des factories concrètes) ou par `make_commands` (pour fournir les commandes *undo/redo*).

3.2.3 Concepts fondamentaux

Object Model *i-score* est basé sur un modèle d'objet hiérarchique. Il faut faire attention de ne pas confondre les hiérarchies de type (lorsque B hérite de A) et les hiérarchies d'objet (lorsque A a des objets enfants de type B).

Ce modèle d'objet repose sur le modèle d'objet de Qt (*QObject*), mais ajoute un identifiant, ou UUID, à chaque objet afin qu'il puisse être retrouvé facilement. Cet identifiant est unique à un niveau hiérarchique donné, pour un nom d'objet donné, et peut être facilement généré dans un terminal par la commande `uuidgen`.

Il existe ensuite quatre sortes d'objets principaux :

— Les classes concrètes héritant de `IdentifiedObject` : Ces classes sont les objets les plus simples. Par exemple, une note (*Midi : : Note*) ou un point dans une courbe (*Curve : : PointModel*).

— Les classes concrètes héritant de `Entity` : Ces classes sont des objets de modèle plus complexes, avec les fonctionnalités suivantes :

  — La capacité d'héberger des composants. Voir *iscore : :Component*.

  — La gestion des metadonnées en temps réel (nom, étiquette, commentaire, couleur ...). Voir *iscore : :ModelMetadata*.

— Les classes abstraites héritant de `IdentifiedObject` : Classes simples qui ont un comportement polymorphe. Par exemple, dans *Curve : :SegmentModel*, les différents types de segments sont implémentés par héritage.

— Classes abstraites héritées de `Entity` : Objets complexes pouvant comporter des composants (components) et des sous-classes. Par exemple, `Process` : `:ProcessModel`.

Les classes abstraites sont les plus complexes :

— Il doit y avoir un *factory* pour les instancier ;

— Ils doivent hériter de `iscore::SerializableInterface` afin de fournir la méthode virtuelle *save*.

Ces classes peuvent être placées dans `EntityMap` : un conteneur spécial qui envoie une notification chaque fois qu'un élément est inséré ou supprimé. Pour en savoir plus, notre `AppInteraction::ProcessModel` contient un exemple d'usage de *EntityMap*.

Process Certains `Process` existent déjà (comme `scenario`, `automation`, `JS`). Comme expliqué en partie 3.2.1, `AppInteraction` est un nouveau `Process`.

Le répertoire `Process/` est le répertoire principalement utilisé lors de la conception d'un plugin, puisqu'il contient les sous-répertoires qui permettent de manipuler la plupart des concepts fondamentaux expliqués ci-dessous.

Panels Les *Panels* sont les objets graphiques à gauche, à droite et en bas de la vue principale du document.

On peut fournir de nouveaux *Panel* via la classe `iscore::PanelDelegate`.

State `State` est une des bibliothèques fondamentales d'*i-score*. Elle établit les concepts de `Value`, d'`Address` et de `Message` qui sont très utilisés dans le logiciel.

Context Dans *i-score*, le logiciel a plusieurs niveaux de contextes :

- Contexte général de l'application(**Application-wide**) : l'objet est accessible partout dans l'application *i-score*. C'est plus ou moins le modèle *Singleton*. Par exemple, les paramètres de l'utilisateur (*user settings*), le *skin*, les *panels*, ...
- Contexte général d'un document (**Document-wide**) : L'objet est accessible partout dans le document. Par exemple, la pile de commandes *undo/redo* ou encore l'instanciation de ConnectionManager dans `AppInteraction::DocumentPlugin` pour le rendre accessible dans tout notre plugin.
- Contexte spécifique pour certains objets .

Les contextes sont accessibles selon les cas depuis `iscore::ApplicationContext`, `iscore::DocumentContext` ou d'autre classes spécifiques.
Toutes les classes enregistrées via les plugins sont accessibles depuis le contexte d'application (`ApplicationContext`).

Par exemple, pour accéder à `AppInteraction::ApplicationPlugin`, il est possible d'écrire :

```
auto& appInteraction =
    iscore::AppContext().applicationPlugin<AppInteraction
                                           ::ApplicationPlugin>();
```

Pour accéder à `AppInteraction::DocumentPlugin`, il est nécessaire d'avoir un `DocumentContext` (puisqu'un document peut l'avoir ou pas). Ces contextes peuvent être obtenus par tout type d'objet appartenant à une hiérarchie de documents passant par `iscore::IDocument::documentContext(anObject)`. Voir *"Object model"*.

Inspector L'*Inspector* correspond au menu permettant de paramétrer un `Process`. On retrouve donc dans le dossier `Inspector` les fichiers concernant l'interface utilisateur liée au `Process AppInteraction`. Á l'ouverture d'*i-score*, l'inspecteur se trouve dans le `panel` situé sur la droite. En effet, chaque `Process` a des caractéristiques différentes donc il faut spécifier les éléments appartenant à l'interface de chaque `Process` indépendamment.

Commands Le concept de *commands* dans notre plugin désigne les commandes de modification du `Process`. On y trouve principalement : `ChangeAddress` , `ChangeInteractionType`, `ChangeMobileDevice`, `SetAppInteractionMin` et `SetAppInteractionMax`.

Pour mieux comprendre ce concept prenons l'exemple de `ChangeAddress.cpp`. La « modification du `Process` » ici implémentée est celle de l'adresse du logiciel externe dont on veut changer un paramètre. La modification de l'adresse peut être effectuée par l'utilisateur soit en tapant la nouvelle adresse au clavier dans le champ *Address* de notre inspecteur, soit en effectuant un glisser-déposer. Une commande qui prenne en charge ce changement est donc nécessaire, d'où la classe `ChangeAddress`.

Cette classe est déclarée comme commande dès le header, puisqu'elle hérite de la classe `i-score::Command`. On trouve également dans le fichier source les méthodes `undo` et `redo`, qui permettent respectivement d'annuler le dernier changement et de ré-effectuer le dernier changement. C'est entre autres dans le but d'avoir accès à ces deux méthodes et aux raccourcis clavier associés que chaque commande d'*i-score* doit posséder sa propre classe.

En plus des fonctions `undo` et `redo` énoncées précédemment, des fonctions de sérialisation et désérialisation sont implémentées. Celles-ci permettent notamment de stocker les valeurs des attributs de la classe lors de la sauvegarde d'un fichier, ou de récupérer les valeurs de ces attributs lors du chargement d'un fichier dans *i-score*.

Widget Les *widgets* sont des classes qui héritent toujours de `QWidget` (directement ou indirectement). Les *widgets*, dans notre cas d'utilisation, correspondent à des menus déroulants, auxquels s'ajoutent toutes les données qui y sont conservées. Ainsi, le widget `InteractionTypeWidget` possède un attribut de type `QComboBox`, qui est un objet Qt correspondant à un menu déroulant. Ce widget possède également un vecteur de chaînes de caractères, qui correspond à tous les items sélectionnables du menu déroulant, et un objet de type `QHBoxLayout` qui permet de paramétrer le rendu visuel du menu afin qu'il soit cohérent avec le reste du logiciel. Sont également implémentés un getter et un setter permettant de récupérer et modifier l'indice du type d'interaction sélectionné, cet indice étant un attribut de l'objet `QComboBox`.

Executor On parle ici du contenu du répertoire `Executor/`, sous-dossier de `Process/` et qui regroupe les fonctions utilisées lors de l'exécution du `Process`. Son implémentation est constituée de méthodes telles que `start()`, `pause()` ou encore `stop()` décrivant si nécessaire des actions spécifiques à suivre lorsque le `Process` est joué ou bien mis en pause. Ces méthodes sont en réalité des implémentations des méthodes de la classe mère de l'`Executor` appelée `ossia::time_process`.

Connection Au sein de notre plugin, le répertoire `Connection/` permet la gestion la connexion entre le plugin et l'application mobile. Cette connexion est

gérée principalement par la classe `ConnectionManager`, qui permet d'obtenir le nombre d'appareils connectés ou encore la liste de ces appareils. `ConnectionManager` possède un vecteur de pointeurs vers des instances de `Connection`, qui décrivent chacun une connexion avec un appareil mobile : le nom de l'appareil et la fonction permettant d'envoyer la demande d'interaction notamment.

Les classes `MockConnection` et `MockConnectionManager` ont quant à elles été implémentées pour assurer nos tests. Ces classes permettent de simuler la présence d'appareils sans qu'il y ait de réelles connexions établies.

Serialization Il existe une méthode de sérialisation unifiée par l'ensemble de *i-score*.

Pour éviter l'encombrement inutile des fichiers sources, le code de `Serialization` est présent dans `[ClassName]Serialization.cpp` : grâce à l'utilisation du *template*, aucun *header* n'est nécessaire.

Pour l'instant, il existe trois façons de sérialiser un objet :

— En tant que binaire, dans un `QDataStream` (partout) ;

— En tant que `QJsonObject` (la plupart des objets) ;

— En tant que `QJsonValue` (pour les types de valeur où il y a généralement un seul membre).

3.3 Apporter des modifications au plugin

3.3.1 Observations générales

— Notez que chaque ajout de fichier induit une modification du CMakeLists.txt racine du plugin.

— Les instructions `qDebug()` fonctionnent à la manière de la fonction `printf()`. Elles permettent l'affichage de messages d'erreur utiles au débugage et aux tests.

— Chaque attribut que vous verrez dans les classes du plugin qui soit de type `int` et qui fasse référence à un paramètre du `Process`, comme `m_interactionType` ou `m_mobileDevice`, correspond à l'indice de l'élément sélectionné par l'utilisateur. Par exemple, `m_interactionType` correspond à l'indice de l'élément sur lequel est arrêté le menu déroulant de l'inspecteur listant les différentes IHM d'interaction.

— Pour sauvegarder les valeurs sélectionnées dans un fichier, il faut utiliser les fonctions de `Serialization` de notre `ProcessModel`. Comme mentionné dans la partie 3.2.3, ceci peut être fait par `QDataStream` (voir figure 16) ou via `JSON` (voir figure 17).

```
192
193     // Save a simple data member
194     m_stream << proc.address();
195     m_stream << proc.interactionType();
196     m_stream << proc.mobileDevice();
197     m_stream << proc.min();
198     m_stream << proc.max();
199
200
```

```
246     // Load a simple data member
247     m_stream >> proc.address();
248     m_stream >> proc.interactionType();
249     m_stream >> proc.mobileDevice();
250     m_stream >> proc.min();
251     m_stream >> proc.max();
252
```

FIGURE 16 – Extrait du code : `Serialization` avec QDataStream

```
261 template <>
262 void JSONObjectReader::read(
263         const AppInteraction::ProcessModel& proc)
264 {
265     obj["SimpleElements"] = toJsonArray(proc.simpleElements);
266     obj["PolyElements"] = toJsonArray(proc.polymorphicEntities);
267     obj[strings.Address] = toJsonObject(proc.address());
268     obj["InteractionType"] = proc.interactionType();
269     obj["MobileDevice"] = proc.mobileDevice();
270     obj["Min"] = proc.min();
271     obj["Max"] = proc.max();
272 }
```

FIGURE 17 – Extrait du code : `Serialization` avec JSON

3.3.2 Les menus déroulants

L'ajout d'un menu déroulant dans l'`Inspector` du plugin revient à la création d'un `Widget`. Il faut, avant tout, ajouter une nouvelle classe widget. Par exemple, un `widget` menu déroulant permettant de sélectionner le dispositif mobile sur lequel sera envoyée la demande d'interaction `MobileDevicesWidget`. Pour ce faire, un setter doit être implémenté dans cette classe `widget`, un signal doit être créé et bien géré (voir section 3.3.7) et des fonctions doivent être implémentés dans l'`Inspector` et le `ProcessModel` pour assurer le fonctionnement suivant du `widget` : lorsqu' il reçoit un signal de la part de la `QComboBox` informant que l'utilisateur a cliqué sur un nouveau type d'interaction, le setter est utilisé pour que le menu déroulant s'arrête sur le type d'interaction demandé, puis un signal transportant l'indice du type d'interaction sélectionné est émis. C'est ce signal que l'`Inspector` va pouvoir détecter pour être informé du nouveau type d'interaction sélectionné.

Lorsque l'utilisateur sélectionne un nouveau dispositif mobile, un signal est émis. L'`Inspector` a au préalable connecté l'émission de ce signal par le menu déroulant à l'exécution d'une de ses propres méthodes : à la détection du signal, l'inspecteur va pouvoir mettre à jour l'indice du dispositif mobile sélectionné, le nouvel indice étant porté par le signal. Les signaux sont plus amplement

développés en partie 3.3.7.

Ne pas oublier d'assurer la sauvegarde des valeurs sélectionnées dans le menu déroulant et d'ajouter les nouvelles classes implémentées dans le `CMakeLists.txt`, comme mentionné dans la section 3.3.1

3.3.3 Inspector

Se trouvent dans la classe `AppInteractionProcessInspector` les champs appartenant à l'interface graphique : le champ pour l'adresse, le champ pour le type d'interaction ou encore le champ pour le min et le max. Widgets et commandes associées sont développés dans leurs parties respectives, retenons simplement que la classe de l'inspecteur possède comme attributs des widgets et instances de `QDoubleSpinBox` (classe fournie par Qt), qui correspondent aux menus déroulants et champs que l'utilisateur a à sa disposition pour paramétrer le `Process`.

Les champs min et max ont été créés afin d'éviter au maximum d'envoyer des valeurs incohérentes aux logiciels annexes : l'application peut alors envoyer une valeur dans l'intervalle $[0, 1]$, que l'`Executor` utilise comme un ratio pour envoyer une valeur appartenant à $[\min, \max]$.

Dans l'inspecteur sont instanciés tous les widgets et autres objets graphiques présents dans le panel inspecteur : instances de `AddressAccessorEditWidget`, `InteractionTypeWidget`, `MobileDevicesWidget` et `SpinBox` pour le min et max. Un pointeur est conservé comme attribut pour chacune de ces instances, puis les différents signaux qu'elles sont susceptibles d'émettre sont connectés aux méthodes de l'inspecteur correspondantes. Ces méthodes seront soit des setters des attributs de l'inspecteur, soit elles activeront des `Commands` : `ChangeAddress`, `ChangeInteractionType`, `ChangeMobileDevice`, `SetInteractionMin` ou `SetInteractionMax`. Les signaux sont plus amplement expliqués en partie 3.3.7.

3.3.4 Executor

Comme énoncé en partie 3.2.3, les méthodes principales de l'exécuteur sont `start()`, `pause()` et `state()`. Cette dernière méthode est appelée à chaque tic d'horloge, puis sa sortie est automatiquement traitée par *i-score* : cette méthode nous sert en l'occurrence à envoyer des messages vers les *devices*.

Une méthode `interactionValueReceived` a été ajoutée au squelette initial de l'exécuteur : lorsque une connexion envoie un signal indiquant qu'une donnée provenant de l'application a été reçue, cette méthode est appelée, car elle a été connectée à ce signal dans le constructeur de l'`Executor` (voir la partie 3.3.7 concernant les signaux). Elle va pouvoir calculer la valeur à envoyer en utilisant la valeur reçue de l'application comme un ratio à appliquer entre les valeurs *min* et *max*, si la valeur est bien dans $[0, 1]$. Ce traitement dépend bien sûr du type des valeurs reçues : l'application envoie des `ossia::value` qui peuvent contenir des entiers, flottants, `array<float,2>`, etc. Il y a donc une vérification

du type de données contenu dans l'`ossia::value`, puis utilisation des valeurs reçues comme ratios pour définir les valeurs finales. L'utilisation du ratio est aujourd'hui implémentée pour les flottants et les vecteurs de 2, 3 ou 4 éléments (`array<float,2>`, `array<float,3>`, `array<float,4>`), qui sont 4 des multiples types gérés par les `ossia::value`. Pour les types non gérés, les valeurs reçues sont directement transmises sans application de ratio. Notons que dans le cas de la réception d'un entier, il ne peut être considéré comme un coefficient et donc ne permet pas d'appliquer un ratio.

La méthode `interactionValueReceived` va ensuite construire le message qui devra être transmis au *device* à modifier : elle crée un message, contenant la valeur à transmettre ainsi que l'adresse à laquelle le message doit être envoyé. Enfin elle stocke ce message dans l'attribut `State::Message m_msg` du `ProcessExecutor`. La méthode qui se charge de transmettre le message au logiciel annexe est `state()`, qui est automatiquement appelée à chaque tic d'horloge. Elle va retourner le message final qui sera ensuite traité et envoyé par *i-score*.

3.3.5 Rendu visuel du `Process`

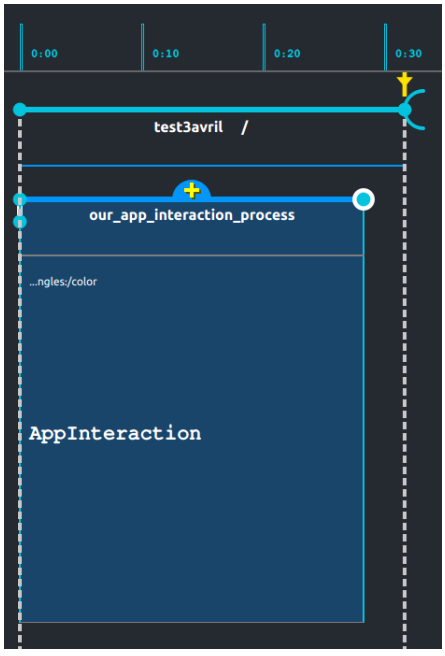

FIGURE 18 – Partition contenant un `Process AppInteraction`

Le visuel actuel d'une `AppInteraction` est assez sommaire, comme vous le voyez sur la figure 18. Il s'agit d'un simple affichage du type de `Process`, soit « AppInteraction ». L'affichage en question est initialisé dans le constructeur de la vue, c'est à dire la classe `AppInteractionView` dans le répertoire

`Process/Layer/`.

L'adresse du *device* sélectionné est également affichée directement sur la « boîte » de l'`AppInteraction`. Ceci est permis par la classe `AppInteractionPresenter` et la vue `AppInteractionView` du répertoire `Process/Layer/`. Un signal est intercepté lors d'un changement de l'adresse afin d'utiliser les méthodes `on_addressChanged()` et `setDisplayedName()` qui, par le biais de l'utilisation d'un `QTextLayout` et d'un objet `QTextLine`, mettent à jour l'affichage.

3.3.6 Le protocole de connexion

Il est à noter qu'une unique instance de `ConnectionManager` est nécessaire dès la création du Process pour gérer la connexion (*plugin - application mobile*). Pour cela, nous avons exploité le concept de `Context`, déjà expliqué, pour instancier `ConnectionManager` une seule fois comme un *Document-wide object* (accessible partout dans notre plugin).
En pratique, une unique instance de `DocumentPlugin` est automatiquement créée pour chaque instance du `Process`. Nous avons donc choisi de conserver notre unique instance de `ConnectionManager` comme attribut du `DocumentPlugin`. (Voir : `AppInteraction/DocumentPlugin/AppInteractionDocumentPlugin.cpp`)

Ensuite, une référence sur l'instance en question est récupérable grâce à un objet appelé *contexte* et est donc utilisable dès lors qu'on connaît ce contexte. Ainsi, on peut notamment récupérer un pointeur vers le `ConnectionManager` comme suit :

```
auto* m_connectionManager =
context.plugin<AppInteraction::DocumentPlugin>().connectionManager();
```

Ce mécanisme est utilisé en particulier dans l'`Executor` : celui-ci a besoin du `ConnectionManager` pour récupérer la liste des `Connection`, afin de transmettre les demandes d'interactions et récupérer les données envoyées par l'application mobile. Il est encore utilisé dans `MobileDevicesWidget` pour récupérer, d'une façon dynamique, la liste des `MobileDevices` (soit les appareils mobiles) connectés afin de les afficher dans le menu déroulant de l'`Inspector`, afin que le compositeur puisse choisir l'appareil auquel est envoyé la demande d'interaction. Dans l'état final de notre projet, cette dernière fonctionnalité a été vérifée avec un faussaire.

Le protocole utilisé doit être clair et bien défini des deux côtés (plugin et application mobile). La figure 19 présente le schéma d'un échange de données entre le plugin et l'application. Le format de message envoyé est le suivant :

```
std::string interaction = fmt::format("{:d}:{:f}",
                                      m_interaction-1,
                                      m_duration);
```

La demande d'interaction est donc une `std::string` comportant l'indice correspondant à l'IHM choisie par le compositeur, le séparateur « : », puis la durée

totale de l'interaction (du `Process`). Le « - 1 » correspond simplement au fait que l'indice 0 correspond au choix d'IHM « None » : aucune demande d'interaction n'est alors envoyée.

Remarquez l'utilisation de la fonction `fmt::format()` : elle permet de construire très facilement une `std::string`. Vous trouverez sa documentation sur le net.

Dans une première version de test, le plugin a réussi d'envoyer le message

```
"hello this is iscore"
```

en utilisant la méthode `openConnection` de `ConnectionManager` à l'ouverture de connexion via la ligne suivante :

```
pSocket->write("hello this is iscore");
```

La connexion est plus amplement détaillée en partie 5.

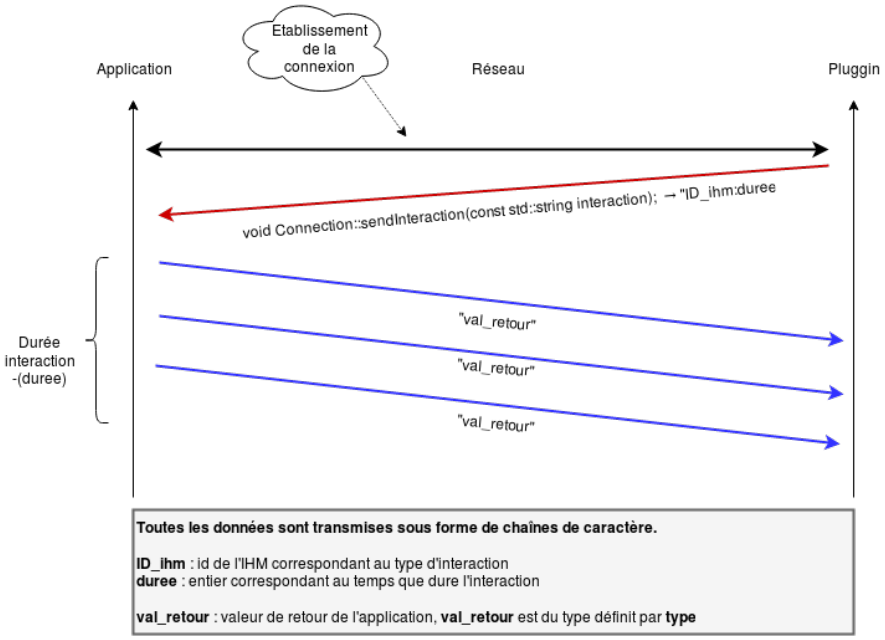

FIGURE 19 – Diagramme d'échange de données

3.3.7 Comprendre les signaux

Les signaux sont des outils fournis par Qt qui permettent la communication entre objets. Dans l'inspecteur par exemple, il s'agit de détecter les signaux émis par les widgets et le `ProcessModel`, afin d'exécuter les fonctions associées. Par exemple, il existe un widget menu déroulant permettant de sélectionner le dispositif mobile sur lequel sera envoyée la demande d'interaction. Lorsque

l'utilisateur sélectionne un nouveau dispositif mobile, un signal est émis. L'inspecteur a au préalable connecté l'émission de ce signal par le menu déroulant à l'exécution d'une de ses propres méthodes : à la détection du signal, l'inspecteur va pouvoir mettre à jour l'indice du dispositif mobile sélectionné, le nouvel indice étant porté par le signal.

Ce mécanisme est implémenté comme suit :

— Déclaration du prototype du signal :

```
signals:
    void mobileDeviceChanged(int);
```

— Envoi d'un signal :

```
emit mobileDeviceChanged(new_index);
```

— Connexion de la réception d'un signal à l'appel d'une méthode :

```
connect(
      m_mdw, //pointeur sur objet qui émet le signal
      &State::MobileDevicesWidget::mobileDeviceChanged, //signal
      this, //instance qui détecte le signal
      &InspectorWidget::on_mobileDeviceChange); //méthode
                                          //appelée si signal
```

La fonction `connect` associe le signal `mobileDeviceChanged` émis par le widget pointé par `m_mdw` à l'appel de la méthode `this.on_mobileDeviceChange`.

Notons que le signal et la méthode qu'elle déclenche doivent prendre les mêmes types de paramètres : les paramètres du signal seront automatiquement passés en paramètres de la méthode qu'il déclenche !

La méthode `con()` est également utilisée dans l'`Inspector`, afin d'intercepter des signaux provenant du `ProcessModel`. Cette méthode applique en réalité la méthode `connect()` mais effectue les conversions de types nécessaires sur le premier paramètre.

Il faut également savoir qu'un signal doit être envoyé par un `QObject`. Si la classe souhaitant intercepter les signaux n'est pas un `QObject`, la syntaxe diffère quelque peu. Un exemple est disponible dans le constructeur du `ProcessExecutor`. L'exécuteur doit connecter l'instance de `Connection` par laquelle passent les demandes d'interaction et les données à la méthode `interactionValueReceived`, afin de pouvoir récupérer les données transmises par l'application. Pour ce faire, et puisque l'exécuteur n'est pas un `QObject`, il utilise une surcharge de `connect()` qui lui permet de ne pas spécifier l'instance de `QObject` qui désire intercepter le signal (l'exécuteur lui-même) mais plutôt de ne passer en paramètre que la méthode à appeler en cas de signal reçu, sous la forme suivante :

```
QObject::connect(
    m_connections[m_mobileDevice-1], //pointeur sur une Connection
    &connection::Connection::interactionValueReturned, //signal
    [=] (const auto& val)
    {
        this->interactionValueReceived(val);
    });
```

Remarquons qu'il est parfois nécessaire de déconnecter des méthodes des signaux qui les déclenchent. Prenons pour exemple la classe `ProcessExecutor` : il est essentiel de déconnecter les signaux qu'il détecte des méthodes associées dans son destructeur. En l'absence de cette déconnexion, un *SegFault* apparaît à la relecture d'une partition : l'instance de `ProcessExecutor` est automatiquement détruite dès lors qu'on stoppe la lecture, mais ses méthodes continuent d'être appelées lors de l'émission des signaux de la seconde lecture ! Cette déconnexion se présente sous la forme suivante :

```
QObject::disconnect(
    m_connections[m_mobileDevice-1], //poiteur sur l'objet qui émet le signal
    &connection::Connection::interactionValueReturned, //signal
    NULL,
    (void**)0);
```

3.4 Remarques

Les QObject Il est à retenir qu'une instance d'une classe héritant de `QObject` doit être manipulée grâce à un pointeur : l'utilisation d'un vecteur de telles instances induiraient sinon une copie de ces instances, ce qui n'est pas possible.

Utilisation de QtCreator Pour compiler *i-score* et le plugin avec l'IDE QtCreator, il faut :

- ajouter dans « Environnement de Compilation » la variable BOOST_ROOT, qui doit donner le chemin vers le répertoire de *boost* (version récente précisée dans les instructions d'installation d'*i-score*)
- préciser dans « Compiler et Exécuter » le chemin vers le répertoire de *cmake* (version récente précisée dans les instructions d'installation d'*i-score*)
- préciser dans « Compiler et Exécuter » le chemin vers le répertoire de *g++* (version récente précisée dans les instructions d'installation d'*i-score*)
- préciser dans le paramétrage du kit de compilation quelles versions de *g++* et *cmake* doivent être utilisées
- ajouter « -j[nombre de coeurs de votre machine] » dans les options de compilation de cmake (cela permet d'accélérer la compilation en la parallélisant)

— cliquer sur « Ouvrir un projet existant », puis sélectionner le CMakeLists.txt racine de *i-score* pour avoir accès à tous les fichiers sources.

4 Application

4.1 Compilation

Pour compiler l'application en version Desktop il suffit d'ouvrir le projet sur **QT Creator** en sélectionnant le fichier **Simple_app.pro** et effectuer les étapes suivantes.

4.1.1 Compilation de l'API OSSIA :

L'application utilise la bibliothèque OSSIA pour réaliser la connexion avec *i-score*. Donc il faut effectuer une petite démarche pour pouvoir la linker au projet. Cela est fait en trois étapes.

- **Mise à jour de OSSIA :**
  Il faut se mettre dans le dépôt git de *i-score* et exécuter les 3 commandes suivantes :
  - git pull
  - git checkout master
  - git submodule update
- **Compilation de OSSIA :**
  Pour pouvoir compiler OSSIA il faut avoir une version de CMake >= 3.5 et une version de Boost >= 1.62.
  Après, il suffit d'exécuter les commandes suivantes dans un dossier build :
  - cmake `chemin/vers/dossier/API/dans/dépôt_iscore`
    -DCMAKE_INSTALL_PREFIX=api-inst
    -DBOOST_ROOT= `chemin/vers/boost`
    -DOSSIA_PD=0 -DOSSIA_PYTHON=0
  - make -j4 (4 correspond au nombre de coeurs dans le CPU)
  - make install
- **Inclusion de OSSIA :**
  Pour inclure OSSIA dans le projet, il suffit d'aller dans les options du projet et sélectionner : Add library -> External Library. Puis il faut remplir le champ library file par le chemin vers le fichier de l'API compilée qui se trouve dans `build/api-inst/include`

4.1.2 Compilation :

Le fichier **Simple_app.pro** contient toutes les commandes nécessaires à la compilation. Ainsi il suffit de cliquer sur le bouton **Run** pour avoir l'exécutable.

4.2 Architecture

Cette application est développée telle que l'interface est en QML et le modèle (le cœur) en C++. Les données sur les interactions sont définies dans le fichier

interactions.json mais on n'a implémenté que la première (Slider) dans ce projet.

4.2.1 Données sur les interactions : JSON

Pour pouvoir manipuler les interactions, on doit stocker quelque part les informations décrivant chacune d'entre elles. Ainsi on définit dans le fichier JSON la liste de toutes les interactions. Chacune est décrite par les attributs suivants :

- *eindex* : un entier identifiant l'interaction.
- *type* : définit l'élément du spectacle sur lequel l'interaction peut agir.
- *sensor* : définit les capteurs que l'interaction fait intervenir, cet attribut est noté **null** si l'interaction n'utilise aucun capteur.
- *name* : nom de l'interaction
- *icon* : le nom du fichier image représentant l'icône de l'interaction
- *file* : le nom du fichier QML contenant l'IHM de l'interaction
- *description* : elle sert à expliquer à l'utilisateur comment effectuer l'interaction.

```
{"interactions": [
    {
            "eindex" : 0,
            "type" : "volume",
            "sensor" : null,
            "name" : "VolumeUI",
            "icon" : "volume_cursor2.png",
            "file" : "CountDown0.qml",
            "description" : "Move the cursor continuously to the right to increase the volume, and to the left to decrease it"
    },
```

FIGURE 20 – Exemple de description d'une interaction

4.2.2 Interface graphique (vue) : Partie QML

Dans cette application la navigation entre les différentes pages est basée sur le *StackView* qui met en œuvre un modèle de pile. Les pages visualisées - sans retour en arrière - par l'utilisateur sont stockées dans la pile et dépilées à nouveau quand il choisit de revenir en arrière. Au lancement de l'application, la fenêtre principale permanente (c'est à dire elle n'est pas dépilée) est chargée. Après, lors d'une interaction, la fenêtre relative est empilée pendant un intervalle de temps bien défini puis dépilée.

Fenêtre permanente :
La fenêtre permanente est chargée au lancement de l'application. Elle est composée de la page de connexion (définie dans le fichier **Principal.qml**) suivie de la

page du menu des interactions (définie dans le fichier **InteractionMenu.qml**). Cette dernière contient la liste des différentes interactions possibles et leur description. Ces données sont récupérées à partir d'un fichier **JSON** contenant toutes les informations nécessaires.

FIGURE 21 – Menu des interactions

La communication entre le fichier Json et les composants QML est faite grâce à un parsing du fichier JSON, effectué grâce à l'objet Javascript `XMLHttpRequest`, qui va ouvrir le fichier JSON de manière asynchrone, donc sans se bloquer pendant l'ouverture. Il lance ensuite un parseur qui va tout d'abord attendre que l'ouverture du fichier soit terminée, puis exécuter le code Javascript correspondant aux actions à réaliser. Le parseur se contente généralement d'extraire des valeurs du fichier JSON et de les placer dans des variables globales, utilisées par la suite dans le code QML.

```
model: ListModel {
    id: listModel
    function completeHandler(string)
    {
        function myParserSearch()
        {
            if (searchReq.readyState == 4)
            {
                var doc = eval('(' + searchReq.responseText + ')');
                var i;
                var counter = 0;
                for (i=0;i<doc.interactions.length;i++)
                {
                    var complete = doc.interactions[i].type;
                    if(feasible[i] == 't' && complete.indexOf(string) != -1)
                    {
                        listModel.append({"eindex": counter, "type":complete, "icon":doc.interactions[i].icon,
                               "description":doc.interactions[i].description});
                        counter++;
                    }
                }
            }

        }
        listModel.clear();
        var searchReq = new XMLHttpRequest();
        searchReq.open("GET", "interactions.json", true);
        searchReq.onreadystatechange = myParserSearch;
        searchReq.send(null);
    }
    Component.onCompleted: {
        completeHandler("");
    }
```

FIGURE 22 – Interaction entre JSON et QML

Fenêtre de l'interaction

La deuxième fenêtre est chargée lors de la réception d'une interaction de la part de *i-score*. Chaque interaction se traduit par le chargement d'une première page de compteur à rebours contenant la description de l'interaction qui va se produire, suivie de l'IHM de l'interaction qui elle aussi est associée à un deuxième compteur.

Même si on n'a implémenté qu'une seule interaction, on a essayé de créer des éléments QML génériques pour faciliter l'ajout d'autres interactions et pour éviter la duplication du code.
En effet, le fichier **mainmv.qml** décrit la page d'attente avant l'affichage de l'IHM et récupère les données de l'interaction à partir du fichier JSON. Cela est fait grâce à la variable globale **ino** initialisée à -1 puis remplie par l'identifiant de l'interaction voulue.

Le code ci-dessous montre le parsing du fichier JSON pour remplir les variables : **iname** (nom de l'interaction), **idesc** (desciption de l'interaction), **iimage** (nom du fichier image représentant l'icône), **ifile** (nom du fichier IHM qui va être empilé)

```
property int ino: -1
property int  countdown: 10
property string iname: ""
property string idesc: ""
property string iimg: ""
property string ifile: ""
signal changeSlide(real r)

Component.onCompleted: {
function myParserInit()
{
   if (initReq.readyState == 4)
   {
       var doc = eval('(' + initReq.responseText + ')');
       //var ino = 0;  //numero de l'interaction reçue par i-score
       iname = doc.interactions[ino].name;
       idesc = doc.interactions[ino].description;
       iimg = doc.interactions[ino].icon;
       ifile = doc.interactions[ino].file;

   }
}
var initReq = new XMLHttpRequest();
initReq.open("GET", "interactions.json", true);
initReq.onreadystatechange = myParserInit;
initReq.send(null);
}
```

FIGURE 23 – Recupération des données

La dynamique de transition entre les pages d'interaction est faite grâce à des compteurs. Le premier compteur est associé à la page d'attente ayant un attribut *countdown* initialisé à 10 (secondes). Á chaque intervalle d'une seconde on le décrémente, si on arrive à zéro on charge l'IHM dans le *stackView*. La figure ci-dessous représente le premier compteur :

```
Timer {
    id: countdownTimer
    interval: 1000
    running: window.countdown > 0
    repeat: true
    onTriggered:
    { window.countdown--
      if (window.countdown == 0)
              stackView.push("qrc:/" + ifile)


   }
}
```

FIGURE 24 – Premier compteur à rebours

La figure ci-dessous représente le deuxième compteur de la page de l'IHM qui elle aussi a un attribut *counter* qui est décrémenté de la même façon jusqu'à fermeture de la fenêtre lorsqu'il s'annule.

```
Timer {
    id: counter
    interval: 1000
    running: page.interactionTime > 0
    repeat: true
    onTriggered:{
        page.interactionTime--
        if (page.interactionTime == 0)
            window.close()
       /* if (page.interactionTime == 3)
            connectionError.open()
        if (page.interactionTime == 1)
            connectionError.close()
          */
    }

}
```

FIGURE 25 – Deuxième compteur à rebours

Les pages des IHM suivent la convient de nommage suivante : **CountDownID.qml**. Ainsi, la seule interaction implémentée correspondant au slider a été réalisée dans le fichier **CountDown0.qml**

4.2.3 Contrôleur : Partie C++

Fichier extract.cpp

Dans ce fichier on implémente la fonction du parsing de la chaîne de caractères

qu'on reçoit de la part de *i-score* afin de récupérer la durée de l'interaction et l'identifiant de son IHM et la mettre dans la structure **datai**.

```
struct datai{
    int id;
    int duration;
};

struct datai* extract_data(const char* str);
```

FIGURE 26 – extract.hpp

Fichier signal.cpp

La communication entre les objets QML et C++ est le coeur de notre application, puisque notre but est de récupérer l'action de l'utilisateur sur l'IHM (composants QML) et la transmettre vers *i-score* (en C++). La communication entre ces objets se fait à l'aide des signaux. Ainsi, pour chaque action sur un objet QML graphique, on crée un signal et on implémente dans **signal.cpp** la fonction `signalHandler` décrivant l'action C++ associée.
La figure 27 représente un exemple de connexion entre l'action sur le *Slider* de l'interaction 0 et la fonction *handleSig*.

```
Signal s ;
QObject::connect(wobject, SIGNAL(changeSlide(double)),
                 &s, SLOT(handleSig(double)));
```

FIGURE 27 – Exemple de connexion entre un objet QML et un objet C++

4.3 Comment ajouter une interaction ?

Pour ajouter une interaction il faut faire les étapes suivantes :

— Ajouter l'interaction dans le fichier JSON en lui attribuant un identifiant et en remplissant les différents attributs.

— Créer le fichier QML de l'IHM en suivant la convention de nommage : **countDownID.qml** avec ID l'identifiant de l'interaction.

— Créer un signal QML et l'associer aux éléments de l'IHM ajoutée.

— Implémenter la fonction gestionnaire de signal dans **signal.cpp** et effectuer la connexion entre les deux.

5 Partie Connexion : OSSIA API

L'établissement de la connexion se fait dans un premier temps à l'aide des classes `QTcpServer` et `QTcpSocket` fournies par Qt. Pour cela, le serveur est instancié dans le constructeur du `ConnectionManager` ce qui permet de l'initialiser dès la création d'un nouveau document. On ouvre le serveur sur un port choisi et on lui fait écouter le réseau.

```
if (m_serv->listen(QHostAddress::Any, m_localTcpPort))
{
    connect(m_serv, &QTcpServer::newConnection,
    this, &ConnectionManager::openConnection);
}
```

Ainsi, lorsque le serveur reçoit une nouvelle connexion, la fonction `openConnection` est appelée. On peut alors récupérer la socket correspondante via

```
QTcpSocket *pSocket = m_serv->nextPendingConnection();
```

Puis on peut procéder à l'utilisation des fonctions OSSIA pour la communication.
Pour l'application, il n'y a qu'à exécuter le code suivant.

```
connect(&m_socket, &QTcpSocket::connected,
this, &ClientConnection::onConnected);
connect(&m_socket, &QTcpSocket::disconnected,
this, &ClientConnection::closed);
m_socket.connectToHost(servAddr, 9999);
```

Par la suite, le serveur instancie un serveur `ossia::oscquery::osquery_server_protocol` et envoie via la socket un message comportant l'adresse IP et le numéro de port du serveur. L'application peut alors instancier un `ossia::oscquery::oscquery_mirror_protocol` qui sera le client. L'application doit ensuite construire son *mobileDevice* et l'arbre qui le représentera dans *i-score*. On peut trouver un exemple sur le dépôt GitHub d' `OSSIA`.
Une fois l'arbre créé, l'application peut retrouver le noeud voulu avec

```
node_base *find_node(node_base& , ossia::string_view)
```

et modifier la valeur attachée grâce à

```
generic_address& generic_address::pushValue(const ossia::value&)
```

La nouvelle valeur est automatiquement envoyée sur le réseau et pour la récupérer depuis le serveur, on peut ajouter un `callback` à l'adresse du noeud, qui va notifier le serveur d'un changement de la valeur du noeud, et lui permettre d'exécuter du code en fonction. Le prototype est :

```
iterator ossia::callback_container<T>::add_callback(T callback)
```

Il faut passer en argument une lambda-expression qui contient le code que l'on souhaite exécuter lors du changement de valeur.