ENSEIRB-MATMECA

PFA - APPLICATION TACTILE POUR LES ARTS INTERACTIFS

Rapport de projet

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

1 Introduction

Du mois d'octobre au mois d'avril nous avons mené ce « projet au fil de l'année » dans le cadre de nos études à l'ENSEIRB-MATMECA. Le projet consiste en la création d'une application mobile pour Android et d'un plugin pour le logiciel *i-score*. Développé par Blue Yeti / LaBRI, ce logiciel exécute des partitions musicales et visuelles créées par un compositeur.

L'objectif de l'application est de pouvoir interagir avec un type de partition dédié afin que l'utilisateur du dispositif mobile puisse prendre part au spectacle de manière sonore ou visuelle, sans forcément avoir accès au logiciel à ce moment-là. C'est ce que nous appelerons interaction. Cette intervention est donc contrôlée par la partition que le compositeur a mise en place au préalable.

Par exemple, une interaction viable pourrait être une secousse du téléphone qui engendre une modification du volume sonore, ou encore une modification de l'intensité des projecteurs.

Ce rapport présente le travail que nous avons effectué sur ce projet, aussi bien en ce qui concerne notre organisation, l'application, le plugin ou la connexion.

2 Logiciel existant : *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, en partie 7, 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*.

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 l'utilisateur que nous appellerons « 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 Organisation

Le projet étant réalisé par une équipe de 7 personnes, nous avons constitué des équipes en répartissant le travail sur les 3 axes principaux de ce projet : l'application, le plugin et la connexion entre ces deux entités.

Tous les 15 jours, nous rencontrions notre responsable pédagogique, à qui nous exposions l'avancée du projet : ce qui a été fait, les problèmes rencontrés. Le but de ces réunions était de choisir les tâches à effectuer lors du prochain sprint et affecter correctement les personnes en fonction de la difficulté des tâches.

Un compte rendu était maintenu à chaque réunion, afin de pouvoir faire le point sur l'avancée des tâches en cours de sprint en le comparant notamment au diagramme de Gantt réalisé lors de la rédaction du document de spécifications.

Des réunions avec le client étaient également fréquemment prévues pour avoir des précisions sur le logiciel existant mais aussi discuter des demandes. De la même manière, un compte rendu était rédigé lors de chaque réunion si celui-ci était nécessaire.

Un dépôt GitHub a été mis en place pour gérer les différents travaux des équipes.

En ce qui concerne la communication, elle se faisait par mail ou par Gitter avec le client et le responsable pédagogique. Un service de messagerie instantanée était utilisé entre nous.

4 Application

Notre projet nécessite qu'une personne sur scène puisse interagir avec le séquenceur *i-score*, et modifier les paramètres du spectacle pendant son déroulement. Pour ce faire, l'utilisation d'une version du séquenceur *i-score* sur un ordinateur portable est inadaptée car cela empêche le compositeur de participer au jeu de scène. Il est donc nécessaire de pouvoir modifier les données du spectacle depuis un dispositif connecté en temps réel avec le séquenceur. Afin que cette modification soit accessible au plus grand nombre, et même à des personnes faisant parte du public, il est nécessaire de développer une application mobile multiplateforme.

4.1 Objectifs

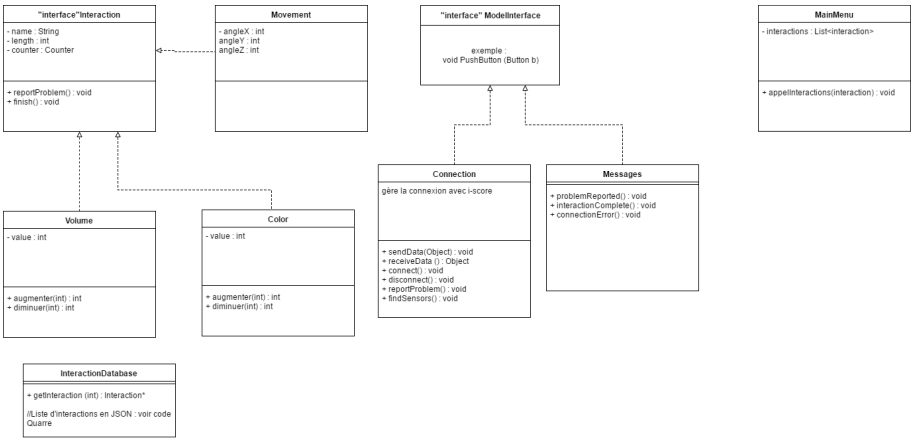

FIGURE 1 – Architecture de l'application

L'application est à créer entièrement. Elle devra pouvoir fonctionner sur Android et iOS, la version fonctionnelle sur Android étant prioritaire. Elle permettra d'effectuer les interactions programmées par le compositeur depuis le séquenceur *i-score*, et doit pour cela pouvoir se connecter au séquenceur. Le but de l'application n'est pas de choisir les effets induits par l'interaction sur le spectacle, ce choix étant laissé uniquement au compositeur lors de la création de la partition.

Le rôle de l'application peut ainsi être décomposé en plusieurs étapes clés. Tout d'abord, il faut récupérer les données que le séquenceur *i-score* nous a envoyées, puis il faut les décoder afin d'en extraire les principaux paramètres des interactions (durée, type...). Chaque interaction est associée à une interface homme-machine contenant un ensemble d'objets graphiques permettant de modifier des données, d'afficher l'état d'un paramètre du spectacle ou de guider l'utilisateur sur la marche à suivre et sur les mouvements à effectuer sur son dispositif. Lors d'une demande d'interaction, il faut ensuite afficher l'interface homme-machine correspondant à l'interaction, de manière à ce que l'interaction puisse être effectuée facilement par l'utilisateur notamment en rapport avec les capteurs disponibles sur le dispositif. Les résultats des actions sur les objets graphiques, ainsi que les mouvements sur le dispositif, sont ensuite transmis au séquenceur *i-score*.

En revanche, certaines fonctionnalités devront être présentes au cours de n'importe quelle interaction :

— Pouvoir arrêter l'interaction avant que celle-ci soit censée se terminer, peu importe si des modifications ont été effectuées ou non ;
— Se déconnecter, sans empêcher les éventuels autres dispositifs de continuer à interagir avec le séquenceur ;
— Interrompre l'exécution de l'application en la passant en arrière-plan, sans que cela conduise à une perte de connexion. L'utilisateur doit donc pouvoir reprendre l'exécution sans avoir à se reconnecter.

Avant toute opération, dès son démarrage, l'application doit proposer à l'utilisateur un ou plusieurs logiciels *i-score* auxquels il peut se connecter. Lorsque le choix est réalisé, l'application envoie une demande de connexion au bon séquenceur, qui ajoute le dispositif dans son arborescence. L'application doit également tester tous ses capteurs de mouvement (accéléromètres, gyromètres ...) et envoyer la liste des capteurs fonctionnels à *i-score*. Elle se place alors en attente des interactions. Avant chaque interaction, un compte à rebours doit être lancé pour que l'utilisateur puisse s'y préparer. Le début de ce compte à rebours doit être signalé par un message sur l'écran du dispositif et une vibration de celui-ci. Ensuite, pour que l'utilisateur comprenne ce qui est attendu de lui, la description de l'interaction s'affiche. Pendant l'interaction, l'application communique en permanence avec le séquenceur afin de lui transmettre les données provenant des capteurs du dispositif et des objets graphiques de l'IHM, pour que le séquenceur connaisse les choix effectués par l'utilisateur.

4.2 Structure du code

L'application que nous avons obtenue a été créée entièrement à partir de zéro. Pour son développement, nous avons choisi d'utiliser un pattern MVC. Le modèle, dont le but est de stocker les informations sur les interactions, est constitué d'un fichier au format JSON. La vue, qui affiche l'interface homme-machine et interagit avec l'utilisateur, a été codée en utilisant le langage QML. Le contrôleur, quant à lui, est codé en C++. Cependant, contrairement à ce que l'on trouve dans un modèle MVC classique, nous avons choisi de faire communiquer directement la vue avec le modèle, pour limiter les échanges d'information entre les différentes parties et ainsi gagner en fluidité.

4.2.1 Le modèle

Le fichier JSON utilise une structure proche de celle d'un tableau. En effet, chaque interaction y est référencée par un index unique débutant à 0. Il contient également les informations suivantes sur les interactions :

— Le type de l'interaction, c'est-à-dire le paramètre qui sera modifié (volume, couleur de lumière ...) ;
— Le ou les capteurs (accéléromètres, gyromètres ...) nécessaires pour sa réalisation. Si l'interaction ne demande aucun mouvement, mais seulement une action sur les objets graphiques de l'écran, alors ce champ vaut `null`. Par convention, chaque capteur est désigné par *Acc* pour accéléromètre et *Gyr* pour gyromètre, suivi éventuellement des directions concernées ;
— Le nom de l'interaction, qui est une chaine de caractère unique. Par convention, il s'agit du type de l'interaction suivi de *Move* si l'interaction demande un mouvement, ou *UI* sinon, mais le client est libre de choisir le nom qu'il veut ;

— L'icône représentant l'interaction. Il s'agit d'une image au format PNG, de taille fixée impérativement à 64x64 ;

— Le fichier contenant la vue spécifique à l'objet graphique de l'interaction. Par convention, ce fichier porte le nom de `CountDown`n`.qml`, avec n le numéro de l'interaction, mais là encore le client est libre d'ajouter un autre fichier ;

— Une description de l'interaction indiquant la marche à suivre pour effectuer cette interaction.

Le format JSON a été choisi pour sa simplicité de compréhension : en effet, le client peut être amené à le modifier s'il veut inventer de nouveaux types d'interaction ou en supprimer certains. De plus, il est facilement analysable par la vue en 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 2 – Interaction entre JSON et QML

4.2.2 La vue

Dans cette application, la navigation entre les différentes pages de la vue est basée sur le *StackView* qui met en œuvre un modèle de pile. Les pages visualisées, hors 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 qui n'est jamais empilée, est chargée. Après, lors d'une interaction, la fenêtre relative correspondant à l'interaction 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. Comme décrit en section 4.2.1, ces données sont récupérées à partir d'un fichier **JSON** contenant toutes les informations nécessaires.

FIGURE 3 – Menu des interactions

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 **mainmw.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** (description 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 4 – 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). À 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 5 – 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* décrémenté de la même façon jusqu'à fermeture de la fenêtre lorsqu'il atteint zéro.

```
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 6 – Deuxième compteur à rebours

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

4.2.3 Le contrôleur

En plus du fichier `ClientConnexion.cpp` utilisé pour la connexion à *i-score*, deux fichiers appelés par un fichier principal `main.cpp` ont été implémentés en C++.

Fichier extract.cpp

Dans ce fichier on implémente le 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**. Cette structure contient l'identifiant d'une interaction et la durée accordée à l'utilisateur pour l'effectuer.

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

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

FIGURE 7 – 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,le fichier `signal.cpp` contient les gestionnaires des signaux définis en section 4.2.5, ainsi que le gestionnaire du signal permettant de prévenir la vue qu'une interaction est attendue, et qu'il est temps d'afficher le décompte à l'utilisateur.

Fichier main.cpp

En dehors de son rôle de fichier principal du code source du contrôleur, le `main.cpp` appelle également le fichier principal de la vue, `Principal.qml`, qui correspond à l'écran d'accueil de l'application. Il se charge également de créer les signaux qui vont permettre l'échange avec la vue, et dont les gestionnaires sont définis dans le fichier `signal.cpp`.

4.2.4 Communication vue-modèle

La communication unidirectionnelle, du modèle vers la vue, se fait par un parsing du fichier JSON à partir de plusieurs endroits dans le code de la vue :

- Dans les fichiers paramétrant les objets graphiques des interactions `CountDown`n`.qml`, afin de récupérer le nom de l'interaction concernée et de l'afficher sur l'écran d'interaction ;
- Dans le fichier paramétrant l'écran d'attente avec le décompte `mainmw.qml`, afin de récupérer et d'afficher le nom, la description et l'icône de l'interaction qui est sur le point de démarrer, et d'appeler le fichier contenant les objets graphiques correspondants ;
- Dans le fichier paramétrant l'écran d'accueil avec la liste des interactions `InteractionsMenu.qml`, afin d'afficher une description de chaque interaction lorsque l'on clique sur son image.

Le parsing se fait 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 définies ci-dessus. 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. Notons que dans le fichier `InteractionsMenu.qml`, le premier parsing a aussi pour rôle dans notre code d'initialiser la variable `feasible`, qui est une chaîne de caractères de longueur égale au nombre d'interactions dans le JSON, et dont chaque caractère indique si l'interaction correspondante est réalisable sur le dispositif en fonction de ses capteurs fonctionnels. Par exemple, si `feasible` vaut "tftt", cela signifie que toutes les interactions du fichier JSON à l'exception de la deuxième sont réalisables.

4.2.5 Communication vue-contrôleur

Deux signaux ont été définis dans le but de permettre une communication efficace entre la vue en QML et le contrôleur en C++ : le premier se déclenche lors de la modification de la jauge de volume pour la seule interaction que nous avons implémenté, afin de récupérer en temps réel dans le contrôleur la position de la jauge, le second quant à lui ne se déclenche que lors de l'appui sur le bouton `Connect` de l'écran de connexion (premier écran de l'application), dans le but de prévenir le séquenceur qu'un nouveau dispositif vient de se connecter. En effet, auparavant nous avions utilisé les outils `QQmlEngine` et `QQmlComponent` dans le contrôleur afin de lire la valeur de variables globales de la vue, ce qui permettait de recevoir les données de la vue, et de pouvoir modifier cette valeur pour envoyer des données. Cependant, cette solution s'est avérée beaucoup plus difficile à mettre en

oeuvre et ne permettait pas une communication efficace en temps réel. C'est pourquoi nous avons choisi d'utiliser plutôt les signaux.

Chaque signal est défini par :

— Un type, qui représente le type des données transitant entre la vue et le contrôleur. Dans le cas de l'interaction permettant de modifier le volume en utilisant un curseur, il s'agit du type décimal `double`. En effet, la valeur 0 correspond à un curseur en butée gauche, et 1 à un curseur en butée droite : toutes les valeurs intermédiaires sont donc codées par un nombre entre 0 et 1. Pour l'appui sur le bouton `Connect`, il s'agit simplement d'un booléen valant `true` lorsque le bouton est pressé, et `false` sinon.

— Une variable de référence, dont la modification déclenche l'envoi du signal. Il s'agit de `changeSlide` pour l'interaction et `connectPushed` pour le bouton de connexion.

— Un gestionnaire (*signal handler*), qui est une fonction écrite en C++ et appelée par le contrôleur lors de l'envoi du signal. Ces fonctions sont définies dans le fichier `Signal.cpp` et son en-tête correspondant `Signal.hpp`. Dans le cas de l'interaction, le gestionnaire se nomme `handleSig` et se contente d'afficher la nouvelle valeur de la variable de référence. Dans le cas de l'appui sur le bouton en revanche, non seulement on affiche le nouvel état du bouton, mais l'on instancie une connexion avec le séquenceur *i-score* en utilisant son adresse IP, en utilisant la classe `ClientConnection`. Ce n'est alors qu'à partir de la réception du signal, et donc de l'appui sur le bouton `Connect`, qu'il est possible de se connecter au séquenceur.

La figure 8 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 8 – Exemple de connexion entre un objet QML et un objet C++

4.3 Fonctionnalités implémentées et comparaison par rapport aux cas d'utilisation

4.3.1 Cas 1 : Effectuer l'interaction spécifiée par la partition

Comme prévu, du côté de l'application, il est possible de déplacer un curseur seul sur une interface homme-machine. À chaque déplacement du curseur, la nouvelle position est stockée dans une variable qui est transmise au contrôleur de l'application, pour être ensuite envoyée au séquenceur. Cependant, un seul type d'interaction a été implémenté pour le moment : il s'agit du curseur que l'on déplace horizontalement pour faire varier le volume. Nous n'avons pas réaliser de tests en conditions réelles. Il n'est donc actuellement pas possible d'agir physiquement sur le volume du dispositif lors d'un spectacle réel, ce type d'interaction n'ayant pas été implémenté.

Bien qu'une seule interaction aie été implémenté jusqu'ici, la mise en place des interactions suivantes ne nécessiterait que peu de manipulations et les principes énoncés ci-après resteront les mêmes pour les autres types d'interactions. Sur l'écran de l'interaction, l'application propose un

curseur ainsi qu'un titre d'interaction permettant à l'utilisateur de se repérer parmi toutes les interactions dont il aura eu la description sur l'écran d'accueil. Un bouton `Finish Interaction` est également présent afin de permettre à l'utilisateur de passer l'interaction. Il n'y a pas en revanche de texte et de logos d'explication, ceux-ci étant présents sur l'écran d'attente qui s'affiche pendant le décompte précédent l'interaction. Ceci permet ainsi d'éviter que l'utilisateur soit perturbé par une quantité d'informations trop importante sur l'écran des interactions, alors que cet écran demande une action rapide.

Lorsque la fin du temps imparti pour l'interaction est atteint, ou que l'utilisateur a passé l'interaction en appuyant sur le bouton prévu à cet effet, l'interface homme-machine de l'interaction disparaît et l'utilisateur se retrouve à nouveau sur l'écran d'attente contenant la liste des interactions possibles sur l'appareil. En effet, l'écran d'attente est toujours placé en arrière plan, permettant ainsi un retour rapide sur celui-ci.

En raison des difficultés liées à l'établissement d'une connexion entre l'application et le séquenceur, les scénarios d'exception prévus n'ont pas été testés, la mise en place de tests sur l'état de la connexion en temps réel étant impossible. Cependant, un écran de perte de connexion, sous forme d'une fenêtre pop-up, a été implémenté dans la vue pour signaler à l'utilisateur que la connexion n'est plus disponible, et que l'application attend le retour de la connexion pour continuer à fonctionner. En revanche, par manque de temps, il n'a pas été inclus dans le code de l'application. Comme prévu par le document de spécifications, si la connexion est rétablie à temps avant la fin de l'interaction, alors l'utilisateur peut reprendre la modification des objets graphiques. Sinon, l'interaction est tout simplement annulée.

4.3.2 Cas 2 : Être informé de la prochaine interaction

Par défaut, le délai $\Delta t23$, défini en page 15 du document de spécifications, a été fixé à trois secondes. Comme prévu, le nom de l'interaction à effectuer s'affiche trois secondes avant son début. Une explication de l'action a réaliser sur l'interface correspondant à l'interaction en cours est également disponible sur le même écran, ainsi qu'un compte à rebours permettant de savoir le temps de préparation restant. Lorsque l'utilisateur prend la main sur l'IHM et peut agir sur les objets graphiques, l'écran d'attente avec l'icône correspondant à l'interaction disparaît.

En conformité avec le document de spécifications, toute perte de connexion à ce stade-là de l'exécution est transparente à l'utilisateur et ne provoquera pas l'arrêt ou la mise en pause de l'application, à condition que la connexion soit revenue avant la fin du délai $\Delta t23$. Cependant, l'utilisateur ne sera pas prévenu qu'il vient de perdre la connexion, bien que la fenêtre pop-up permettant de le prévenir ait été implémentée dans la vue comme pour le cas précédent.

4.3.3 Cas 3 : Établir la connexion de l'application avec un logiciel *i-score*

Comme cela avait été spécifié dans le cahier des charges, lors de son lancement, l'application demande à l'utilisateur de se connecter au séquenceur en appuyant sur un bouton. Cependant, *i-score* ne place finalement pas le nom du dispositif dans son arborescence, mais un pseudo qui aura été choisi par l'utilisateur, car en plus d'être plus difficilement récupérable au niveau de l'application, le nom du dispositif ne permet pas d'identifier rapidement un utilisateur sur le séquenceur. Sur l'écran

de démarrage de l'application qui contient le bouton de connexion, on a donc aussi un champ qui permet d'entrer son pseudo. Une demande de connexion contenant ce pseudo est ensuite envoyé à *i-score* via son adresse IP, et celui-ci peut y répondre favorablement ou non. Contrairement à ce qui avait été défini dans le document de spécifications, si aucun séquenceur n'est trouvé à l'adresse IP indiquée, l'application peut quand même se lancer normalement. En revanche, elle ne recevra évidemment aucune demande d'interaction, sauf dans le livrable que nous avons rendu où une demande d'interaction est simulée par un faussaire, dans un but démonstratif. La même situation se produit si *i-score* rejette la demande de connexion qui lui est envoyée.

4.3.4 Cas 4 : Transmettre les caractéristiques du smartphone ou de la tablette

Il était prévu que lors de son premier lancement après installation sur l'appareil, l'application récupère la liste des capteurs présents sur le dispositif, notamment les accéléromètres et gyromètres correspondant aux trois directions possibles. Elle devait ensuite demander à l'utilisateur d'effectuer des mouvements prédéfinis afin de les tester un par un, puis envoyer la liste des capteurs fonctionnels au séquenceur *i-score* afin que celui-ci les intègre dans son arborescence. Ce test aurait pu être réalisé également à n'importe quel moment, sur demande de l'utilisateur. En outre, en cas de perte de la connexion, il devait être possible d'attendre son rétablissement puis d'envoyer à cet instant-là la liste des capteurs. Ce cas d'utilisation n'a tout simplement pas été mis en oeuvre car la seule interaction implémentée ne nécessite pas de mouvement du dispositif : les capteurs sont donc inutiles dans la version de l'application délivrée.

4.3.5 Cas 5 : Transmettre en temps réel les données relatives à l'interaction en cours

Lors de l'utilisation d'un objet graphique présent sur l'écran d'interaction, la traduction entre le mouvement effectué par l'utilisateur et les valeurs numériques obtenues en langage QML se fait bel et bien comme prévu et de manière transparente pour le développeur car elle est gérée par des bibliothèques QML fournies avec Qt. Il n'y a toutefois pas de communication directe entre la vue développée en QML et le séquenceur *i-score* : les valeurs numériques transitent par le contrôleur de l'application en C++. Le passage du QML vers le C++ se fait par un système de signaux créés dans le contrôleur et émis au sein de la vue, comme défini en section 4.2.5. Les signaux permettent ainsi de gérer une communication en temps réel, sans devoir utiliser une boucle qui bloquerait le programme, ou créer de nouveaux threads qui complexifieraient l'écriture et la gestion du programme.

Le délai disponible pour l'exécution de l'interaction, défini dans le document de spécifications comme $\Delta t34$, a été ici fixé à quinze secondes mais peut être modifié par le séquenceur lors de l'envoi de sa demande d'interaction. Lorsque ce délai est atteint et que l'utilisateur n'a pas agi sur son application, ou alors lorsqu'il a appuyé sur le bouton permettant de passer l'interaction, comme prévu par les scénarios alternatifs du document de spécifications, l'application cesse de récupérer les données provenant des objets graphiques : en effet, leur modification est rendue impossible par leur disparition de l'écran. Ceci fonctionne également grâce aux signaux : de fait, lorsqu'une valeur correspondant à la variable de référence d'un signal n'est plus modifiée, le signal n'est plus envoyé. L'écran d'interaction se ferme et l'utilisateur revient sur l'écran d'attente.

4.3.6 Cas 6 : Transmettre les demandes d'interaction

En ce qui concerne ce cas d'utilisation, une différence avec le cahier des charges est que l'application ne reçoit pas de notification lorsque le séquenceur lui envoie une demande d'interaction. Pour le reste, elle reçoit juste un numéro qui correspond à l'identifiant de l'interaction à exécuter. L'application ne mémorise pas directement les informations de l'interaction, mais celles-ci sont stockées d'une part dans un fichier de vue dédié en QML, et d'autre part dans un fichier `interactions.json`. Le fichier de vue contient les informations relatives aux objets graphiques à présenter à l'utilisateur, et le fichier JSON contient le type de l'interaction, les capteurs à utiliser dans le cas où un mouvement de l'utilisateur est nécessaire, le nom de l'interaction, le fichier image correspondant à son icône et sa description. En outre, nous n'avons finalement pas prévu de temporisation entre la réception de l'interaction et le début du compte à rebours, car nous avons considéré que le séquenceur envoyait une demande d'interaction au moment où il était temps de lancer le compte à rebours.

5 Plugin

Le projet consistant en la création d'une application mobile lié à l'activité du séquenceur *i-score*, il fallait que le logiciel soit capable d'échanger des données avec l'application. Pour réaliser ce travail, nous avons implémenté un plugin pour *i-score*.

5.1 Objectifs

Outre le fait de devoir être fonctionnel, le plugin deoit réaliser plusieurs tâches :

— Fournir à *i-score* la possibilité de sélectionner une interaction qui sera spécifique aux échanges avec notre application.

— Assurer la connexion avec une application mobile, en d'autres mots, il s'agit là de veiller à la bonne direction des informations. En effet, en reprenant le document de spécification, on peut lire que *i-score* doit pouvoir prendre en charge plusieurs appareils mobiles. Il faut donc s'assurer de la bonne répartition des données.

— Assurer le relais des données avec les logiciels externes que l'on notera *devices* par la suite. Les informations reçues doivent être traitées puis envoyées vers les devices prévus et ce travail doit être implémenté dans le plugin.

5.2 Réalisation

5.2.1 Intégration du plugin

Le séquenceur intermedia est assez permissif au niveau de l'ajout de nouveaux types de `Process` puisque ces types correpondent à des plugins indépendants simplement ajoutés dans l'architecture : il suffit de positionner les sources du plugin dans le dossier `base/addons/` pour que le `CMakeLists.txt` associé à ce dossier repère le plugin et l'intègre au logiciel *i-score*.

Le client nous a fourni un modèle général de plugin contenant les fichiers essentiels à implémenter pour ajouter un type d'interaction. Notre travail sur cet addon a donc consisté en le couplage de nos besoins avec les contraintes d'implémentation pour la compatibilité avec *i-score*.

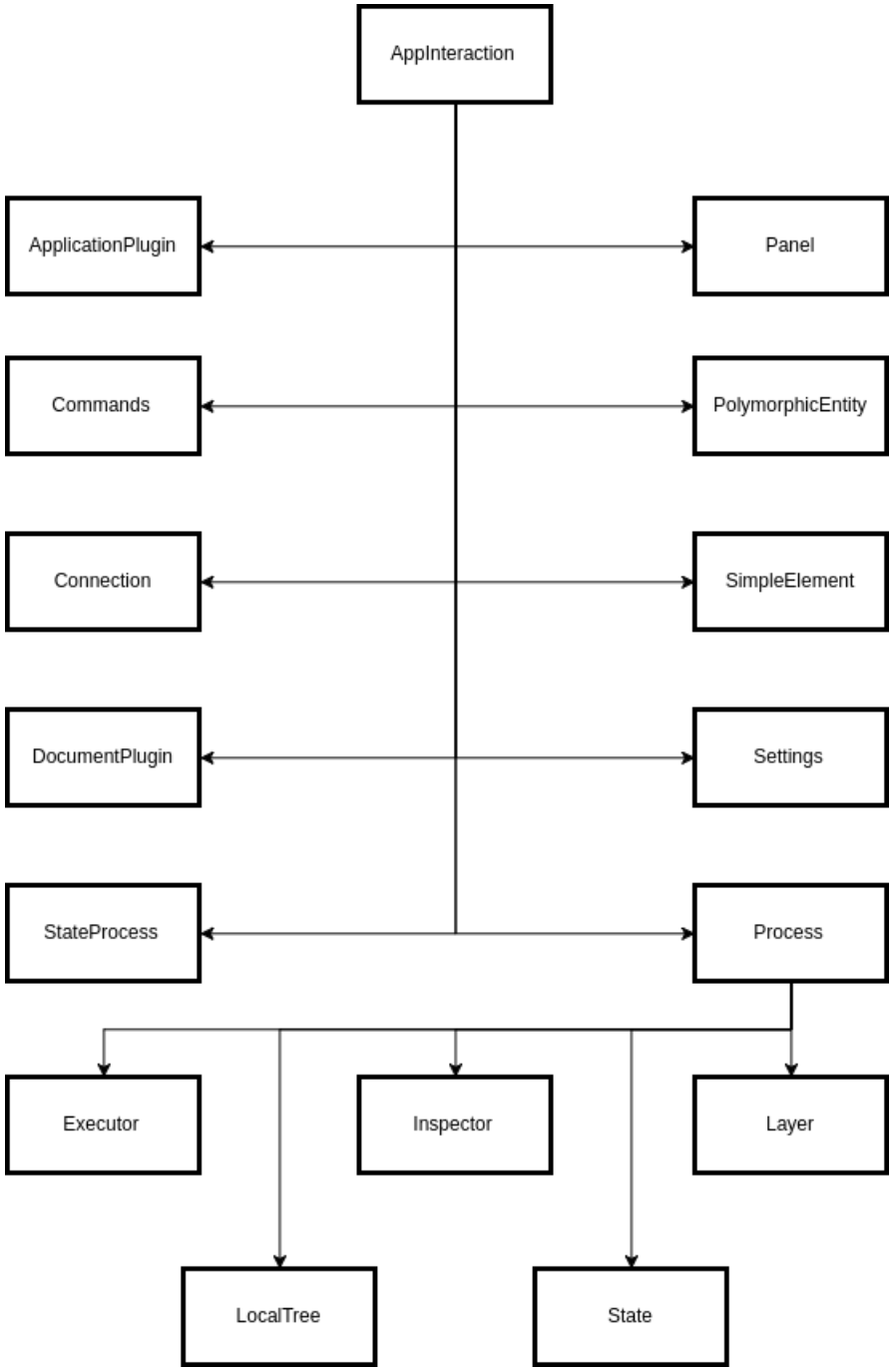

FIGURE 9 – Architecture du dossier de plugin

La figure 9 présente la répartition des dossiers de notre plugin. Les principaux dossiers qui nous ont été utiles étant les suivants :

— `Process` (classes `ProcessModel`, `Inspector`, `Executor`, `Layer` notamment)

— `Commands`

— `State` (implémentation des *Widgets*)

— `Connection`

— `DocumentPlugin`

Nous détaillons dans la suite les implémentations et concepts fondamentaux que nous avons mis en oeuvre dans ce plugin.

5.2.2 Inspector

L'*Inspector* correspond au menu permettant de paramétrer un `Process`. On retrouve ici les fichiers concernant l'interface utilisateur liée au `Process` que nous avons créé. Chaque `Process` ayant des caractéristiques différentes il faut donc spécifier les éléments appartenant à l'interface de chaque `Process` indépendamment. Nous avons implémenté dans la source `AppInteraction-ProcessInspector` 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. Les widgets (partie 5.2.4) et les variables créés à cette occasion sont liés aux commandes développées dans le répertoire `Commands` (partie 5.2.3). La figure 10 présente la partie de l'inspecteur correspondant aux paramètres de l'`AppInteraction` dans la version actuelle d'*i-score* et de notre plugin.

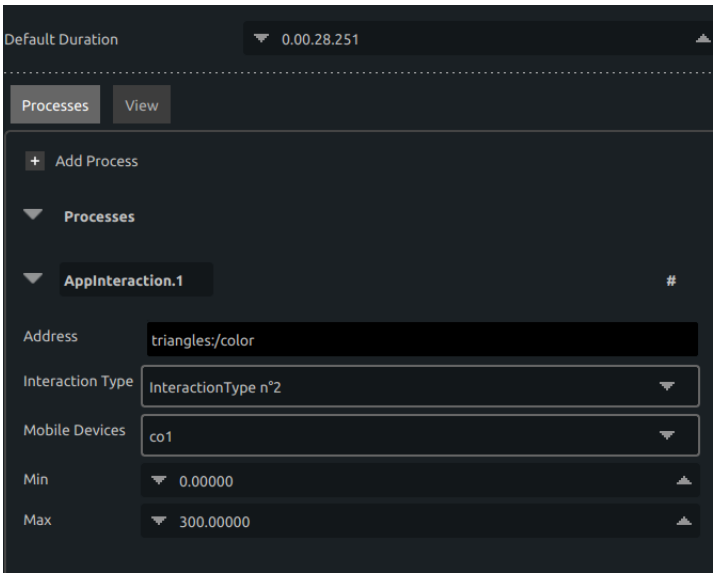

FIGURE 10 – Paramétrage d'une `AppInteraction` dans l'inspecteur

Notons que 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 notamment utilisés des signaux. Les signaux sont des outils fournis par Qt qui permettent la communication entre objets. Dans l'inspecteur, 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 peut mettre à jour l'indice du dispositif mobile sélectionné, le nouvel indice étant porté par le signal.

Les signaux ont été utilisés à de multiples reprises lors de l'implémentation du plugin. Par exemple, la classe `AppInteractionPresenter` du dossier `AppInteraction/Process/Layer` l'utilise pour afficher l'adresse du *device* que l'interaction doit modifier sur la partition en elle-même et non uniquement dans l'inspecteur.

5.2.3 Commands

Le dossier `Commands/` contient toutes les sources liées aux commandes de modifications du `Process`. Prenons l'exemple de `ChangeAddress.cpp`. Chaque type de `Process` peut être lié à un paramètre d'un logiciel externe dont les adresses sont contenues dans un arbre : c'est ce paramètre que l'on souhaite modifier grâce à l'interaction avec l'application. Il peut par exemple s'agir de la couleur d'un visuel généré par un logiciel comme Processing.

De cette manière, si on désire changer la cible de notre `Process`, en d'autres termes si l'on souhaite modifier un autre paramètre, il faut changer l'adresse du champ prévu à cet effet. Il nous faut une commande qui prenne en charge ce changement *i.e* qui remplace l'ancienne adresse par la nouvelle. Le header de ce fichier contient en particulier les attributs suivants :

— les informations relatives à l'ancienne adresse,

— les informations relatives à la nouvelle adresse.

La classe est déclarée comme une commande dès le header : elle hérite de la classe `i-score::Command`. On retrouve également dans le fichier source deux méthodes `undo` et `redo` qui sont des *features* d'*i-score* et qui permettent respectivement d'annuler et de réeffectuer le dernier changement. Dans le but, entre autres, d'avoir accès à ces deux méthodes et aux raccourcis clavier associés que chaque commande d'*i-score* possède sa propre classe.

En plus des fonctions `undo` et `redo` énoncées précédemment, des fonctions de sérialisation et désérialisation sont implémentées. Celles-ci permettent 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*. La sauvegarde des données est détaillée en partie 5.2.5.

Les commandes `ChangeInteractionType` et `ChangeMobileDevice` possèdent une structure et une implémentation proches de celles de `ChangeAddress`. `ChangeInteractionType` concerne le choix d'une IHM d'interaction pour l'application, tandis que `ChangeMobileDevice` concerne la sélection

du dispositif mobile que devra effectuer l'interaction. La différence avec `ChangeAddress` étant que le changement d'adresse s'effectue pour l'utilisateur en tapant la nouvelle adresse au clavier ou en effectuant un glisser-déposer, tandis que le choix d'IHM et de dispositif mobile se font grâce à des menus déroulants. Ces menus sont implémentés sous forme de *Widgets*, comme détaillé en partie 5.2.4.

En ce qui concerne les commandes permettant de modifier le *min* et le *max*, respectivement `SetAppInteractionMin` et `SetAppInteractionMax`, seul un header contenant la déclaration de la commande est nécessaire. Nous avons préféré rester cohérents avec l'implémentation des champs relatifs aux *min* et *max* dans les autres types de `Process`, ils ne possèdent donc pas les méthodes `undo` ou `redo`.

5.2.4 Les *Widgets*

Les *Widgets* correspondent dans notre cas 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, correspondant à tous les items sélectionnables du menu déroulant et un objet de type `QHBoxLayout` permettant 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 offrant la possibilité de récupérer et modifier l'indice du type d'interaction sélectionné, cet indice étant un attribut de l'objet `QComboBox`.

Le widget fonctionne comme suit : 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'inspecteur détecte pour être informé du nouveau type d'interaction sélectionné.

La création de ce widget n'était pas sous-entendu dans l'arborescence de fichiers initialement fournie par le client. Mais après avoir observé le comportement d'autres `Process` comme l'`Automation`, il nous a paru évident que nous en avions besoin afin d'obtenir un visuel cohérent avec *i-score* en lui-même et de manier correctement nos données. Pour implémenter les widgets, nous nous sommes inspirés de la structure des widgets existants dans *i-score*, comme le `UnitWidget` qui permet de choisir une unité en fonction du type de données manipulé. Nous avons donc pu récupérer l'architecture de ce widget, puis l'avons modifié pour tester son comportement et se l'approprier, avant de pouvoir implémenter nos propres widgets.

5.2.5 ProcessModel

La classe `ProcessModel` est une sorte de classe *de référence* du `Process`, elle contient des données relatives au `Process`. Elle implémente entre autres des getters et setters qui permettent de manier ces données. L'instance de `ProcessModel` du `Process` est notamment passée comme paramètre au constructeur de l'`Inspector` et lui permet d'accéder aux données qu'elle conserve.

Dans le code du `ProcessModel` sont également implémentées une méthode `read` et une méthode `write`, *i-score* peut donc lire et écrire un fichier : on peut donc grâce à ces méthodes sauvegarder l'état d'un process à l'enregistrement d'un fichier et récupérer cet état au chargement du fichier. Nous avons complété ces méthodes afin qu'elles puissent également manier les données que nous utilisons dans `AppInteraction` : type d'interaction sélectionné, adresse de la variable à modifier, etc.

5.2.6 Connection

Ce dossier contient toutes les sources concernant la connexion des appareils mobiles. D'une part il contient un fichier `Connection.cpp` décrivant la connexion d'un appareil : le nom de l'appareil, la fonction permettant d'envoyer la demande d'interaction. Nous avons également implémenté un fichier `ConnectionManager.cpp` qui lui offre la possibilité de gérer les appareils mobiles connectés, il peut obtenir le nombre d'appareils connectés ou encore la liste de ces appareils.

Pour pouvoir assurer nos tests, nous avons également dupliqué ces fichiers sous la forme de faussaires permettant de simuler la présence d'appareils sans qu'il y ait de réelles connexions établies, comme expliqué en partie 7.

La connexion entre l'application mobile et *i-score* est plus amplement expliquée en partie 6.

5.2.7 Executor

Ce sous dossier de `Process` 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`.

Dans le cadre de notre travail avec un appareil mobile, nous avons ajouté une méthode `interactionValueReceived()` : lorsqu'une connexion envoie un signal indiquant qu'une donnée provenant de l'application a été reçue, cette méthode est appelée automatiquement. Elle a en effet été connectée à ce signal dans le constructeur de l'`Executor`. Elle va pouvoir calculer la valeur à envoyer aux *devices* en utilisant la valeur reçue de l'application comme un ratio, comme expliqué en partie 5.2.2. 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 (dans le cas où ces valeurs reçues appartiennent bien à $[0, 1]$). 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`. Les types restants ne nous on pas paru essentiels pour le moment : les valeurs reçues sont alors 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.

`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 transmis. Enfin elle stocke ce message dans un attribut du `ProcessExecutor` de type `State::Message`. 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 qui sera ensuite traité et envoyé par *i-score*.

5.2.8 DocumentPlugin

La classe `DocumentPlugin` nous a été utile pour résoudre le problème suivant : chaque `Process` doit posséder une seule et unique entité de `ConnectionManager`, instanciée dès la création du `Process` : elle permet d'ouvrir des connexions et de fournir la liste des dispositifs mobiles connectés à tout endroit du code de l'`AppInteraction`.

Une unique instance de `DocumentPlugin` est automatiquement créée pour chaque instance du `Process`. 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.

On peut alors récupérer un pointeur vers le `ConnectionManager`, que nous avons donc choisi de conserver comme attribut du `DocumentPlugin`. Ce mécanisme est utilisé en particulier dans l'`Executor`, classe qui est détaillée en partie 5.2.7 : celui-ci a besoin du `ConnectionManager` pour récupérer la liste des `Connection`, transmettre par la suite 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.

5.3 Avancée globale du plugin

Actuellement, le plugin implémente les fonctionnalités suivantes dans *i-score* :

- Créer un `Process` du type `AppInteraction` ;
- Ajuster la durée de ce `Process` ;
- Définir l'adresse du *device* à « contrôler » ;
- Définir le dispositif mobile connecté qui reçoit la demande d'interaction ;
- Définir le type d'IHM affiché sur l'application lorsque l'utilisateur interagit ;
- Définir des valeurs minimum et maximum à envoyer au *device* ;
- Modifier tous les paramètres énoncés précédemment ;
- Lire une partition contenant un/des `Process` de type `AppInteraction`, la mettre en pause, la stopper, la rejouer ;
- Envoyer une valeur à l'adresse du *device* (valeur pour l'instant définie par les faussaires des classes de connexion) ;
- Enregistrer une partition au format `.scorejson` contenant une `AppInteraction` ;
- Charger une partition enregistrée au format `.scorejson` contenant une `AppInteraction` et récupérer les données de ce `Process` : adresse du *device*, IHM, min, max, etc.

La figure 11 correspond au rendu visuel d'une partition contenant un `Process AppInteraction`.

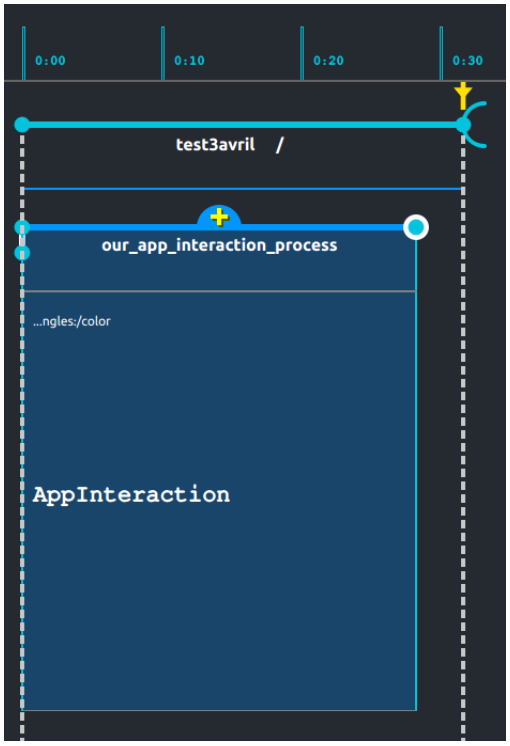

FIGURE 11 – Partition contenant un `Process AppInteraction`

5.4 Comparaison avec le document de spécifications

Lors de l'établissement du document de spécifications, nous avons fortement sous-estimé le degré de complexité du logiciel *i-score* et de la création d'un plugin compatible avec ce logiciel. Les cas d'utilisation traitant de l'utilisation d'*i-score* sont les cas « 6 : TRANSMETTRE LES DEMANDES D'INTERACTION » et « 7 : MANIPULER LES NOUVEAUX TYPES D'INTERACTIONS » ne semblent pas aujourd'hui pouvoir se suffire à eux-mêmes. Le CAS 3 concerne également le plugin, mais sera développé dans la partie connexion (partie 6).

Le scénario du cas 7 stipulait l'ajout du `Process AppInteraction` dans une partition, le choix du nombre de téléphones exécutant l'interaction ainsi que d'éventuelles conditions supplémentaires. Nous n'avions alors, pas même évoqué la présence d'un lien à établir entre les *device* et le `Process`, ni même l'existence d'un choix d'IHM ou la nécessité de pouvoir sauvegarder les données de la partition dans un fichier.

Le cas 6 précise simplement la nécessité de l'envoi d'une demande d'interaction à un instant précis. Nous savions déjà que l'instant d'envoi de cette demande serait à définir dans une version

plus avancée d'*i-score*, Monsieur CELERIER nous ayant prévenu de la complexité de la tâche : *i-score* ne peut pas aujourd'hui envoyer des données concernant un `Process` avant que l'on arrive au début de la lecture du `Process` en question. La version actuelle du plugin permet donc d'envoyer la demande d'interaction au moment où la partition commence à lire l'`AppInteraction`, soit dès que l'interaction doit être exécutée par l'utilisateur de l'application, comme expliqué en partie II.4 du document de spécifications. Comme stipulé dans ce cas 6, cette demande est bien envoyée à un appareil mobile sélectionné par le compositeur au préalable. Le message exact envoyé est décrit en partie 6.

La réalisation du cas 7 est quant à elle plus floue : il est bien possible de créer un `Process` `AppInteraction` et de le modifier. Cependant, les paramètres modifiables ne sont pas ceux décrits dans le document de spécifications. Nous avons préféré nous focaliser sur le cas basique où chaque `AppInteraction` serait associée à un seul et unique dispositif mobile et étendre les possibilités de paramétrage afin de gérer ce `Process` de façon plus cohérente : l'adresse du *device* est indispensable, tandis que le min et le max correspondent à une volonté d'homogénéité avec *i-score* et de pertinence des valeurs transmises. L'attribution automatique et conditionnelle des interactions parmi plusieurs dispositifs mobiles connectés comme elle est décrite en page 16 (partie II.4) du document de spécifications n'a donc pas été implémentée. Il revient donc au compositeur de définir explicitement quel mobile est associé à quelle interaction. Cela permet par exemple au compositeur de cibler un dispositif mobile comme celui gérant le son et un autre comme gérant l'image, ce qui peut se révéler très intéressant si le compositeur a connaissance des capteurs disponibles sur les dispositifs mobiles connectés. Un dispositif mobile possédant un capteur de mouvement non fonctionnel pourra être utilisé plutôt sur des IHM de type *slider*, tandis qu'un autre possédant des capteurs dernier cri pourra utiliser une interface créée spécialement pour lui.

Autre point qui n'est que sous-entendu dans le document de spécifications : le plugin doit être capable de traiter les données envoyées en continu par l'application. Un problème reste présent aujourd'hui avec les faussaires : 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. Il n'est donc aujourd'hui possible de transmettre qu'une seule valeur par le biais des faussaires.

La possibilité de sauvegarder une partition et de la recharger après fermeture du fichier ne sont pas non plus des fonctionnalités accessoires et ont donc été implémentées. Elles ont également pu faciliter les phases de tests du plugin.

Enfin, dans un souci de cohérence visuelle et de confort du compositeur, nous avons décidé d'afficher l'adresse du *device* sélectionné sur la partition : si l'inspecteur est fermé, il est encore possible de voir rapidement le paramètre contrôlé par l'interaction.

6 Connexion

Notre projet se découpe en deux parties distinctes : le plugin et l'application. Afin de faire le lien entre ces deux entités lors de l'utilisation, nous avons dû mettre en place un protocole de connexion, depuis l'ouverture de la connexion sur le réseau entre deux entités jusqu'à l'échange de données en temps réel.

6.1 Objectifs

6.1.1 Protocole de connexion

La première chose à faire pour permettre au séquenceur et l'application d'entrer en contact sur le réseau était de définir un protocole de connexion. Une fois les deux entités connectées au réseau, l'application peut lancer une recherche de service *i-score*. Si un séquenceur est disponible, l'association a lieu. La connexion est alors ouverte et l'application écoute sur le réseau.

6.1.2 Modèle d'échange de données

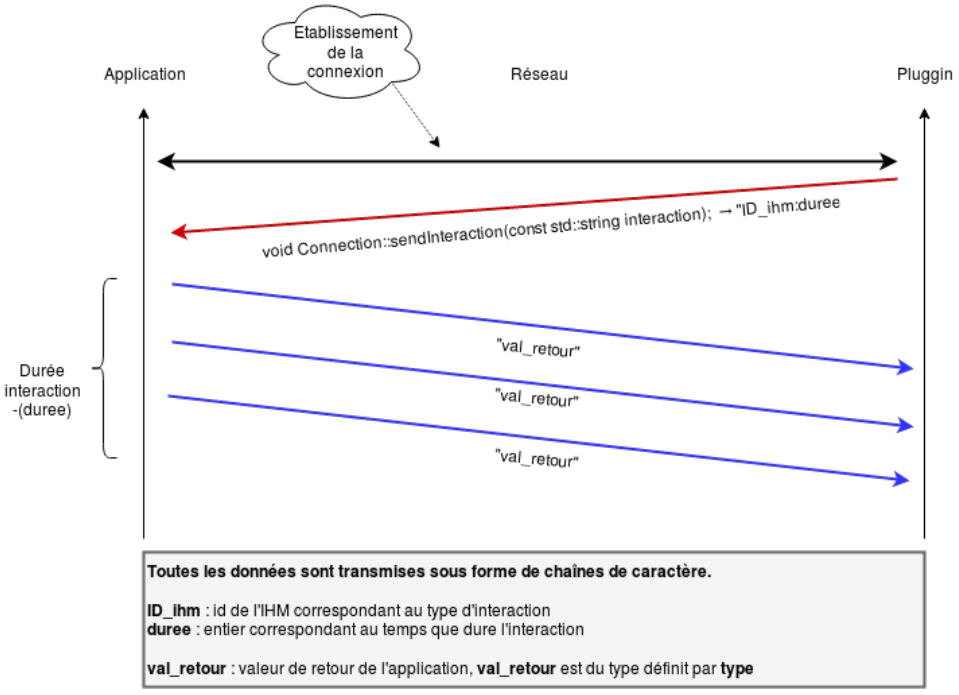

FIGURE 12 – Diagramme d'échange de données

Nous avons dans un second temps dû définir un modèle d'échange de données entre le séquenceur et l'application. La connexion étant établie, c'est le séquenceur qui donne l'ordre à l'application de transmettre des données via un message caractérisant une interaction.

Pour chaque interaction, le séquenceur fournit à l'application l'indice de l'IHM choisie par le compositeur et la durée de transmission des données

Une fois le message d'interaction reçu, l'application va donc, durant toute la durée d'interaction, transmettre des données récupérées via l'IHM.

Le type de données envoyé est `ossia::value` : il est donc fourni par l'API OSSIA. Il peut renfermer des valeurs de nombreux types : `int`, `float`, `array<float,3>`, etc. *I-score* est ensuite capable de convertir ces données en le type demandé par le *device* sélectionné.

6.2 Réalisation

6.2.1 Architecture côté serveur (*i-score*)

Nous avons défini deux classes qui permettront de prendre en charge les connexions du côté d'*i-score*. La première est la classe `Connection` qui représente une application connectée au logiciel. On instanciera cette classe à chaque fois qu'une nouvelle connexion est détectée, pour pouvoir la gérer. La deuxième classe, `ConnectionManager`, est justement l'entité qui se charge d'instancier les `Connection`, et de gérer le serveur principal. La classe `ConnectionManager` n'est instanciée qu'une seule fois par document ouvert dans *i-score*. Elle possède un tableau contenant toutes les instances de `Connection` pour que le code principal puisse accéder à l'application souhaitée et lui envoyer une interaction par exemple.

6.2.2 Architecture côté client (application)

Dans l'application, nous avons une classe `ClientConnection` qui prend en charge l'établissement de la connexion avec le serveur.

6.2.3 Implémentation du protocole de connexion

Tout d'abord nous avons choisi d'utiliser l'API OSSIA fournie par le client qui, étant écrite spécialement pour *i-score*, permet d'automatiser certains concepts sur lesquels *i-score* repose, notamment la représentation des devices en arbre. Ainsi, on garantit la compatibilité entre les deux côtés de la connexion. En revanche, OSSIA n'a pas été conçue pour la recherche de serveur sur le réseau, ce qui nous oblige à trouver une manière d'établir une connexion entre le client et le serveur dans un premier temps afin de pouvoir mettre en place la communication via OSSIA dans un second temps. Nous avons donc établi le protocole décrit dans le paragraphe suivant.

Pour chaque nouveau document ouvert dans *i-score* on met en place dans la classe `ConnectionManager` un serveur TCP à l'aide de la classe `QTcpServer` fournie par Qt. Ce serveur écoutera le réseau de manière permanente pour qu'une application puisse s'y connecter n'importe quand. L'application pourra utiliser un protocole de recherche de service sur le réseau (comme Zeroconf) pour détecter la présence du serveur et récupérer l'adresse IP et le numéro de port. Une fois qu'une application s'est connectée sur le serveur TCP, on lui envoie un message comprenant l'adresse IP ainsi que le numéro de port sur lesquels ou va ouvrir le serveur dédié à la communication. On instancie alors un type de serveur fourni par OSSIA, un `ossia::oscquery::oscquery_server_protocol`.

Du côté de l'application, lorsque celle-ci reçoit le message venant d'*i-score*, elle instancie un `ossia::oscquery::oscquery_mirror_protocol` qui est la classe client du protocole `oscquery`.

L'application va ensuite pouvoir construire son arbre et l'envoyer sur le réseau pour que le serveur le récupère. L'arbre sera de la forme suivante :

\ interaction
  | "nom interaction 1"
  | "nom interaction 2"
  .
  .
  .

C'est donc un modèle où chaque client a son propre serveur et où la communication repose sur le protocole `oscquery` implémenté par OSSIA. Lorsque les classes serveur et client sont instanciées et que l'arbre est envoyé, la connexion est établie et on peut commencer à échanger des données.

6.2.4 Implémentation du modèle d'échange de données

La communication est toujours initiée par *i-score* puisque l'application est constamment en attente d'un message du serveur. Lors de l'envoi d'une interaction à une application, on procède de la manière suivante : d'abord *i-score* envoie un message précisant quelle IHM afficher et pendant combien de temps la jouer. Ensuite, on ajoute un `callback` sur le noeud de l'arbre correspondant à l'interaction en question, c'est-à-dire qu'on observe la valeur de ce noeud et si cette valeur change, on peut exécuter du code. Dans ce cas, on émet un signal qui transmet la nouvelle valeur du noeud. Ce procédé permet donc de suivre les modifications apportées par l'application en temps réel.

6.3 Avancée globale de la connexion et comparaison

À ce jour, seul l'établissement de la connexion est fonctionnel. On peut faire tourner *i-score* sur une machine, entrer en dur l'adresse IP et le numéro de port du serveur dans le code de l'application et après pression du bouton `connect` de l'application, celle-ci reçoit un message de bienvenue du logiciel. Nous avons passé du temps à essayer de mettre en place l'échange sans toutefois aboutir à une version stable.

```
Debug: New Connection named first connection opened. (ConnectionFaussaire:10,
connectionFaussaire::ConnectionFaussaire::ConnectionFaussaire(std::__cxx11::string))
Debug: New Connection named second connection opened. (ConnectionFaussaire:10,
connectionFaussaire::ConnectionFaussaire::ConnectionFaussaire(std::__cxx11::string))
Debug: TODO (EventInspectorWidget:198, Scenario::EventInspectorWidget::EventInspectorWidget(const
Scenario::EventModel&, const iscore::DocumentContext&, QWidget*))
Debug: New Connection named first connection opened. (ConnectionFaussaire:10,
connectionFaussaire::ConnectionFaussaire::ConnectionFaussaire(std::__cxx11::string))
Debug: New Connection named second connection opened. (ConnectionFaussaire:10,
connectionFaussaire::ConnectionFaussaire::ConnectionFaussaire(std::__cxx11::string))
Debug: Interaction msg : 1:5,660819, sent to first connection.  (ConnectionFaussaire:24, void
connectionFaussaire::ConnectionFaussaire::sendInteraction(std::__cxx11::string))
Debug: Interaction index : 1, Interaction duration : 5.000000 (ConnectionFaussaire:29, void
connectionFaussaire::ConnectionFaussaire::sendInteraction(std::__cxx11::string))
Debug: Interaction msg : 1:24,879258, sent to second connection.  (ConnectionFaussaire:24, void
connectionFaussaire::ConnectionFaussaire::sendInteraction(std::__cxx11::string))
Debug: Interaction index : 1, Interaction duration : 24.000000 (ConnectionFaussaire:29, void
connectionFaussaire::ConnectionFaussaire::sendInteraction(std::__cxx11::string))
Debug: Interaction msg : 2:23,159316, sent to first connection.  (ConnectionFaussaire:24, void
connectionFaussaire::ConnectionFaussaire::sendInteraction(std::__cxx11::string))
Debug: Interaction index : 2, Interaction duration : 23.000000 (ConnectionFaussaire:29, void
connectionFaussaire::ConnectionFaussaire::sendInteraction(std::__cxx11::string))
```

FIGURE 13 – Exemple d'affichage par des `qDebug()` lors de l'exécution d'une partition contenant des `AppInteraction`.

7 Tests et Intégration

7.1 Tests système

De nombreux tests ont été mis en place tout au long du développement du plugin et de l'application.

Dans le plugin a été très largement utilisée la commande `qDebug()`, qui permet d'afficher des messages d'erreur sur la sortie de l'application. Nous avons alors pu afficher toutes sortes de données, afin de vérifier leur cohérence. Aujourd'hui, de nombreux `qDebug()` sont toujours en place, les faussaires étant toujours utilisés pour faire fonctionner le plugin. Il est donc possible de vérifier le bon déroulement du programme. Des outils d'*i-score* ont également été utiles, comme le panel `Messages` qui permet de surveiller l'envoi et la réception de messages entre *i-score* et les logiciels externes.

En ce qui concerne l'application, la fonction `console.log()`, permettant d'afficher des messages sur la sortie de l'application tout comme `qDebug()`, a été utilisée pour obtenir les valeurs des variables dans les extraits de code Javascript, c'est-à-dire là où les objets QML propagent des signaux. Par exemple, lors de sa création, un objet QML propage un signal `Completed` ; un objet bouton propage un signal `Pushed` lorsque l'on appuie dessus. Nous avons également ajouté de manière temporaire un bouton `Simulate an interaction` sur l'écran d'attente avec le menu des interactions, afin de simuler la réception d'une demande d'interaction depuis le séquenceur. D'autres tests ont également été réalisés en modifiant le contenu du fichier JSON.

7.2 Intégration

Des classes faussaires utilisées à la place des classes `Connection` et `ConnectionManager` ont pour but de vérifier le fonctionnement du plugin, indépendamment de l'état de marche des vraies classes de `Connection`.

Elles ont donc pu mettre en lumière des défauts de fonctionnement du plugin, en simulant une réelle connexion avec l'application. Par exemple, un segfault apparaissait si, après une première lecture, on relancait une partition. Il se trouve que l'instance de `ProcessExecutor` est automatiquement détruite dès lors qu'on stoppe la lecture, mais que ses méthodes étaient toujours appelées à la réception de signaux : il a donc fallu, dans son destructeur, déconnecter cette instance des signaux qu'elle détectait.

Nous avons également pu, grâce à ces faussaires, comprendre une erreur que nous avions commise : pour pouvoir utiliser les signaux dans les classes de connexions, nous avons dû les faire hériter de la classe `QObject`. Cependant, nous essayons d'utiliser directement des instances de `Connection` et `ConnectionManager` sans passer par des pointeurs. Cette pratique était problématique, puisque l'utilisation de vecteurs de `Connection` induisait une copie des instances utilisées, or un `QObject` ne peut pas être copié.

Aujourd'hui, les faussaires permettent de créer deux fausses connexions, et d'envoyer une valeur lorsqu'une partition est jouée. On a ainsi pu modifier la couleur ou encore la densité des formes géométriques d'une exécution du logiciel Processing. Notons que nous avons utilisé un fichier exemple d'utilisation de Processing fourni dans les sources d'*i-score* pour lancer Processing. Les figures 14 et 15 exposent un exemple d'utilisation des process `AppInteraction` pour modifier plusieurs paramètres du programme exécuté par Processing. Nous avons également utilisé le panel « Messages » de *i-score*, qui permet de voir les messages envoyés et reçus par *i-score* : lorsqu'un message est envoyé à un logiciel externe, la destination et la valeur sont clairement affichées.

La principale partition utilisée pour tester le `Process AppInteraction` est le fichier `test_dataspace_processing.scorejson`. Il est disponible sur le dépôt dans `addon/fichiers_tests/`. Un fichier `.scorebin` s'y trouve également : ce fichier-ci nous a permis de réaliser que ce second format était mal pris en charge par notre plugin. La prise en charge est donc aujourd'hui partielle, puisque le paramétrage exact des `AppInteraction` ne peut être récupéré via ce format.

7.3 Tests recettes

Les tests de recettes n'ont pas été mis en place car le plugin et l'application ne sont pas encore capables de communiquer correctement.

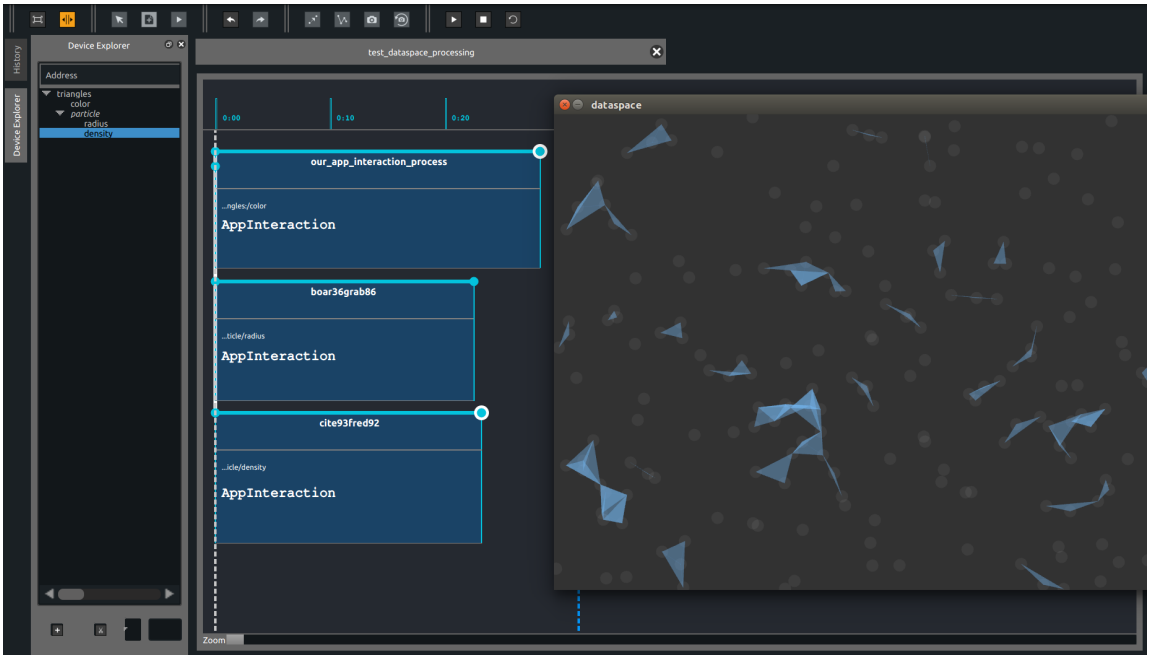

FIGURE 14 – Partition *i-score* et affichage de Processing avant lecture de la partition

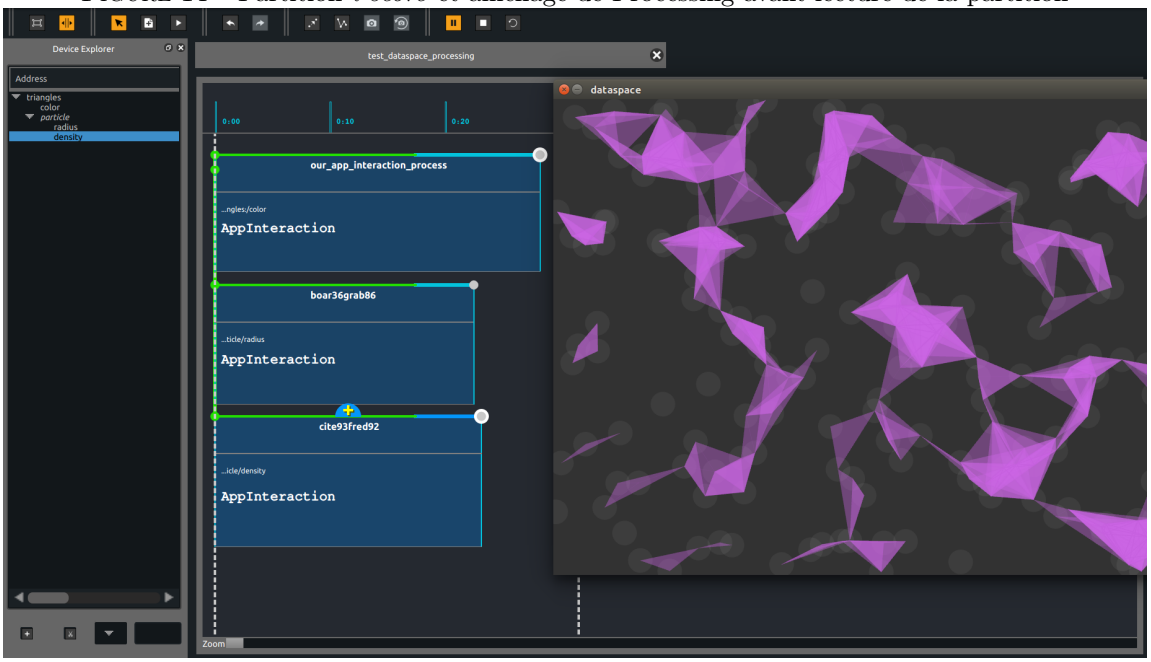

FIGURE 15 – Lecture de la partition et affichage de Processing modifié

8 Rétrospective générale

Ce projet nous a donné la possibilité de mettre en place des cycles de production. En effet, la longévité et l'ampleur de celui-ci convenait à l'application de cette méthode, qui s'est révélée très utile pour la progression de notre travail.

Nous nous sommes rendus compte que lors du déroulement du projet, tout ne se passait pas comme le planning le prévoyait. Pour l'illustrer, nous pouvons reprendre le diagramme de Gantt de la figure 18 créée pour le rendu du document de spécifications, que vous trouverez en annexe en page 38. Notons que nous n'avons pas pris en compte certaines tâches importantes telles que l'implémentation des tests ou encore l'intégration, l'écriture du rapport, du manuel d'utilisation et du manuel de maintenance.

De plus, les périodes fixées sur le diagramme de Gantt n'ont pas du tout été celles adoptées lors du développement. En effet, lorsque la phase d'implémentation a débuté, nous avons réévalué l'importance de certaines tâches. Le principal exemple que l'on peut citer est la création du plugin. Alors que nous avions planifié 1 mois pour la production de celui-ci, la réalité en a été différente puisque dès le début, l'implémentation de ce plugin a été une de nos trois principales tâches.

La première phase de mise en place de l'environnement prise en compte dans la tâche « Prise en main des outils de développement »a également pris plus de temps qu'on l'aurait souhaité, ce qui a retardé le réel commencement du projet.

Le temps de s'approprier le code du logiciel *i-score* a considérablement prolongé les délais que nous avions prévus, en particulier pour le plugin et la connexion qui demandaient une familiarisation avec les outils de développement. Lors de l'implémentation de la communication avec les `devices` par exemple, nous avons passé beaucoup de temps à comprendre et nous approprier le fonctionnement des adresses de *devices* et des `ossia::value`, qui sont tous deux utilisés lors de l'envoi de messages. Nous avons alors parfois dû nous en remettre à notre client pour pouvoir comprendre et avancer.

Le principal problème que nous avons rencontré concernant la connexion a été de comprendre comment est conçue OSSIA et ce qu'elle permet de faire, ainsi que ce qu'elle ne permet pas de faire. Il nous a fallu du temps avant de comprendre qu'il fallait utiliser un serveur TCP de Qt pour amorcer la connexion car OSSIA ne permet pas de rechercher et d'accepter des connexions de manière flexible. Il a aussi été difficile d'appréhender l'échange de données, c'est-à-dire savoir quelles classes utiliser et quel type de protocole mettre en place. Nous n'avons malheureusement pas eu le temps de terminer proprement la partie connexion après avoir réussi à obtenir une idée précise du modèle à adopter.

Du côté de l'application, de nombreux problèmes de compréhension du langage QML nous ont empêché d'avancer au début. En effet, ce langage a la particularité d'être très différent de tous ceux qu'on a utilisés jusqu'à maintenant, car pour pouvoir coder une interface homme-machine avec ce langage, il est nécessaire de créer une arborescence d'objets graphiques, et chaque action sur un objet envoie un signal qu'il faut pouvoir gérer. De plus, chaque objet graphique est codé comme une liste d'attributs standardisés : il a donc fallu apprendre à utiliser chacun des attributs dont nous avions besoin. En outre, ce langage est très récent et très peu utilisé : peu de documentation et de forums étaient disponibles sur Internet pour nous aider. Enfin, la connexion et l'échange de données entre le contrôleur et la vue d'une part, et le modèle et la vue d'autre part, ont été très laborieux et nous ont demandés plusieurs semaines de travail. Notamment, pour l'envoi de données

du contrôleur vers la vue, nous étions partis initialement sur une solution qui semblait fonctionner, mais s'est avérée inadaptée après un long temps d'utilisation : il a fallu réécrire tout le code en utilisant les signaux.

Au fur et à mesure de l'avancée du projet, l'objectif principal a convergé vers une implémentation fonctionnelle avec un minimum de possibilités, notamment pour que les échanges puissent être testés. C'est la raison pour laquelle lors de la 13e semaine du projet, il a fallu moduler les équipes de travail, en particulier pour renforcer l'équipe s'occupant de l'échange de données.

9 Conclusion

Ce projet nous a beaucoup apporté en matière de gestion d'équipe, notamment en ce qui concerne la communication et la répartition des tâches. C'était également la première fois que nous avions affaire à une personne ayant le statut de *client*, tout en conservant une communication régulière avec le responsable pédagogique. Nous avons donc appris à gérer un projet dans ces circonstances. Nous avons également pu acquérir des connaissances en matière d'applications mobiles, de communication réseau, ou encore concernant le framework Qt.

Malgré les difficultés d'amorçage du développement que nous avons rencontrées, nous avons essayé de mener ce projet enrichissant au mieux. Le manuel d'utilisation et de maintenance que vous trouverez en annexe vise à ce qu'une prochaine équipe, potentiellement à l'occasion d'un PFA, puisse reprendre ce projet sans problème de compréhension et puisse le mener à son terme.

10 Annexes

10.1 Images

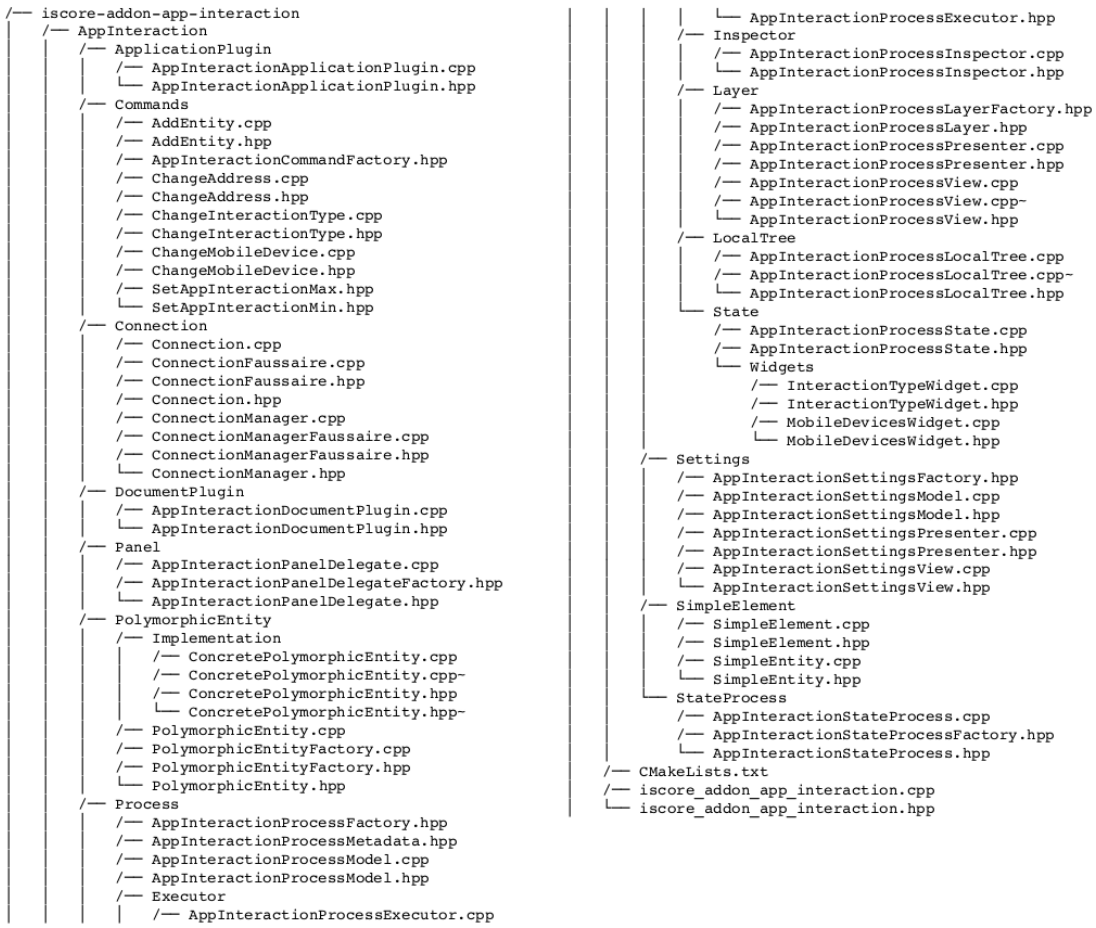

FIGURE 16 – Arborescence du plugin

	Nom de la tâche	Durée
	Exécution complète du projet	18 semaines
1	Implémentation du plugin	18 semaines
1.1	Prise en main des outils de développement	5 semaines
1.2	Familiarisation avec le tutoriel fourni	8 semaines
1.3	Réalisation d'une version 0	11 semaines
1.3.1	Création d'une interaction fonctionnelle	7 semaines
1.3.2	Communication avec un logiciel externe	5 semaines
1.3.3	Gestion des appareils connectés	2 semaines
1.4	Implémentation d'un faussaire de connexion	4 semaines
2	Implémentation de l'application	18 semaines
2.1	Architecture	4 semaines
2.2	Prise en main des outils de développement	4 semaines
2.3	Réalisation d'une version 0	10 semaines
2.3.1	Interaction C++ / QML	4 semaines
2.3.2	Création du menu listant les interactions	5 semaines
2.3.3	Création de la vue d'une interaction	3 semaines
2.3.4	Récupérer des données saisies par l'utilisateur	5 semaines
2.3.5	Implémentation des fonctions de connexion	2 semaines
3	Implémentation du protocole de connexion	18 semaines
3.1	Prise en main des outils de développement	12 semaines
3.2	Établissement d'un schéma de connexion	3 semaines
3.3	Implémentation des fonctions d'échange	4 semaines
3.3.1	Côté plugin	3 semaines
3.3.2	Côté application	5 semaines
3.4	Connexion entre les deux entités	1 semaine
3.5	Architecture	3 semaines
4	Rédaction du rapport et des manuels	2 semaines
5	Intégration et préparation du code au rendu	1 semaine

FIGURE 17 – Diagramme de Gantt actualisé

FIGURE 18 – Diagramme de Gantt du document de spécification

10.2 Document de spécifications

PFA - Application tactile pour les arts interactifs

Document de spécifications

Client :
CELERIER Jean-Michaël

Responsable pédagogique :
ROLLET Antoine

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

1er décembre 2016

Table des matières

Introduction

Ce projet s'inscrit dans le cadre de notre deuxième année d'études à l'ENSEIRB-MATMECA et sur une durée d'environ six mois. Ici, il s'agit de créer une application mobile pour Android et iOS. Cette application doit permettre d'effectuer des interactions sur des partitions musicales et visuelles créées à partir du logiciel i-score, développé par Blue Yeti / LaBRI. Afin de pouvoir faire communiquer l'application avec le logiciel, nous devons également implémenter un plugin pour i-score. L'objectif principal de ce projet est de permettre au compositeur et aux interprètes de tous pouvoir prendre part au jeu de scène.

i-score est un séquenceur qui permet à l'utilisateur non seulement de créer des partitions, mais aussi de définir au cours de cette dernière des interactions. Nous appelons "interaction" l'action menée par l'utilisateur de l'application, dans le but de modifier le spectacle de manière sonore ou visuelle. Par exemple, secouer le téléphone est une interaction qui peut engendrer une modification du volume sonore, ou encore une modification de l'intensité des projecteurs. Cette action est définie au préalable par le compositeur, lors de la conception de la partition. Notre application devra signaler les interactions possibles à l'utilisateur, qui pourra alors les réaliser. Comme tout ce qui touche au domaine des arts interactifs, un soin particulier doit être apporté à l'interface graphique, tant au point de vue de l'ergonomie qu'au niveau de l'esthétique. De plus, les temps de réponse doivent être minimaux afin de ne pas désynchroniser les différents éléments d'une partition.

Ce document de spécifications détaille notamment les différents besoins fonctionnels et non fonctionnels que nous avons établis, les différents types de capteurs et d'interactions que nous prévoyons d'utiliser et d'implémenter, une description des tests prévus, une maquette de l'application, ainsi que des diagrammes UML utiles à la compréhension.

Première partie
Étude de l'existant

Le logiciel i-score existant est un séquenceur intermedia permettant d'écrire des scénarios interactifs. L'utilisateur du logiciel, que nous appelons le compositeur, écrit une partition. Lors de l'exécution de celle-ci, i-score peut contrôler des logiciels satellites avec lesquels il communique par le biais du protocole OSC ou Minuit par exemple, et ce au fil des processus (ou événements) programmés par le compositeur. Ces évènements permettent de modifier des paramètres dans les logiciels annexes. Il peut s'agir de paramètres tels que la luminosité, le contraste ou encore le volume sonore. Les évènements peuvent être des listes de commandes, des automations, des triggers ...

La partition est donc définie pour chaque paramètre sous forme d'un fil conducteur. Nous allons implémenter de nouveaux événements à insérer dans cette partition en développant un plugin au logiciel i-score. Ces événements seront ajoutés et paramétrés par le compositeur et permettront à des personnes que nous appelons interprètes d'interagir avec le logiciel par le biais d'une application que nous allons créer.

Aujourd'hui, des interactions sont d'ores-et déjà permises par i-score. Par exemple, un trigger permet de programmer des événements conditionnels, dont l'allure simplifiée est la suivante : *"à t=5, si la luminosité est supérieure à 20, diminuer la luminosité à 15"* . Notre rôle est de permettre à une personne sur scène d'interagir avec i-score pendant un spectacle : il faut donc un moyen d'interaction en temps réel qui permet l'envoi des données pendant que la partition est jouée et qui n'oblige pas le compositeur à utiliser directement i-score sur un ordinateur pour modifier les données de son spectacle, afin qu'il puisse prendre part au jeu de scène. Les nouvelles interactions que nous allons implémenter permettront donc au compositeur d'écrire des scénarios de la forme *"de t=20 à t=25, le deviceA modifie le volume grâce à une jauge et le deviceB ajoute des effets sonores grâce à un launchpad"*, où deviceA et deviceB désignent deux dispositifs mobiles (smartphone, tablette, ...) sur lesquels est installée notre application.

Du côté de l'application, elle est à créer intégralement. Il s'agit principalement d'implémenter des interfaces pour chaque nouvelle interaction et de pouvoir communiquer avec i-score. Du côté i-score, il s'agit d'implémenter un plugin permettant d'ajouter les nouveaux types d'interactions et de communiquer avec l'application, en utilisant notamment les bibliothèques fournies par le client.

Deuxième partie
Besoins fonctionnels

Pour visualiser le travail à faire sur le projet, on distinguera les fonctionnalités à créer avec notre application et les différents cas d'utilisation que l'interprète et le compositeur pourront rencontrer lors de l'utilisation de l'application.

1 Fonctionnalités

Le séquenceur i-score doit être pourvu de nouvelles possibilités d'interactions à insérer dans les partitions. L'ensemble de ces interactions peut être scindé en deux catégories :

- Action de l'utilisateur sur l'écran du dispositif sur lequel est installée l'application, *ie* directement sur l'IHM.
- Mouvement du mobile/tablette, détecté par des capteurs (autres que l'écran) déterminés au préalable par le compositeur.

Ces interactions sont détaillées dans la partie besoins non fonctionnels. Il appartient au compositeur de choisir quel effet auront ces interactions sur le déroulement du spectacle, et ce en utilisant les fonctionnalités existantes du logiciel i-score. Notre rôle en ce qui concerne le logiciel i-score est uniquement de permettre au compositeur d'ajouter nos interactions dans ses partitions.

En accord avec les cas d'utilisation, l'application doit être capable d'obtenir et traiter les données reçues depuis i-score, afficher les IHM nécessaires lors de demandes d'interactions et envoyer les résultats des interactions à i-score. En ce qui concerne les interactions, chacune doit posséder sa propre IHM dans l'application. Cependant, il est également nécessaire que l'utilisateur de l'application dispose des fonctionnalités suivantes quelle que soit l'interaction en cours :

1. Annoncer la fin d'une interaction avant la fin du temps imparti (qu'elle ait été effectuée ou non).
2. Informer le séquenceur que l'interaction a été effectuée mais n'a pas été détectée par les capteurs relatifs à celle-ci.
3. Se déconnecter.
4. Passer l'application en arrière plan (pour ouvrir une autre application par exemple) et pouvoir reprendre normalement, sans avoir à se reconnecter.

2 Cas d'utilisation

2.1 Diagramme des cas d'utilisation

L'utilisation de l'application implique différents cas d'utilisation concernant deux acteurs : l'utilisateur et le séquenceur i-score.

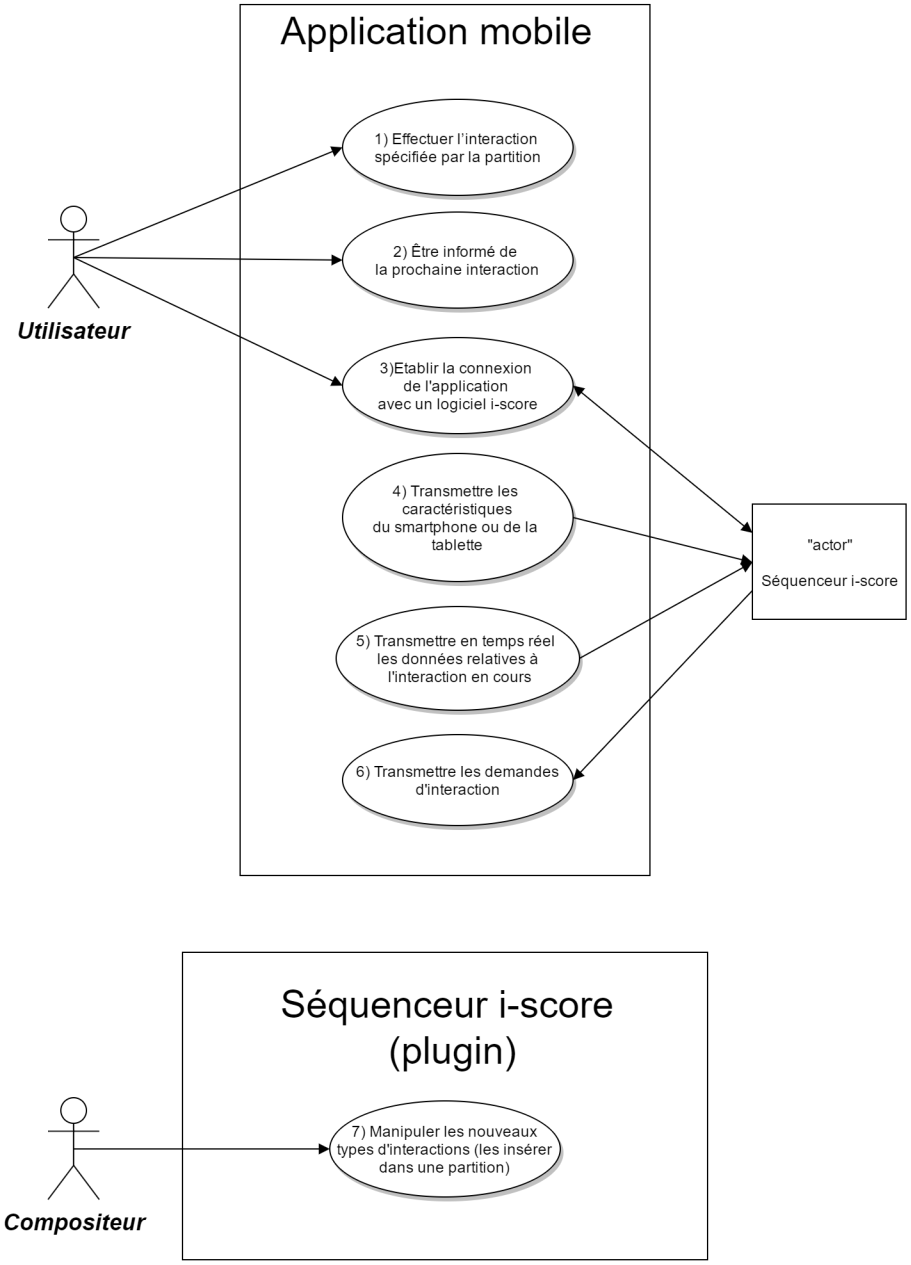

FIGURE 1 – Diagrammes des cas d'utilisation

2.2 Description textuelle des cas d'utilisation

2.2.1 Résumé des cas d'utilisation

Système : Application Mobile

1. Effectuer l'interaction spécifiée par la partition
L'utilisateur agit sur les potentiomètres, curseurs...ou physiquement sur le dispositif pour obtenir un effet visuel (allumage, couleur de projecteur...) ou sonore (volume, musique, bruitage...).

2. Être informé de la prochaine interaction
L'utilisateur reçoit un message et une notification avec vibreur sur l'écran du dispositif indiquant qu'une interaction est attendue, puis il se prépare pour la prochaine interaction en lisant sa description pendant le compte à rebours.

3. Établir la connexion de l'application avec un logiciel i-score
L'utilisateur cherche le logiciel i-score auquel il va se connecter et lui envoie une demande de connexion. Ensuite, i-score ajoute le dispositif dans son arborescence.

4. Établir la connexion de l'application avec un logiciel i-score
L'application établit et envoie à i-score la liste des capteurs utilisables afin de connaître les mouvements réalisables sur chaque dispositif.

5. Transmettre en temps réel les données relatives à l'interaction en cours.
L'application communique à i-score les choix effectués par l'utilisateur pendant l'interaction sous forme de données provenant des capteurs et de l'IHM.

6. Transmettre les demandes d'interaction
Le séquenceur choisit le dispositif auquel la prochaine interaction doit être envoyée et l'envoie ou l'annule si aucun dispositif n'est prêt. Les applications mobiles concernées se préparent ensuite à afficher l'interaction.

Système : logiciel i-score

7. Manipuler les nouveaux types d'interactions
Le compositeur ajoute, modifie et supprime les nouveaux types d'interactions dans la partition.

2.2.2 Détail des cas d'utilisation pour le système : application mobile

- Cas 1 : Effectuer l'interaction spécifiée par la partition
  Résumé : L'utilisateur agit sur les potentiomètres, curseurs,...ou physiquement sur le dispositif, pour obtenir un effet visuel (allumage, couleur de projecteur...) ou sonore (volume, musique, bruitage...).
  Acteur humain : l'utilisateur
  Pré-condition : Une partition est en cours. Une interaction est attendue, l'IHM est adaptée à l'interaction attendue ou conforme au choix du compositeur. L'utilisateur peut avoir été

prévenu au préalable de l'interaction, si le compositeur l'a spécifié dans la partition.
Scénario nominal :

1. L'application propose différents objets graphiques tels qu'une jauge, un slider ou encore une palette de couleurs, avec un texte et des logos d'explication.
2. L'utilisateur agit sur les objets graphiques (ou en exécutant un mouvement sur son dispositif) : il modifie la jauge, appuie sur un/des boutons du launchpad, déplace le curseur de la palette de couleurs ou encore appuie sur un des divers boutons qui lui sont proposés.
3. L'utilisateur peut également avoir à effectuer des mouvements, spécifiés dans la description de l'interaction : secouer, incliner...
4. A la fin du temps imparti pour l'interaction, l'IHM disparaît et est remplacée par un écran d'attente.

Scénario alternatif :

1. L'application propose différents objets graphiques, tels qu'une jauge, un slider ou encore une palette de couleurs, avec un texte et des logos d'explication
2. L'utilisateur agit sur les objets graphiques (ou en exécutant un mouvement sur son dispositif) : par exemple il modifie la jauge, appuie sur un/des boutons du launchpad, déplace le curseur de la palette de couleurs, ou encore appuie sur un des divers boutons qui lui sont proposés. Ou il passe directement à l'étape suivante.
3. L'utilisateur appuie sur le bouton permettant de passer ou terminer une interaction.
4. L'écran d'attente de la prochaine interaction s'affiche.

Scénario d'exception 1 :

1. La connexion wifi s'interrompt brusquement
2. Le message suivant s'affiche " Problème de connexion ! Votre interaction va être reportée ou annulée définitivement. "
3. Lors du retour de connexion à l'instant $t + T(\textit{sans connexion})$, le temps imparti pour l'interaction n'est **pas encore écoulé** : l'utilisateur peut continuer d'exécuter son interaction, donc retour à l'étape 2 du scénario nominal.

Scénario d'exception 2 :

1. Voir étapes 1. et 2. du scénario d'exception 1
2. Lors du retour de connexion à l'instant $t + T(\textit{sans connexion})$, le temps imparti pour l'interaction est **écoulé** : interruption de l'interaction et retour à l'état initial (attente de demande d'interaction).

Post-condition : L'application doit récupérer les informations relatives aux mouvements effectués et les envoyer à i-score.

- Cas 2 : Être informé de la prochaine interaction
Résumé : L'utilisateur reçoit un affichage visuel d'une interface spécifique avec un compte à rebours indiquant le temps restant avant l'interaction avec son nom et un petit rappel des méthodes offertes pour interagir.
Acteur humain : l'utilisateur

Pré-condition : Une nouvelle interaction est en file d'attente. Le temps restant à l'exécution de cette interaction est inférieur à un temps prédéfini (fixé par le compositeur, ou par le type d'interaction).
Scénario nominal :

1. Le nom d'interaction s'affiche sur l'écran dès l'instant t2, avec un descriptif succinct et un compte à rebours indiquant les secondes restantes avant que l'interprète doive agir.
2. Cet affichage continue jusqu'à t3, temps où l'utilisateur doit agir.

Scénario d'exception :

1. La connexion wi-fi s'interrompt brusquement
2. Le message suivant s'affiche "Problème de connexion ! Votre interaction va être reportée ou annulée définitivement. "
3. Lors du retour de connexion à l'instant $t + T(\textit{sans connexion})$, on reprend la partition comme si tout était normal à $t + T(\textit{sans connexion})$.

Post-condition : L'interprète a été notifié suffisamment à l'avance de l'interaction qui est attendue de lui. Il est donc prêt à l'exécuter (ou la passer). L'IHM relative à l'interaction attendue s'affiche.

- Cas 3 : Établir la connexion de l'application avec un logiciel i-score
  Résumé : L'utilisateur demande une connexion, qui doit être acceptée par i-score afin qu'il ajoute le dispositif dans son arborescence.
  Acteur humain : l'utilisateur
  Acteur logiciel : le séquenceur i-score
  Pré-condition : L'application a été installée, ainsi que tous les éventuels dispositifs supplémentaires nécessaires à la communication réseau (sockets...), sur un dispositif mobile. Le logiciel i-score a été installé sur un ordinateur et est en attente de connexion(s) de dispositifs mobiles. Le dispositif mobile et l'ordinateur sont connectés au même réseau.
  Scénario nominal :
  1. Le séquenceur i-score est ouvert aux demandes de connexion venant de dispositifs mobiles (qui seront les "device" : voir doc i-score).
  2. L'utilisateur appuie sur le bouton de connexion.
  3. La demande d'établissement de la connexion est envoyée au logiciel i-score.
  4. i-score accepte la connexion et le nom du device est affiché dans l'arborescence de i-score.

  Scénario d'exception :
  1. Après l'étape 2, pas de logiciel i-score trouvé.
  2. Message d'erreur et retour à l'écran d'accueil de l'application.

  Scénario d'exception :
  1. Après l'étape 3, i-score refuse la connexion. Sortie du cas d'utilisation : retour à l'écran d'accueil de l'application.

  Post-condition : Le séquenceur i-score a accepté la demande de connexion. Il peut désormais lancer une partition, à laquelle le dispositif connecté pourra participer.

- Cas 4 : Transmettre les caractéristiques du smartphone ou de la tablette
  Résumé : La liste des capteurs en état de fonctionnement du dispositif sur lequel est installée l'application est établie et envoyée à i-score.

  Acteur logiciel : le séquenceur i-score

  Pré-condition : L'application est installée sur le dispositif. L'application est lancée, l'utilisateur souhaite prendre part au jeu d'une partition.
  Scénario nominal :

  1. Automatiquement, au premier lancement de l'application, l'application récupère la liste des capteurs présents dans le dispositif mobile.
  2. Ensuite l'application teste, un par un, tous les capteurs utiles pour le séquenceur, présents dans une liste définie au préalable par les développeurs. Elle définit et sauvegarde ainsi en mémoire lesquels sont utilisables et lesquels sont hors d'usage ou inaccessibles. Ce test des capteurs se fait à la première ouverture de l'application et peut être réitéré par simple demande de l'utilisateur de l'application. Pour tester les capteurs, il est possible que l'application sollicite l'utilisateur (par exemple : lui demander d'effectuer un mouvement pour vérifier le fonctionnement d'un capteur donné).
  3. L'application transmet à i-score la liste des capteurs en état de fonctionnement suite à l'établissement de la connexion.
  4. i-score enregistre cette liste de capteurs comme caractéristiques du dispositif mobile.

  Scénario alternatif :

  1. Perte de connexion
  2. Renvoi des données au retour de la connexion, si la perte des données a été repérée.

  Post-condition : Le séquenceur connaît les caractéristiques des appareils. Il pourra par la suite s'en servir pour décider à qui envoyer les prochaines demandes d'interaction.

- Cas 5 : Transmettre en temps réel les données relatives à l'interaction en cours
  Résumé : L'application communique à i-score les choix effectués par l'utilisateur pendant l'interaction, sous forme de données provenant des capteurs et de l'IHM.
  Acteur logiciel : le séquenceur

  Pré-condition : Le séquenceur et l'application doivent être connectés, l'application a dû afficher la demande d'interaction au préalable, l'IHM est affichée et l'interaction est en cours. Le séquenceur est en attente pour recevoir des données de l'application.

  Scénario nominal :

  1. Le dispositif récupère les informations résultant de l'interaction en cours (modifier une jauge, choisir une couleur, secouer le téléphone,...) et les transmet en temps réel à l'application.
  2. Le séquenceur effectue les traitements programmés dans la partition en fonction des données reçues, pour finalement apporter les modifications voulues au spectacle.

  Scénario alternatif :

  1. L'utilisateur est censé interagir mais rien ne se passe.
  2. Sortie du cas d'utilisation après l'écoulement du temps qui lui est imparti et retour à l'état d'attente (écran d'accueil ou compte à rebours).

Scénario alternatif :

1. L'utilisateur clique sur le bouton de fin de l'interaction.
2. Le dispositif arrête de récupérer les informations des capteurs. Fin de l'interaction en cours.
3. Retour à l'état initial d'attente.

Post-condition : Le séquenceur a reçu les informations nécessaires des effets de l'interaction . Il les a traités convenablement pour mener les changements requis dans le spectacle. (Par exemple : changer la couleur des projecteurs, baisser le volume sonore,. . .)

- Cas 6 : Transmettre les demandes d'interaction
  Résumé : Le séquenceur choisit le dispositif auquel la prochaine interaction doit être envoyée et l'envoie ou l'annule si aucun dispositif n'est prêt. Les applications mobiles concernées se préparent ensuite à afficher l'interaction.
  Acteur logiciel : le séquenceur i-score.
  Pré-condition : La connexion entre le logiciel et l'application est établie. Une partition est lancée. Une interaction est prévue à un instant t à venir et le séquenceur connaît les capteurs en état de marche de chaque dispositif.
  Scénario nominal :
  1. Le séquenceur détecte qu'il est temps de transmettre la prochaine demande d'interaction.
  2. Le séquenceur décide de l'appareil auquel demander l'interaction.
  3. Le séquenceur envoie la demande d'interaction à l'appareil choisi.
  4. L'application est notifiée d'une demande d'interaction .
  5. Elle mémorise les données relatives à cette interaction (effets, actions à effectuer et/ou liste des objets graphiques, durée, temps restant avant le début)
  6. Elle lance une temporisation, pour savoir à quel moment il faut basculer dans le cas 2 et donc prévenir l'utilisateur de l'interaction à venir.

  Scénario alternatif :
  1. Il y a plus d'interactions simultanées que d'appareils connectés. On ignore les interactions en trop.

  Scénario d'exception :
  1. Le temps restant pour l'interaction est inférieur au délai d'information de l'interprète.
  2. Sortie du cas d'utilisation : informer l'utilisateur de l'interaction à venir directement.

  Post-condition : L'application est informée de la prochaine interaction. On peut désormais afficher le compte à rebours au bon moment.

2.2.3 Détail des cas d'utilisation pour le système : logiciel i-score

— Cas 7 : Manipuler les nouveaux types d'interactions
Résumé : Le compositeur ajoute, modifie et supprime les nouveaux types d'interactions dans les partitions.
Acteur humain : le compositeur
Pré-condition : Le compositeur édite une partition.
Scénario nominal :

1. Le compositeur ajoute une interaction dans une partition.
2. Il indique combien de téléphones peuvent participer à cette interaction et les éventuelles conditions (nombre minimum, maximum ou exact de téléphones connectés pour que cette interaction soit prise en compte).
3. Il peut ensuite modifier l'interaction ajoutée.

À noter que ce scénario existe déjà de base dans i-score, mais on pourra le répéter avec de nouveaux types d'interactions

Post-condition : La partition comporte de nouvelles interactions.

3 Diagramme de séquence

Pour expliciter le modèle de cas d'utilisation, on retrouve le diagramme de séquence permettant de montrer le déroulement d'une utilisation normale de l'application.

D'après le diagramme de la figure 2, l'exécution de notre application comporte deux grandes phases : connexion avec i-score et réalisation d'interactions. À noter que la partition doit avoir été écrite par le compositeur sur i-score au préalable.

Au premier lancement de l'application, des tests sur les capteurs du dispositif mobile de l'application (téléphone ou tablette) sont lancés et la liste des capteurs en état de fonctionnement est sauvegardée dans l'application. L'interprète qui a lancé l'application fait ensuite une demande de connexion avec le séquenceur i-score, pendant laquelle l'application envoie également ses caractéristiques.

Après que la connexion ait été établie, l'application reste en état d'attente jusqu'à ce que le compositeur joue une partition. Dès lors, i-score peut envoyer la prochaine interaction à l'application, qui affiche par la suite un compte à rebours permettant à l'interprète de se préparer à exécuter l'interaction demandée. Après quelques secondes, l'IHM de l'interaction est affichée. L'interprète peut donc l'effectuer. Comme indiqué dans le diagramme, l'application transmet à i-score le flux de données résultant de cette interaction.

Ensuite, le séquenceur peut envoyer la prochaine demande interaction pour qu'elle soit traitée par l'application, effectuée par l'interprète et ainsi de suite ...

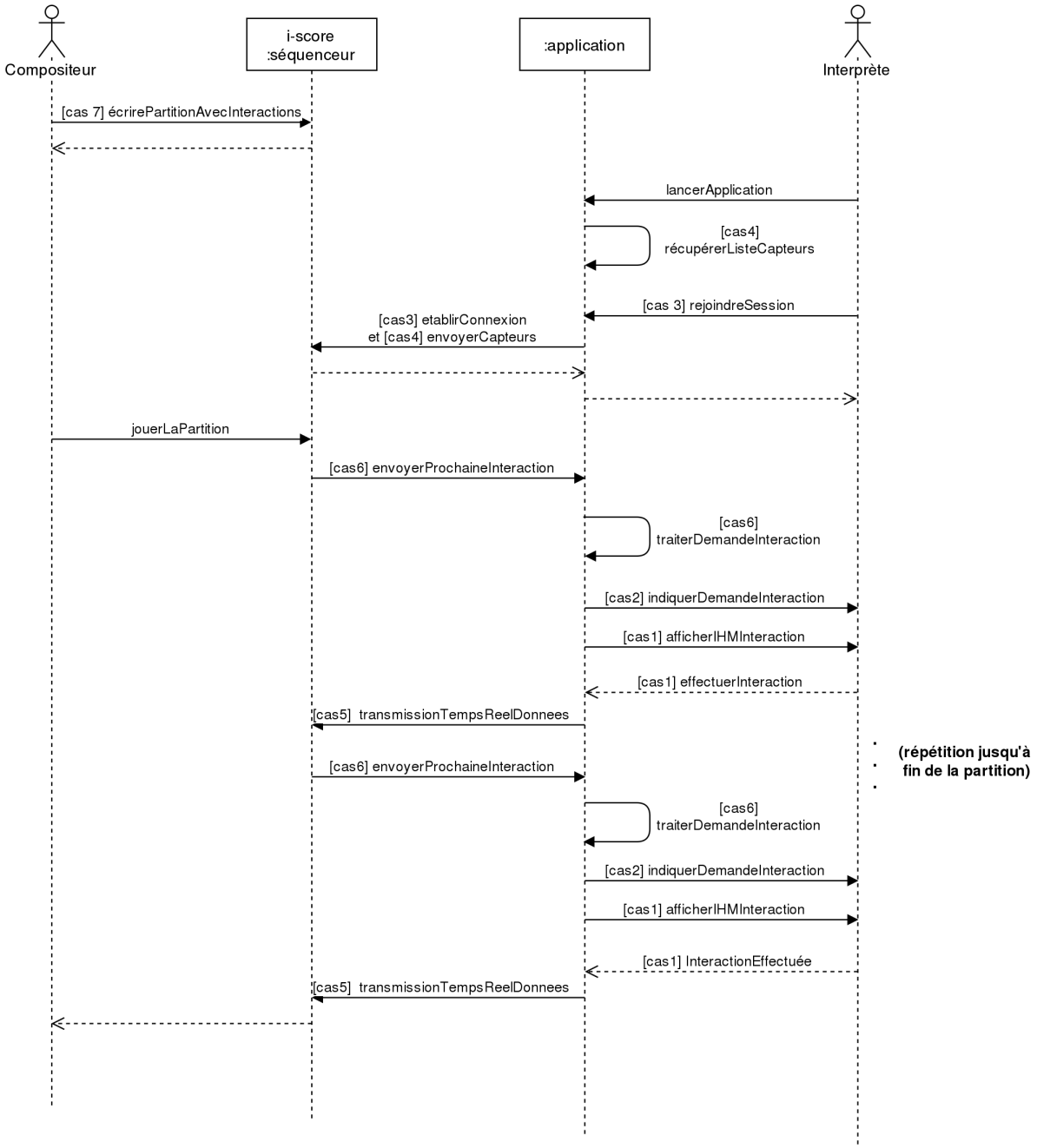

FIGURE 2 – Diagramme de séquence.

4 Remarques

- En ce qui concerne la transmission des données relatives aux **interactions à venir**, nous allons dans un premier temps nous limiter à demander l'exécution d'une interaction sans annonce préalable à l'interprète. On n'a alors pas de compte à rebours, les temps $t2$ et $t3$ définis ci-dessous sont confondus.
  Ensuite, nous tâcherons de développer une version capable de prévenir l'utilisateur, et utilisant des heuristiques lorsque c'est nécessaire : l'utilisateur est informé de l'interaction à venir soit par un compte à rebours et une description de l'interaction demandée, soit par un écran d'attente signifiant l'arrivée d'une interaction sans préciser un décompte de temps et/ou sans préciser le type d'interaction. En effet, i-score permet de créer des partitions utilisant, entre autres, des boucles, des instructions conditionnelles et des probabilités. Ces fonctionnalités empêchent parfois de prévoir quelles interactions sont à venir, d'où l'utilisation d'heuristiques. À noter que cette version du projet reste assez ambitieuse, tant les fonctionnalités d'i-score sont diverses.

  En ce qui concerne la version avancée de l'application, nous avons identifié 4 dates définies comme suit :

  - **t1** = date à laquelle le séquenceur envoie l'interaction à l'application
  - **t2** = date de début du compte à rebours
  - **t3** = date de début de l'interaction
  - **t4** = date de fin de l'interaction

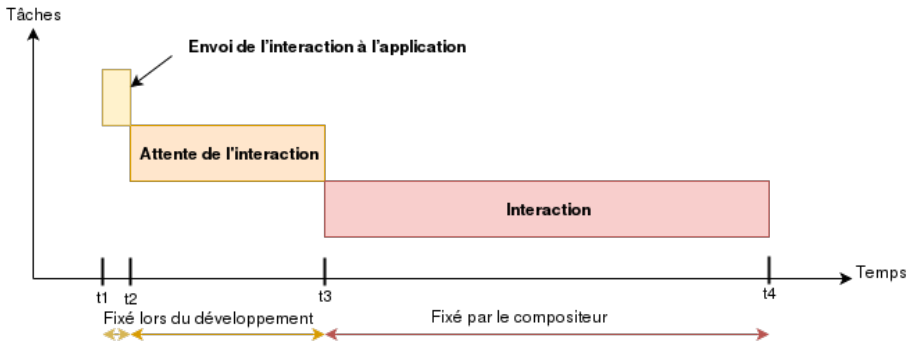

FIGURE 3 – Diagramme de Gantt décrivant la séquence des évènements.

Dans ce modèle final, l'application reçoit à l'avance l'interaction à effectuer et sait également dans combien de temps elle sera exécutée sur la partition du séquenceur. De cette manière, l'application pourra mieux préparer l'utilisateur à l'interaction. On précise, via le diagramme ci-dessus, que les intervalles Δt12 = t2-t1, ainsi que Δt23 = t3-t2, sont fixés lors du développement. Seul Δt34 = t4 - t3 est fixé par le compositeur lors de l'édition des interactions d'une partition.

- En ce qui concerne l'**attribution des interactions** aux différents dispositifs mobiles, plusieurs cas sont à considérer. Dans un premier temps, nous considérerons que l'on a un nombre limité de dispositifs mobiles à pouvoir interagir lors de la lecture d'une partition. La partition i-score gère la répartition entre eux, en utilisant une logique explicite à l'allure suivante : *"à t = 40 secondes, si 2 dispositifs sont connectés, alors deviceA fait interaction 1 et deviceB fait interaction 2, sinon si 3 dispositifs sont connectés, alors tous les téléphones font l'interaction 4, sinon etc"*.
  Après avoir implémenté cette méthode d'attribution des interactions, nous pourrons éventuellement réfléchir à d'autres stratégies de répartition, comme une attribution aléatoire des interactions entre les dispositifs, ou encore l'utilisation d'une file d'attente. Nous évoquerons également dans la partie V la possibilité de permettre la participation d'un nombre variable de dispositifs mobiles à chaque interaction.

Troisième partie
Besoins non fonctionnels

1 Ergonomie et interface

Les différentes vues et interfaces proposées par l'application doivent être intuitives à l'utilisation tout en arborant une certaine esthétique. De plus, il faut associer à chaque type d'interaction une ou plusieurs interfaces adaptées suivant l'outil désiré (sliders, palettes, boutons, etc).

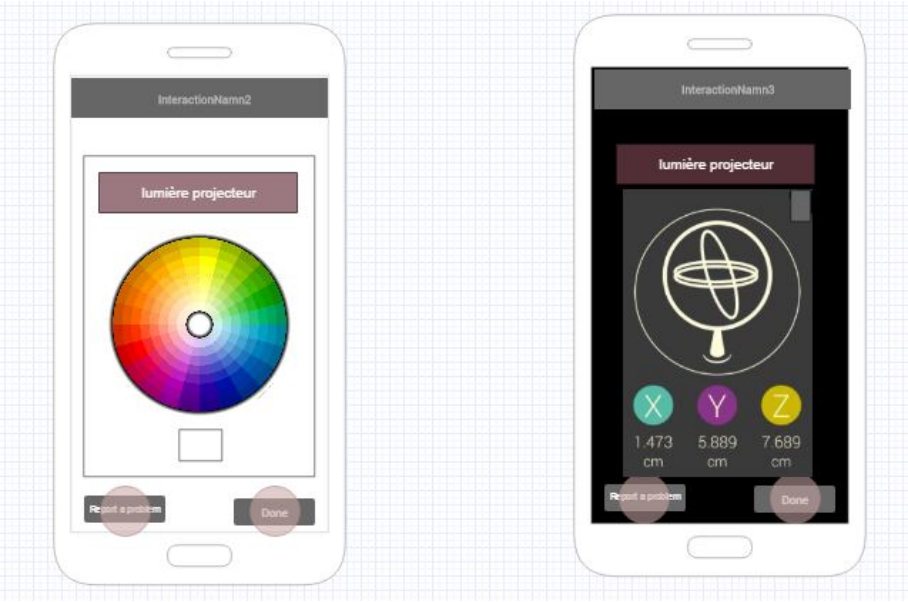

FIGURE 4 – Exemples d'interfaces pour les deux types d'interactions (graphique ou via capteurs)

Un certain degré d'attention sera porté sur l'ergonomie. Les interfaces doivent être pratiques et simples à utiliser. Différents types d'interfaces seront implémentés suivant le type d'interaction (sliders, palettes, boutons, etc). Comme spécifié dans le détail des cas d'utilisation, lors du compte à rebours précédant une interaction et pendant le temps imparti à l'interaction, l'utilisateur aura accès à une description de l'interaction attendue.

Comme expliqué plus haut, il existe différents types d'interactions, auxquels seront associés différentes interfaces et différentes modifications du spectacle. Les interactions proposées seront les suivantes :

— Action de l'utilisateur sur l'écran du dispositif sur lequel est installée l'application, directement sur l'IHM :
  - Sliders
  - Jauge
  - Simple zone 2D où la position du doigt définit un couple de paramètres (x, y).
  - Palette de couleurs
  - Launchpad
  - Boutons divers
  - Combinaisons, prédéfinies dans un premier temps par l'équipe de développement, de plusieurs interactions simultanées à effectuer depuis le même dispositif (par exemple, une palette de couleurs permettant de modifier la couleur d'un projecteur, ainsi qu'un curseur permettant de varier l'intensité de sa lumière). Voir partie V pour les combinaisons d'interactions créées par le compositeur.
  - Autres objets graphiques, à prévoir pour une éventuelle évolution, implémentée si le délai restant est suffisant (voir partie V).
— Mouvement du mobile/tablette, détecté par des capteurs (autres que l'écran) :
  - Translations longitudinales
  - Translations transversales
  - Translations verticales
  - Lacets
  - Roulis
  - Tangage

Afin de permettre les interactions du second type, le dispositif mobile sur lequel est installée l'application doit posséder un certain panel de capteurs en état de fonctionnement :

- Gyromètre
- Accéléromètre
- Magnétomètre

Les maquettes suivantes décrivent l'allure globale de l'application et montrent le type d'IHM que l'utilisateur aura à sa disposition.

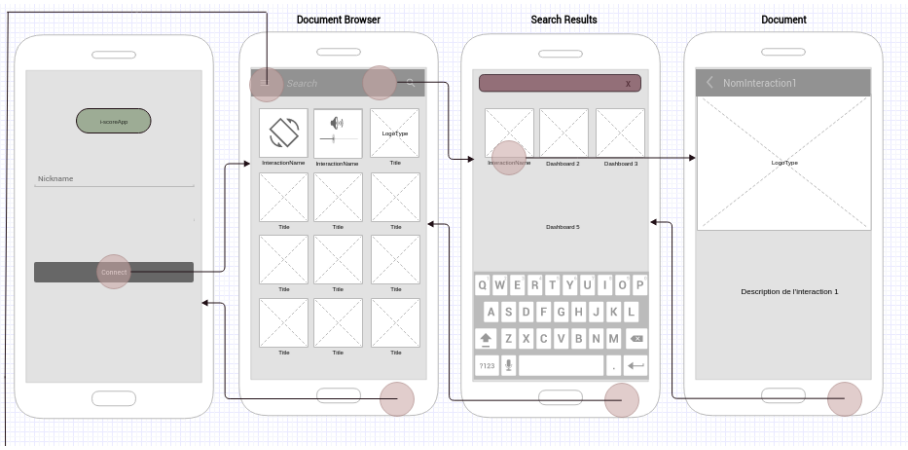

FIGURE 5 – Page de lancement de l'application et pages de description des interactions possibles.

FIGURE 6 – Page de déconnexion utilisateur et menu.

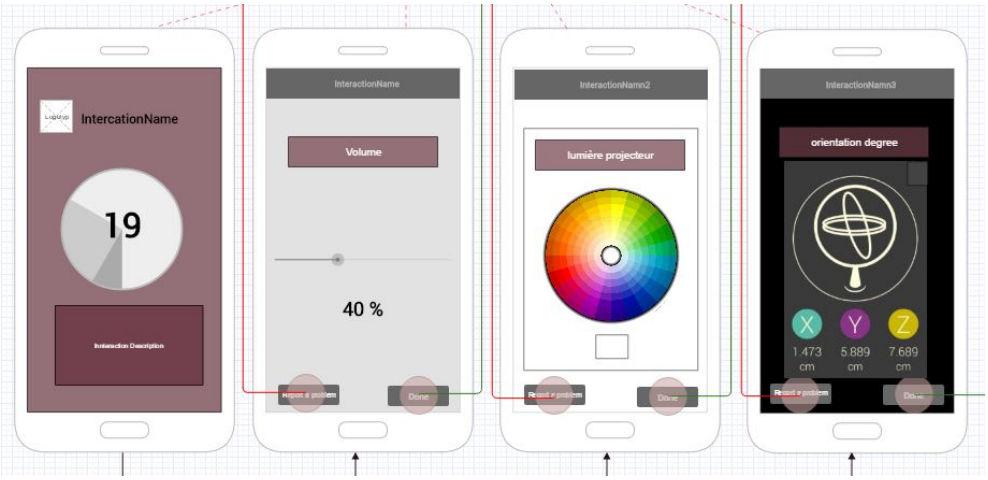

FIGURE 7 – Arrivée d'une interaction : page du compte à rebours avec description de l'interaction, exemples d'IHM d'interactions (slider, palette de couleurs puis interaction liée au tangage et au roulis du dispositif mobile).

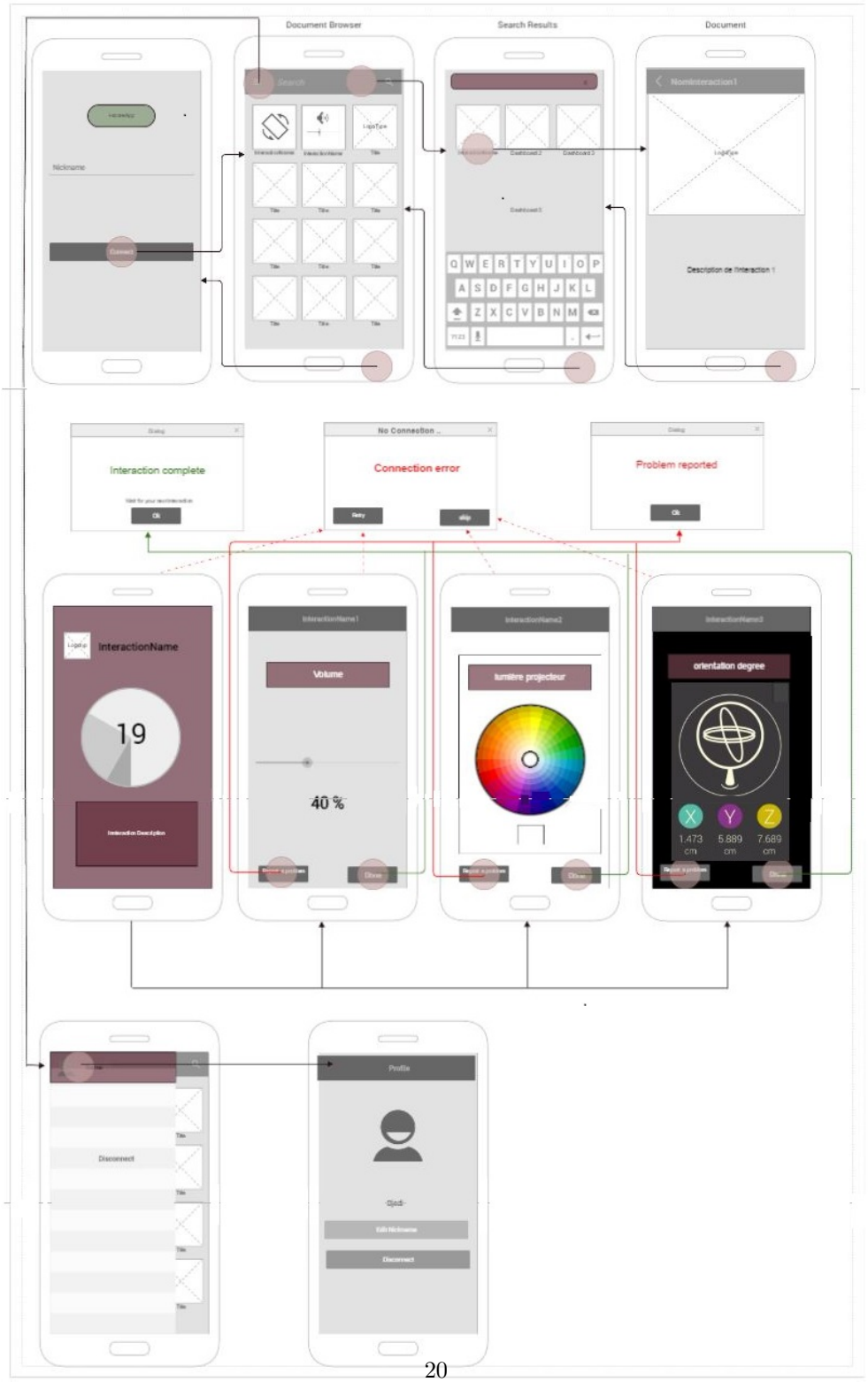

FIGURE 8 – Rendu global.

2 Compatibilité

Un seul code sera implémenté pour cette application qui est donc considérée comme hybride. Grâce à l'environnement de développement Qt Creator, le code pourra être porté sur les deux plateformes mobiles Android et iOS.

3 Réponse temps réel

L'application devra être connectée en temps réel au logiciel i-score afin de lui transmettre les données relatives aux interactions effectuées par l'utilisateur.

4 Connectivité

L'application a besoin d'être connectée dans un réseau local pour fonctionner correctement.
En ce qui concerne l'établissement de la connexion entre le logiciel i-score et l'application, notre client nous a proposé d'utiliser le protocole Zeroconf grâce à la bibliothèque KDNSSD qui l'implémente en C++ et qui est aussi utilisée dans le code d'i-score. Ce protocole offre la possibilité de rechercher des services et de s'y connecter, ce qui correspond tout à fait à ce dont nous avons besoin. Nous pourrons ainsi rechercher les appareils disposant d'un logiciel i-score ouvert sur le réseau local et nous y connecter.

La connexion entre i-score et notre application sera gérée grâce à la bibliothèque OSSIA fournie par le client. Celle-ci permet une gestion plus haut niveau de la connexion que si l'on utilisait directement des sockets et permet d'utiliser un protocole de type TCP ou UDP.
Aujourd'hui, cette bibliothèque permet l'utilisation d'OSC qui passe par le protocole UDP, donc non connecté. Cela permet à l'application de continuer d'envoyer des données sans problème après avoir perdu puis retrouvé sa connexion au réseau local.
Cependant, cette connexion UDP peut ne pas être très fiable et induire des pertes de paquets, pouvant causer des problèmes lors de l'utilisation de notre application qui doit communiquer des données en temps réel.
Nous utilisons tout de même cette bibliothèque, car la perte ponctuelle de quelques paquets serait la plupart du temps imperceptible pour l'utilisateur. Notre rôle n'est pas de chercher un moyen de rendre cette connexion plus fiable, en utilisant un autre protocole. En revanche, l'utilisation d'OSSIA permet aux développeurs de cette bibliothèque d'en modifier l'implémentation, notamment afin d'améliorer la fiabilité lors d'échanges de données, sans influer sur notre implémentation de l'application.

Quatrième partie

Architecture et conception

1 Outils et langages

Le souhait du client est que l'application soit déployable sur iOS et Android. Pour réaliser cela, il a proposé de travailler avec le framework QtQuick qui est composé d'un langage appelé QML, adapté à ce qui concerne l'interface et le design, d'un module C++ QtDeclarative qui intègre le QML avec les objets C++, ainsi que de l'IDE QtCreator. De plus, il offre la possibilité de développer en multi-plateformes sans avoir à réécrire de code, avec une simple recompilation.

2 Le code à ajouter à i-score

2.1 L'architecture d'i-score

Le logiciel i-score est écrit en C++ et est basé sur le design pattern *abstract factory* qui, grâce à l'utilisation d'interfaces, permet de cacher l'implémentation des classes et d'éviter au code client de savoir de quelles classes sont les objets qu'il souhaite instancier. De cette manière, il est facile de rajouter de nouvelles fonctionnalités sans modifier le code existant puisqu'il suffit d'étendre les classes déjà présentes dans i-score. Le code de i-score est donc organisé en un certain nombre de plugins qui viennent se greffer sur l'architecture de base. Il implémente ainsi les process, qui sont les actions possibles avec les périphériques connectés au logiciel.

2.2 L'ajout des interactions

Pour que le compositeur utilisant i-score puisse insérer des interactions utilisant l'application mobile dans sa partition, il faudra écrire un plugin qui ajoute un nouveau type de process : les interactions avec l'application mobile. Il contiendra une classe `interaction` qui héritera de la classe `process`. De plus, ce plugin sera en charge de la communication avec les dispositifs utilisant l'application et connectés à i-score via l'API OSSIA.

3 Découpage en modules

Dans le but d'avoir un meilleur point de vue sur les implémentations à effectuer dans ce projet, nous pouvons diviser sa conception en plusieurs modules dont les principaux axes seront :

- Les ajouts dans i-score
- L'application
- La connexion i-score - application

On retrouvera le détail sur le schéma suivant :

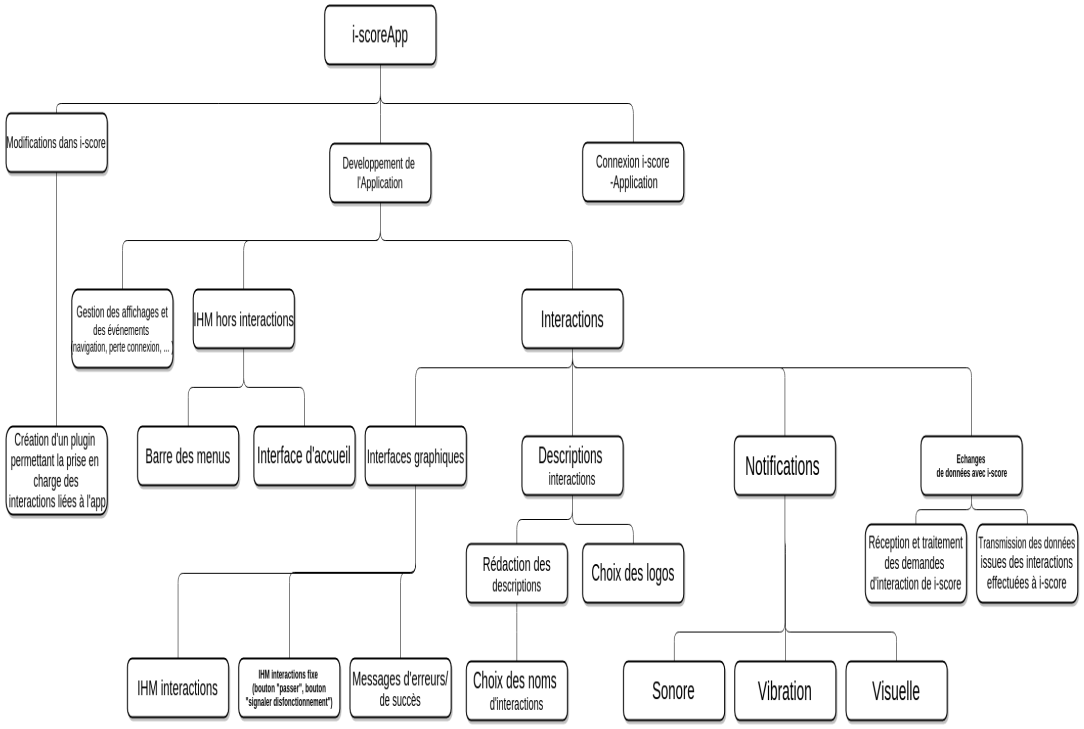

FIGURE 9 – Découpage en modules (WBS ou OT Organigramme des Tâches)

Cinquième partie
Évolution du système

Les possibilités d'interactions sont quasiment infinies, nous avons donc, avec l'aide de notre client Jean-Michaël Celerier, fait un choix quant aux interactions que nous souhaitons traiter dans un premier temps. Nous travaillons dans l'objectif de produire une application mobile/tablette, mais il serait possible d'étendre ce projet aux cartes Arduino par exemple et ainsi décupler les possibilités offertes aux compositeurs et interprètes. Plusieurs évolutions du système sont donc possibles, au niveau du déroulement du spectacle comme au niveau des interactions.

Au niveau du déroulement du spectacle, il serait très intéressant de ne plus limiter le nombre de dispositifs mobiles pouvant effectuer chaque interaction. On gérerait ainsi le cas d'une participation ouverte du public où chacun peut se connecter et se déconnecter quand il le souhaite. Dans ce cas, on attribue les interactions par groupe de personnes simplement en fonction du nombre d'utilisateurs connectés. Le compositeur pourrait alors spécifier les interactions de la façon suivante :

- De $t = 0$ à $t = 10$, tous les utilisateurs font l'interaction 1.
- De $t = 10$ à $t = 20$, répartir les interactions 2, 3 et 4 entre les participants.
- De $t = 15$ à $t = 30$, un participant fait l'interaction 5.
- De $t = 30$ à $t = 40$, pas d'interaction.

En ce qui concerne les interactions, il serait intéressant de laisser au compositeur la possibilité de créer sa propre combinaison d'interactions, qui seront effectuées simultanément par le même utilisateur : il pourra choisir de faire apparaître simultanément, sur l'écran de l'application, plusieurs interfaces d'interactions, comme un slider et une palette de couleurs.

Nous pourrons également implémenter de nouvelles interactions plus élaborées :

- Une palette de couleurs en 2d carrée, avec une bille (curseur) qui se déplace sur la palette. L'utilisateur peut lancer la bille dans n'importe quelle direction pour qu'elle rebondisse sur les bords de la palette. La couleur envoyée est celle sur laquelle se trouve la bille.
- Un cercle d'enceintes, représenté sur l'écran avec un curseur, pour simuler une spatialisation du son : plus le curseur est proche d'une enceinte, plus le volume de cette enceinte est élevé par exemple.
- ...

Sixième partie
Plans de tests et validation

1 Tests unitaires

Une batterie de tests unitaires sera développée en parallèle de l'application afin de tester l'intégralité du code. Sur conseil de notre client, nous pourrons notamment utiliser des outils tels que

`Travis CI` pour tester le code à chaque `push` sur le dépôt, ou encore `coveralls.io` pour tester la couverture du code.

2 Tests systèmes

Les plans de tests permettant de vérifier le fonctionnement correct de notre application sont les suivants.

Remarque : Pour chaque test, nous supposons que les éventuelles préconditions ont été testées et fonctionnent.

- Test cas 1 : Effectuer l'interaction spécifiée par la partition

  1. Créer une partition avec plusieurs interactions successives :
     (a) Un bouton permettant de lancer un fichier audio a.mp3. Prévoir un temps de 10 secondes pour cette interaction ;
     (b) Une jauge permettant de régler le volume en secouant le téléphone vers le haut ou le bas (volume de base : 50%). Prévoir un temps très long pour cette interaction (par ex. 10 minutes) ;
     (c) Un launchpad avec 3 touches, correspondant aux fichiers son b.mp3, c.mp3 et d.mp3. Prévoir un temps court (par ex. 10 secondes) ;
     (d) Une palette de couleurs permettant de régler la couleur d'un projecteur et en même temps une jauge permettant de régler l'intensité de la lumière émise par le projecteur (réglage de base : projecteur blanc) ;
     (e) Une interface de spatialisation du son *(réglage de base : point au milieu, son identique sur toutes les enceintes).
  2. Exécuter la partition. Lors de l'arrivée de la première interaction, appuyer sur le bouton : vérifier le lancement du son a.mp3. Ensuite, secouer le téléphone : vérifier qu'il n'y a pas de variation d'intensité du volume, ou de tout autre paramètre.
  3. Au bout de 10 secondes, vérifier la disparition de l'IHM et son remplacement par un écran d'attente. Le son doit en revanche continuer.
  4. Lors de l'arrivée de la seconde interaction, régler la jauge de volume en déplaçant le curseur sur l'IHM de haut en bas et vérifier l'augmentation ou la diminution du volume.
  5. Secouer le téléphone de haut en bas, y compris avec des mouvements non strictement rectilignes et vérifier l'augmentation ou la diminution du volume.
  6. Fixer le volume au minimum et essayer de le diminuer encore sur l'IHM, puis en secouant le téléphone vers le bas : vérifier qu'il n'y a pas de changements ni d'erreur (par ex. augmentation du volume, plantage de l'application...)
  7. Remettre le volume à 50%, puis augmenter le volume depuis l'IHM tout en secouant le téléphone vers le bas : vérifier que la première action effectuée est bien celle prise en compte.

*Uniquement si celle-ci a été implémentée : voir partie *Planning prévisionnel*.

8. Interrompre la connexion wi-fi et la rétablir aussitôt : vérifier l'apparition d'un message "Problème de connexion ! Votre interaction va être reportée ou annulée définitivement", lors de l'interruption de la connexion, puis la disparition de ce message lors du rétablissement de la connexion. Effectuer des actions sur l'IHM puis le dispositif pour modifier le volume : vérifier la prise en compte de ces actions.
9. Appuyer sur le bouton permettant de terminer l'interaction : vérifier l'apparition de l'écran d'attente. Le mouvement du téléphone vers le haut ou le bas ne doit plus permettre de modifier le volume.
10. Lors de l'arrivée de la troisième interaction, appuyer sur les différentes touches du launchpad et vérifier le lancement du son correspondant.
11. Interrompre la connexion wi-fi et attendre la fin des 10 secondes : vérifier l'apparition de l'écran d'attente.
12. Pour la quatrième interaction, vérifier que les modifications de l'intensité et de la couleur du projecteur s'effectuent normalement.
13. Tester la spatialisation du son si elle a été implémentée.

- Test cas 2 : Informer l'utilisateur de la prochaine interaction
    1. Fixer Δt12 à 3 secondes et Δt23 à 5 secondes [†]
    2. Créer la même partition qu'à l'étape 1 du test cas 1. La première interaction doit commencer 10 secondes après le début de la partition et la deuxième interaction doit durer 10 secondes.
    3. Exécuter la partition. Vérifier qu'au bout de 5 secondes, le nom d'interaction s'affiche sur l'écran, avec un descriptif succinct et un compte à rebours indiquant les secondes restantes (initialement 5) avant que l'interprète doive agir. Vérifier ensuite qu'à l'arrivée à 0, le compte à rebours laisse place à l'IHM correspondant à l'interaction.
    4. Passer l'interaction en appuyant sur le bouton correspondant.
    5. Pendant le compte à rebours de l'interaction suivante, interrompre la connexion wi-fi. Vérifier l'affichage d'un message "Problème de connexion ! Votre interaction va être reportée ou annulée définitivement."
    6. Attendre 15 secondes, puis vérifier que la prochaine interaction à s'afficher est l'interaction 3.
    7. Pour chaque interaction, vérifier que le descriptif affiché pendant le compte à rebours corresponde à la (ou les) interaction(s) attendue(s), avec un titre et un rappel des méthodes offertes pour interagir.

- Test cas 3 : Établir une connexion et une synchronisation
    1. Connecter l'ordinateur où se trouve le séquenceur au wi-fi, mais pas le dispositif. Vérifier l'affichage d'un message d'erreur "Aucune connexion réseau" sur le dispositif.

[†] Cf. définition des Δt.

2. Connecter le dispositif au wi-fi. Vérifier l'ajout du dispositif dans l'arborescence i-score.
3. Mettre i-score en attente de connexions.
4. Demander une connexion de l'application vers i-score.
5. Transmettre un message du séquenceur vers le dispositif, puis du dispositif vers le séquenceur : vérifier la réception des messages.
6. Sans interrompre la connexion du premier dispositif, répéter les étapes 2 à 3 avec un autre dispositif mobile, puis effectuer les vérifications indiquées.

- Test cas 4 : Transmettre les caractéristiques du smartphone ou de la tablette
  1. Définir la liste des capteurs pouvant être présents sur un dispositif mobile et pouvant être utiles pour effectuer les interactions. Cette liste peut être factice et référencer des capteurs à priori inutiles (par ex. thermomètre)
  2. Choisir de préférence un dispositif mobile défectueux ou ne possédant pas l'un de ces capteurs et y implémenter cette liste. Lancer un test des capteurs présents sur le dispositif mobile. A l'aide d'une autre application (par ex. boussole implémentée en natif, ou Sensor Kinetics), vérifier que tous les capteurs indiqués comme en état de fonctionnement sont bien présents et fonctionnent, et que les capteurs indiqués comme absents ne fonctionnent pas. Dans l'arborescence du séquenceur, le nom attribué au dispositif mobile devrait s'afficher, ainsi que la liste des capteurs qui fonctionnent. Vérifier que toutes ces données sont correctement affichées.
  3. Interrompre la connexion wi-fi du dispositif, puis lancer un nouveau test des capteurs. Rétablir enfin la connexion. Vérifier que le séquenceur affiche les mêmes données qu'à l'étape 2.

- Test cas 5 : Transmission temps réel des données relatives à l'interaction en cours
  1. Créer la même partition qu'à l'étape 1 du test cas 1.
  2. Exécuter la partition. Les interactions a, b et e [‡]seront effectuées normalement par utilisation de l'IHM, ou action sur le téléphone quand c'est possible ; l'interaction c sera passée en attendant la fin du temps imparti et l'interaction d sera passée en utilisant le bouton prévu à cet effet.
  3. À chaque interaction, vérifier que les informations reçues par le séquenceur sont correctes.

- Test cas 6 : Transmettre les demandes d'interaction sur l'application
  Première partie :
  1. Fixer Δt12 à 5 secondes et Δt23 à 5 secondes [§]

[‡]Si la spatialisation du son a été implémentée.
[§]Cf. définition des Δt.

2. Créer une partition contenant deux interactions. La première commence 15 secondes après le début de la partition et la deuxième commence 3 secondes après la première.
3. Entre 5 et 10 secondes après le lancement de la partition, vérifier que l'application a reçu la demande d'interaction et qu'elle a mémorisé les données relatives à cette interaction (effets, actions à effectuer et/ou liste des objets graphiques, durée, temps restant avant le début)
4. Cinq secondes après la réception de la demande d'interaction, vérifier l'affichage du compte à rebours.
5. Immédiatement à la fin de la première interaction, vérifier qu'un compte à rebours démarrant à 3 secondes s'affiche pour l'interaction suivante.

Deuxième partie :

1. Créer la même partition qu'à l'étape 1 du test cas 1.[¶] Ajouter trois interactions f, g et h identiques à la d. Les interactions d et e doivent se chevaucher, ainsi que les interactions g et h (mais e commence après d, et h après g).
2. Fixer le nombre de dispositifs devant interagir pour chacune des interactions suivantes, désignées par leur lettre :
   (a) Un seul dispositif ;
   (b) Trois dispositifs ;
   (c) Deux dispositifs ;
   (d) Au maximum deux dispositifs ;
   (e) Au minimum deux dispositifs ;
   (f) Quatre dispositifs ;
   (g) Trois dispositifs ;
   (h) Trois dispositifs.
3. Connecter 3 dispositifs à i-score. L'un de ces dispositifs doit avoir au moins l'un de ses accéléromètres manquants ou défaillants, de telle manière qu'un mouvement haut-bas ne puisse pas être pris en compte.
4. Pour les interactions a à f, vérifier que le bon nombre de dispositifs a reçu la demande d'interaction selon le tableau suivant :

[¶] Si la spatialisation du son n'a pas été implémentée, remplacer l'interaction e par une interaction identique à la d.

Interaction	*Nombre de téléphones devant recevoir l'interaction*	*Remarques*
a	1	
b	2	Le dispositif n'ayant pas reçu la demande doit être celui dont le capteur est défectueux
c	2	
d	2	
e	1	Le dispositif qui reçoit la demande doit être celui qui ne l'a pas reçue à l'interaction d
f	3	
g	3	

5. Au moment de l'interaction `h`, vérifier que l'interaction est annulée et qu'elle n'est reçue par aucun dispositif. Ensuite, interrompre la connexion wi-fi de l'un des dispositifs effectuant l'interaction `g` : vérifier que cela n'a aucun impact sur les autres dispositifs et que le séquenceur continue à fonctionner normalement.

- Test cas 7 : Manipuler les nouveaux types d'interactions
    1. Créer une partition avec une seule interaction : une jauge permettant de régler le volume en secouant le téléphone vers le haut ou le bas, d'une durée de 20 secondes et avec la participation possible d'un seul dispositif mobile. Vérifier l'affichage de cette nouvelle interaction sur i-score.
    2. Modifier l'interaction pour qu'au moins deux téléphones puissent l'effectuer et qu'elle dure non plus 20 mais 10 secondes. Vérifier l'affichage des modifications sur i-score. Si le cas 1 a déjà été testé, vérifier que l'interaction demandée correspond bien aux nouveaux paramètres.
    3. Supprimer l'interaction : vérifier la disparition de cette nouvelle interaction sur i-score.
    4. Répéter les cas 1 à 3, mais en ajoutant l'interaction sur une partition existante au lieu d'une partition vierge.

3 Tests de recettes

Des tests grandeur nature pourront également être menés, en accord avec notre client, toujours afin de s'assurer du bon fonctionnement de l'application dans des conditions réelles. Un spectacle pourra être mis en place avec plusieurs utilisateurs possédant chacun un mobile ou une tablette pourvu de l'application et une partition rédigée par un compositeur. Cela permettra de simuler des utilisations courantes et "extrêmes" de l'application pour s'assurer du bon déroulement du spectacle dans le maximum de scénarios possibles.

Nous allons notamment tester notre application dans des situations types, qui nous ont été suggérées par le client :

1. **Régie de spectacle** : Les partitions sont globalement fixées à l'avance, mais ponctuellement, la personne qui s'occupe de la régie veut pouvoir modifier les paramètres du spectacle. Par exemple, elle veut modifier la couleur d'un projecteur uniquement de 1min30 à 1min50 après le début du spectacle : dans ce cas, l'interaction écrite est unique et un seul téléphone peut effectuer cette interaction. Si d'autres téléphones cherchent à se connecter à i-score, la connexion est refusée ;
2. **Participation limitée du public** : on a un nombre fixe de téléphones pour chaque interaction et le séquenceur gère la répartition entre ces téléphones en utilisant une logique explicite comme ceci : *"Si 3 téléphones sont connectés, alors ils font l'interaction 1 ; si 4 téléphones sont connectés, alors ils font l'interaction 2, etc"*. Dans ce cas, le compositeur écrit plusieurs interactions qui peuvent se chevaucher, avec un nombre de téléphones pouvant participer pouvant varier d'une interaction à l'autre.

Septième partie
Planning prévisionnel

Sur conseil de notre client, nous allons dans un premier temps implémenter une version basique, mais opérationnelle, de l'application. Plus précisément, cette version n'implémentera qu'un nombre réduit de possibilités d'interactions, n'informera pas l'utilisateur de l'interaction à venir avant de lui demander de l'effectuer, et ne testera pas si les capteurs du dispositif mobile sont en état de marche ou non. Nous procéderons ensuite de façon itérative, afin d'ajouter des fonctionnalités et obtenir progressivement une version plus complète de l'application et du plugin.

Ensuite, si le délai restant pour le projet est suffisant, nous pourrons ajouter des fonctionnalités à l'application et au plugin. Celles-ci sont expliquées en partie V.

Le diagramme de Gantt ci-dessous présente nos prévisions en ce qui concerne l'implémentation de l'application et du plugin de i-score.

	Nom de la tâche	Durée
	Exécution complète du projet	5 mois
1	Ajouts dans i-score	1 mois
1.1	Création d'un plugin permettant la prise en charge des interactions liées à l'application	1 mois
2	Développement de l'application	17 semaines
2.1	Création des affichages et des évènements	2 semaines
2.2	IHM hors connexion	2 semains
2.2.1	Barre des menus	1 semaine
2.2.2	Interface d'acceuil	1 semaine
2.3	Interactions	12 semaines
2.3.1	Interfaces graphiques	4 semaines
2.3.1.1	IHM interactions	3 semaines
2.3.1.2	IHM interactions fixes(bouton "signaler dysfonctionnement"	1 semaine
2.3.1.3	Messages d'erreurs/ de succès	1 semaine
2.3.2	Description interactions	1 semaine
2.3.2.1	Redaction des noms d'intéractions (choix des noms)	4 jours
2.3.2.2	Choix des Logos	3 jours
2.3.3	Notifications	12 jours
2.3.3.1	Sonores	3 jours
2.3.3.2	Vibrations	3 jours
2.3.3.3	Visuelles	3 jours
2.3.4	Echanges de données avec i-score	4 semaines
2.3.4.1	Réception et traitement de demandes d'interaction de i-score	2 semaines
2.3.4.2	Transmission de données issues des interactions effectuées à i-score	2 semaines
3	Connexion i-score App	1 semaine

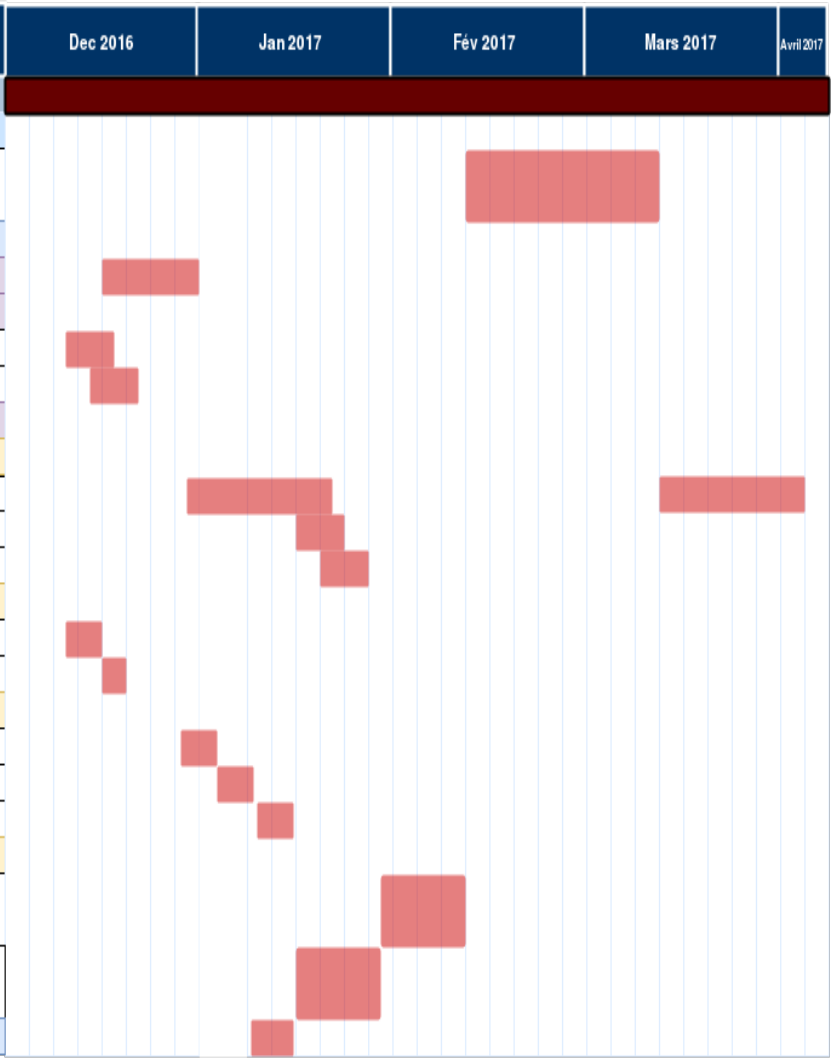

FIGURE 10 – Diagramme de Gantt du planning prévisionnel

Huitième partie
Lexique

— **Séquenceur** : Un séquenceur musical est un équipement permettant de faire jouer automatiquement un instrument de musique électronique ; il peut être de type matériel ou logiciel (logiciel dans notre cas)

— **Interaction** : Action menée par l'utilisateur dans le but de modifier le spectacle de manière sonore ou visuelle. Par exemple, secouer le téléphone est une interaction qui peut engendrer une modification du volume sonore, ou encore une modification de l'intensité des projecteurs

— **Interprète (ou Utilisateur)** : C'est notre acteur principal, c'est la personne qui agit sur l'application pour faire les interactions.

— **Partition** : Une partition, dans le cas de i-score, est un code traduisant le plan d'un spectacle musical. Elle sert à jouer avec les caractéristiques suivantes : la durée, l'intensité, la lumière, les sons, les effets, les interactions et leur placement dans le temps, etc.

— **Compositeur** : Le compositeur est la personne qui programme la partition sur le séquenceur.

— **Dispositif [mobile]** : Smartphone ou tablette sur lequel on a installé l'application mobile.

— **IHM** : Interface Homme Machine ; c'est l'ensemble des éléments graphiques affichés sur l'écran du dispositif mobile qui informent l'utilisateur ou sur lesquels il peut agir.

— **Launchpad** : Interface graphique constituée d'un clavier contenant de nombreuses touches colorées. Chaque touche lance un son pré-enregistré lorsque l'on appuie dessus.

— **Slider** : Curseur à déplacer sur une dimension (soit vertical, soit horizontal).

— **Automation** : En informatique musicale, dans un séquenceur, l'automation consiste à programmer des changements de réglages pendant la lecture d'un morceau, comme la variation de volume d'une piste audio. Cette mise en place peut se faire par mimétisme : le logiciel enregistre en temps réel des mouvements venant de l'utilisateur pour les reproduire lors des prochaines exécutions du morceau. Elle peut également se faire au moyen du tracé d'une courbe dans le séquenceur, représentant l'évolution du paramètre indiqué en fonction du temps (dans cet exemple, le volume en fonction du temps).

— **WBS ou OT Organigramme des Tâches** : Le Work Breakdown Structure est un arbre représentant la liste structurée de tous les travaux du projet. Les travaux sont à ce stade uniquement identifiés. Ce graphe, utilisant la notation de précédence, représente les relations entre les tâches du type « Fait partie de ». C'est une vision maîtrise d'œuvre des tâches afin de déterminer les niveaux de visibilité du projet.

Neuvième partie
Webographie

- Notion de "device" dans i-score : `http://i-score.org/communicating-in-osc-with-i-score/`
- Bibliothèque OSSIA (i-score, API, ...) : `https://github.com/OSSIA/`
- Bibliothèque KDNSSD (Zeroconf) : `https://github.com/KDE/kdnssd`

- Les différents mouvements détectés par les capteurs d'un dispositif mobile : `http://www.android-mt.com/news/mieux-comprendre-capteurs-34388` de test :
  - Travis CI : `https://travis-ci.org/`
  - Coveralls : `https://coveralls.io/`

10.3 Manuel d'utilisation et de maintenance

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 :

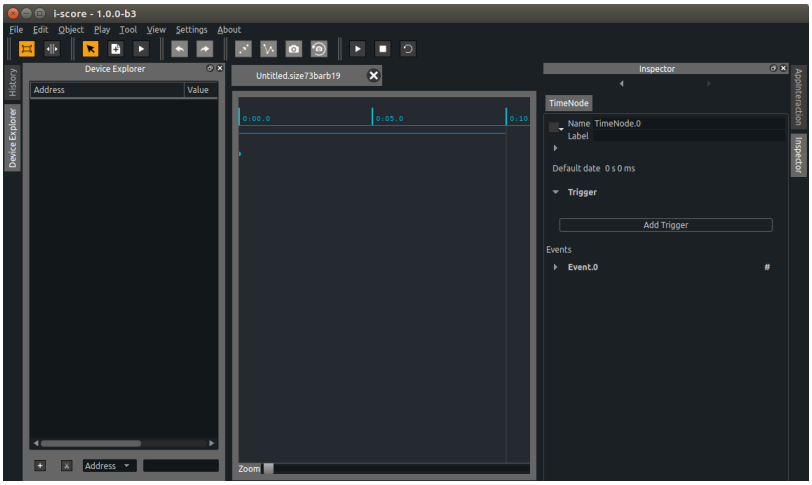

FIGURE 1 – Fenêtre principale

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.

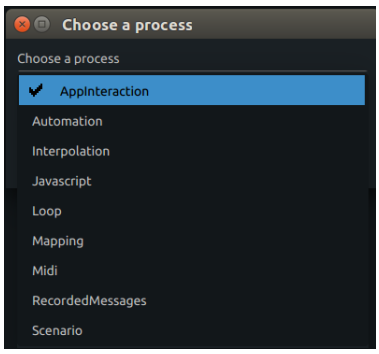

FIGURE 2 – Le menu pour faire le choix du `Process`

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

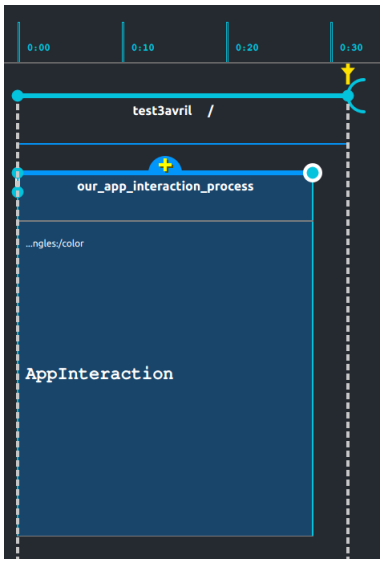

FIGURE 3 – Partition contenant uniquement un `Process AppInteraction`

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`.

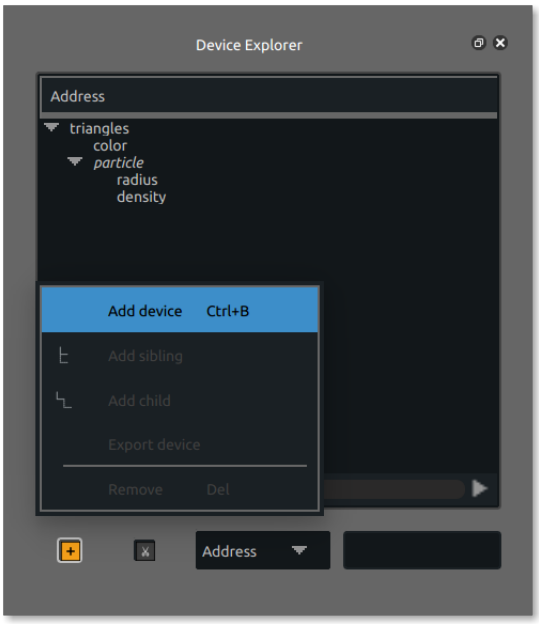

FIGURE 4 – Le `DeviceExplorer` de *i-score*

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

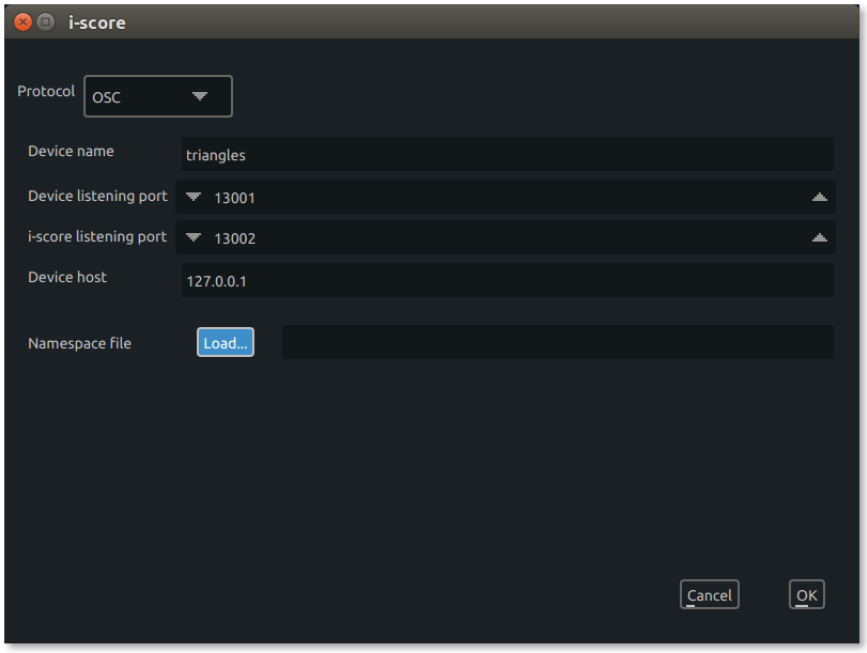

FIGURE 5 – Ajout d'un `OSCDevice`

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`.

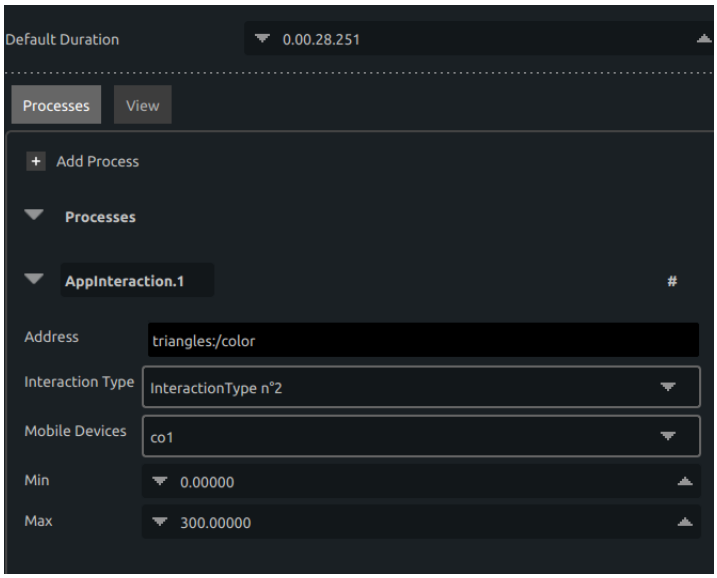

FIGURE 6 – Paramétrage d'une `AppInteraction` dans l'inspecteur

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)

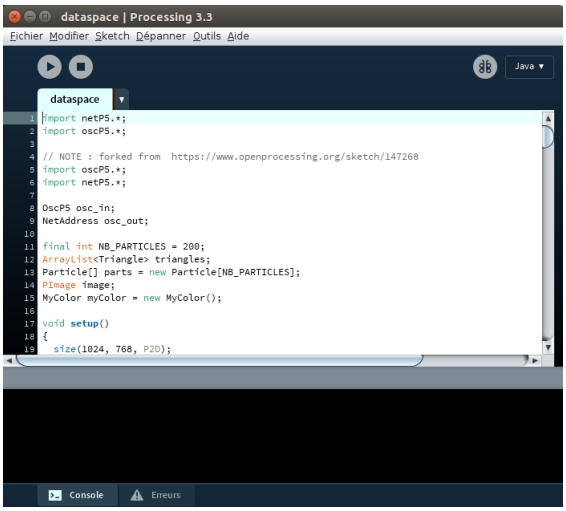

FIGURE 7 – Exemple chargé dans `Processing`

— 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.

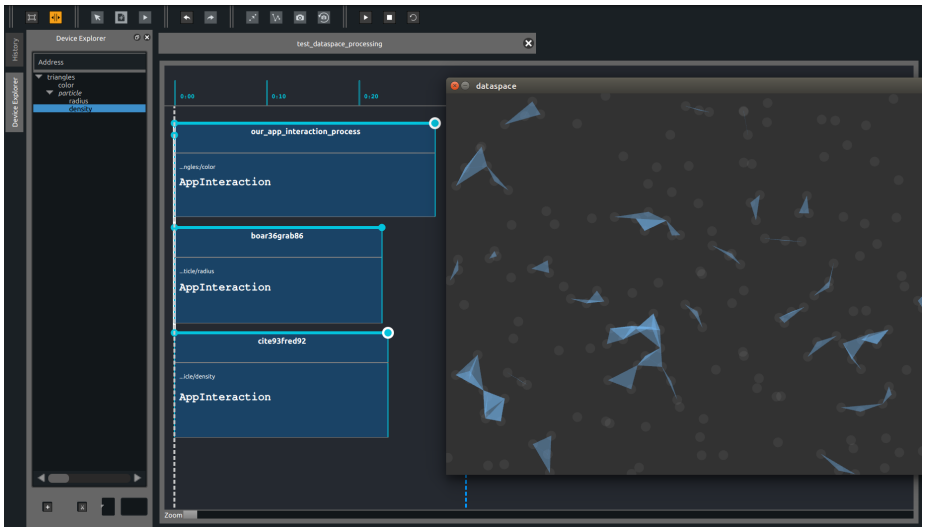

FIGURE 8 – Partition *i-score* et affichage de Processing avant lecture de la partition

— 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.

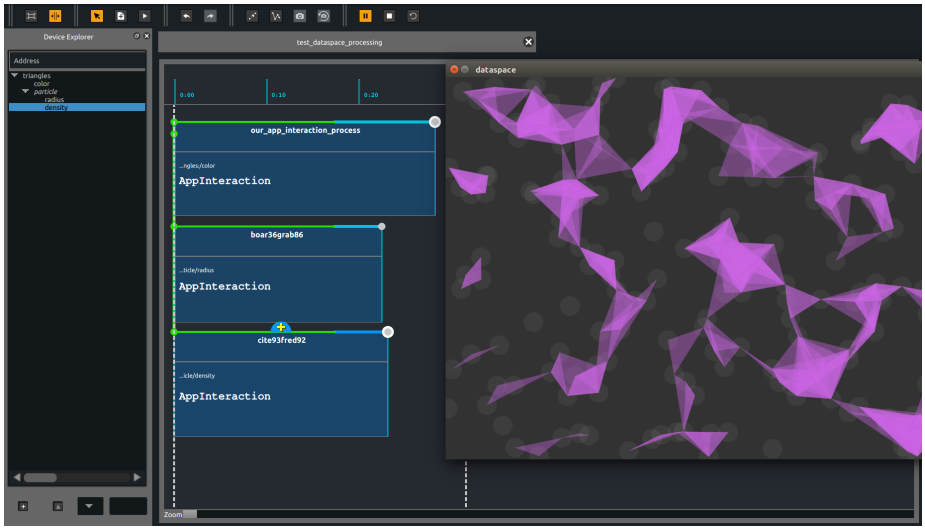

FIGURE 9 – Lecture de la partition et affichage de Processing modifié

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.

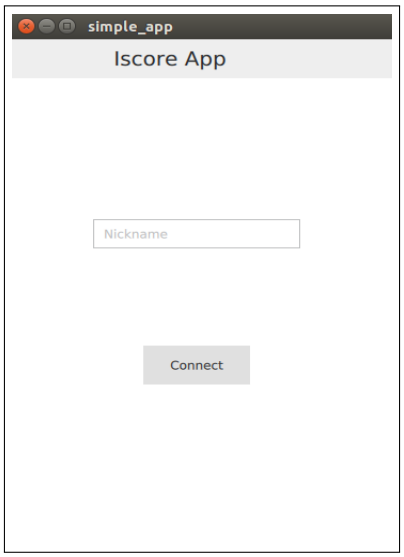

FIGURE 10 – Écran de connexion

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.

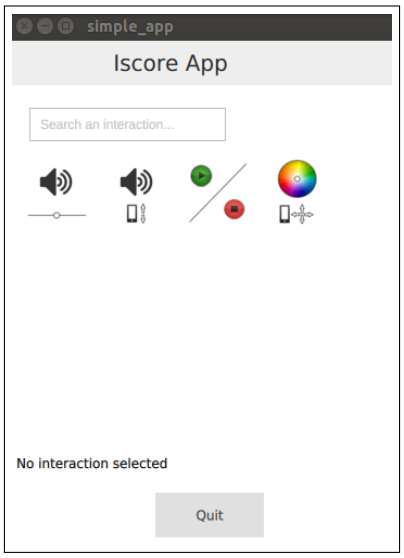

FIGURE 11 – Menu d'interactions

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.

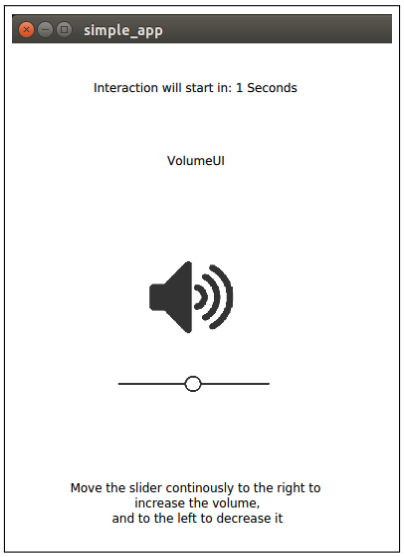

FIGURE 12 – Écran d'attente pour l'interaction VolumeUI

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.

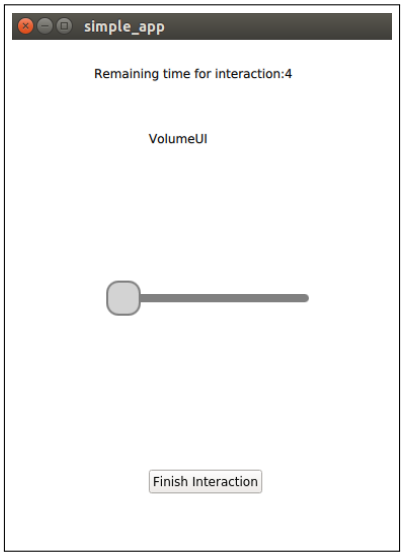

FIGURE 13 – IHM de l'interaction VolumeUI

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*.

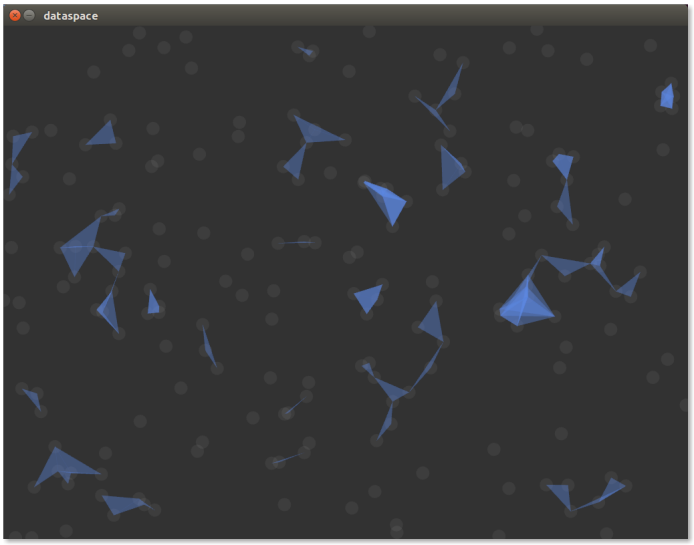

FIGURE 14 – Rendu visuel du programme `Documentation/Examples/-Processing/Dataspace/dataspace/dataspace.pde`

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.

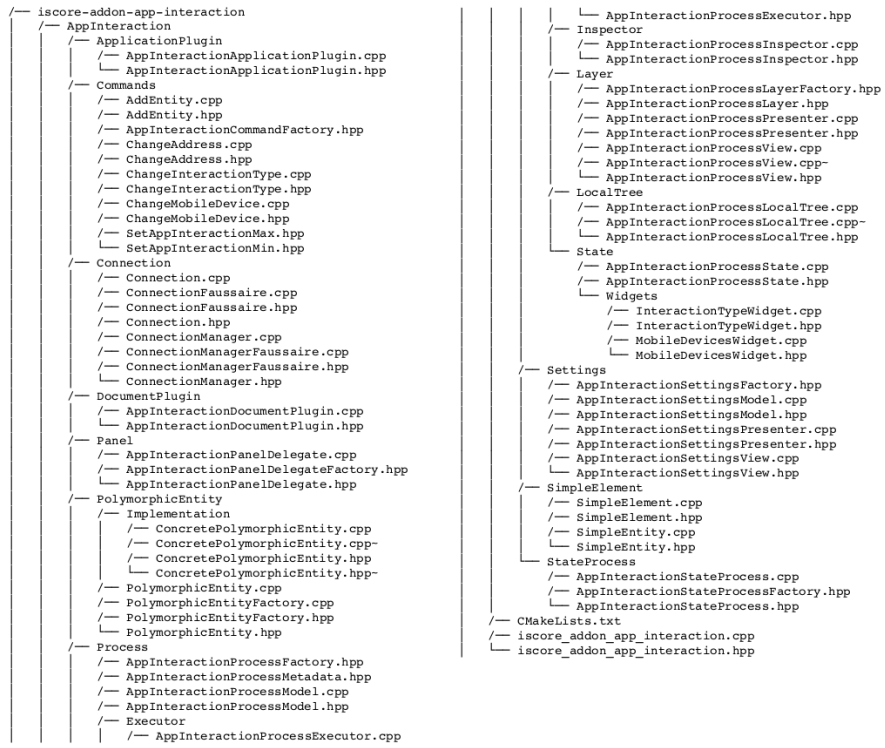

FIGURE 15 – Arborescence des fichiers sources plugin

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`

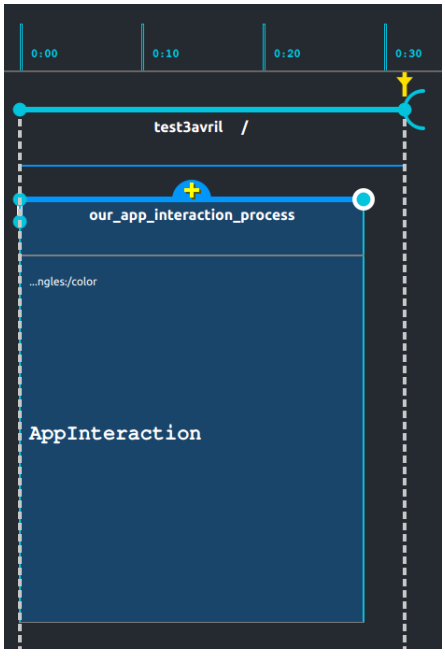

FIGURE 18 – Partition contenant un `Process AppInteraction`

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.

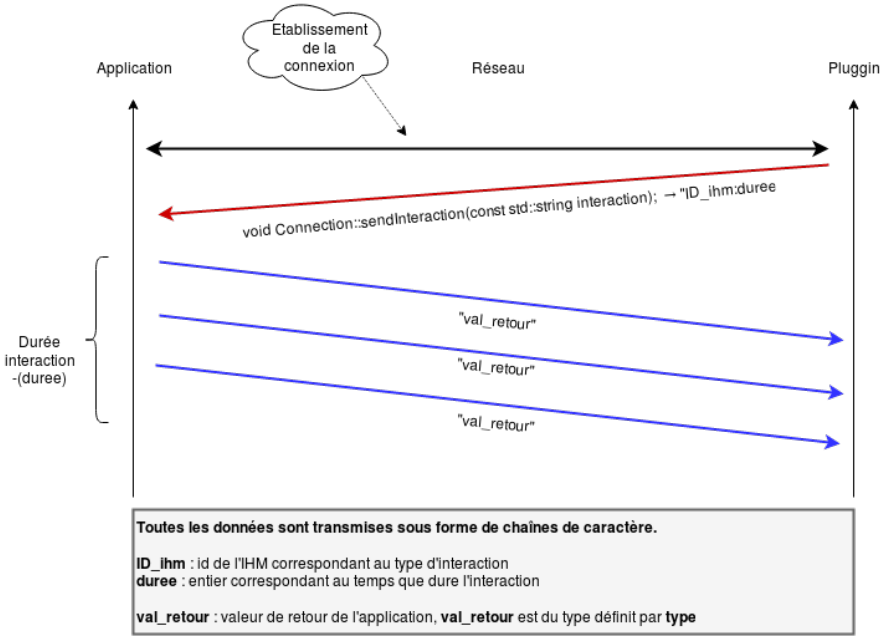

FIGURE 19 – Diagramme d'échange de données

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.