Rapport de stage de fin d'études

Interaction à distance et reconnaissance de gestes avec le capteur Leap Motion

Marie Pietrowski

Du 18 mars au 18 septembre 2013

Tuteur IntuiLab : Bruno Marchesson

Tuteur ENSEEIHT : Vincent Charvillat

IntuiLab ▪ Les Triades A ▪ 130 Rue Galilée ▪ 31670 LABEGE

Remerciements

Je tiens tout d'abord à remercier vivement Bruno Marchesson pour ces précieux conseils qui m'ont permis d'améliorer mon travail et d'enrichir mes connaissances.

Je remercie également Vincent Encontre de m'avoir permis d'effectuer ce stage à IntuiLab ainsi que l'ensemble de l'IntuiTeam pour son accueil chaleureux.

Un grand merci au stagiaire Rémy Bauer qui, par son aide et le partage de ses connaissances m'a beaucoup appris tout au long de mon stage.

Je souhaite par la même occasion remercier toute l'équipe enseignante du cursus IMA sans qui je n'en serais pas là aujourd'hui.

Enfin, c'est avec tout mon amour que je remercie ma famille, qui a toujours su me soutenir.

Sommaire

Remerciements 2
Sommaire 3
1. Introduction 5
2. Contexte 6
2.1. Présentation de l'entreprise 6
2.2. Présentation d'IntuiFace 6
2.3. Présentation des langages et formats utilisés à IntuiLab 8
2.4. Objectifs du stage 9
2.5. Présentation du Leap Motion 9
3. Le travail effectué sur la détection de gestes et de postures 11
3.1. Travail préliminaire 11
3.2. Mise en place d'un premier algorithme de détection des gestes et des postures 12
3.2.1. Reconnaissance de gestes 12
3.2.2. Reconnaissance de postures 14
3.3. Création d'un visualiseur et processus de tests 16
3.3.1. Phase 1 : enregistrement des gestes 16
3.3.2. Phase 2 : visualisation et analyse des gestes 17
3.3.3. Phase 3 : rejeu des gestes 20
3.3.4. Phase 4 : validation 20
3.4. Amélioration des algorithmes 21
3.4.1. Reconnaissance de gestes : mise en place d'un buffer de gestes 22
3.4.2. Reconnaissance de postures : ajout d'évènements détection en cours/détection perdue 24
3.5. Intégration au Composer 26
3.5.1. La notion de Façade 26
3.5.2. Le descripteur JSON 28
3.5.3. Les design accelerators 28
4. Le travail effectué sur le pointage et la manipulation directe 34
4.1. Travail préliminaire 34
4.2. Mise en place d'un premier algorithme de pointage 36
4.3. Intégration au Composer 38

4.4. Premiers tests 39
5. Ce qu'il reste à faire 42
5.1. Sur la détection de gestes et postures 42
5.2. Sur le pointage 42
6. Les difficultés rencontrées 43
7. Les bénéfices de cette expérience 44
8. Conclusion 45
Webographie 46
Annexes 47
A. Etat de l'art scientifique et technologique sur la reconnaissance de gestes 47
B. Exemple de fichier CSV d'enregistrement de geste 52
C. Exemple de fichier de statistiques 53
D. Extrait du descripteur 54
E. Etat de l'art sur la manipulation directe dans une IHM avec des interactions à distance 56
F. Diagramme de classe simplifié de la DLL 60

1. Introduction

Ce rapport présente mon stage de fin d'études dans le cadre de ma formation d'ingénieur en Informatique et Mathématiques Appliquées de l'ENSEEIHT, que j'ai eu l'opportunité d'effectuer au sein de l'entreprise IntuiLab de mars à septembre 2013.

La société a assis sa notoriété sur le développement d'interactions homme-machine innovantes, notamment tactiles et multi-touch. Avec l'essor des mécanismes d'interactions à distance, IntuiLab souhaitait élargir son offre et intégrer ces nouvelles technologies à sa solution IntuiFace.

Le but de ce stage était donc de développer de nouveaux modes d'interactions sur ces technologies émergentes d'interactions à distance. La société désirait introduire dans sa solution deux modules basés sur deux capteurs différents : le Leap Motion et la Kinect.

Nous étions donc deux stagiaires à travailler en parallèle sur ce sujet, et tandis que Rémy Bauer développait le module pour Kinect, j'étais en charge de coder celui pour le Leap Motion.

Dans ce rapport, je vais commencer par présenter la société IntuiLab ainsi que sa solution logicielle IntuiFace et les moyens techniques qu'elle utilise, puis je décrirai plus en détails les objectifs qui m'ont été confiés. Ensuite, je présenterai l'ensemble du travail que j'ai effectué durant ces 6 mois et les résultats obtenus. Enfin, je finirai par un bilan de cette expérience.

2. Contexte

2.1. Présentation de l'entreprise

Fondée en 2002 et située à Labège, IntuiLab est un leader mondial dans la conception d'applications multi-touch pour la création d'expériences utilisateur interactives.

La société réalise des partenariats avec des acteurs de différents niveaux et possède des clients dans plusieurs domaines tels que la grande distribution, la banque, l'assurance, les télécoms, l'immobilier ou encore le secteur de la mode.

Au début, la mission d'IntuiLab consistait à répondre aux besoins spécifiques de ses clients en créant directement des présentations interactives sur écran tactile à la demande. Aujourd'hui, l'entreprise a acquis le statut d'éditeur de logiciel grâce à sa plateforme logicielle nommée **IntuiFace**, qui permet de créer facilement des présentations numériques tactiles sans avoir de connaissances en programmation et de les exécuter sur n'importe quel équipement tactile (table, écran …).

La société mobilise une équipe pluridisciplinaire d'une trentaine d'employés (l'IntuiTeam) composée de concepteurs IHM, d'ergonomes, de graphistes et de développeurs.

2.2. Présentation d'IntuiFace

IntuiFace est composé de deux outils séparés, le Composer et le Player.

Le **Composer** est l'outil de création graphique qui permet de concevoir des expériences interactives. L'utilisateur peut ajouter à sa présentation toutes sortes d'éléments (des éléments simples comme des collections d'images, des vidéos, des textes, des boutons, des pages web ou des éléments plus complexes tels qu'un chronomètre, la météo en temps réel, une liste de « tweets », etc), importer du contenu ou encore ajouter des fonctions de navigation. Il peut surtout lier le déclenchement d'évènements spécifiques à des actions sur sa présentation afin de la rendre dynamique (par exemple l'évènement « appui sur un bouton » change la ville de référence de l'élément météo).

Les objets de présentation de contenu, comme les images, sont appelés des **assets**. Les **collections** sont des groupes d'assets. Tous ces composants ont des propriétés configurables. Les évènements que l'on peut lier à des actions sont nommés des **triggers**. La dernière notion est celle d'**interface asset** : ce sont des assets qui permettent l'accès aux propriétés, actions et triggers associés à des services extérieurs au Composer, comme une DLL C#, un web service ou encore un fichier Excel. C'est ce concept que l'on retiendra particulièrement, car le plugin Leap Motion à développer est en fait un interface asset.

Sur cette image, on peut voir que l'interface du Composer est très intuitive :

Figure 1- Capture d'écran de l'interface du Composer

Le **Player** est l'outil qui permet d'exécuter les présentations créées avec le Composer.

Il existe également un outil qui permet d'utiliser un terminal mobile (iPad, smartphone ou tablette Android, etc) comme télécommande pour naviguer et interagir avec la présentation à distance, appelé **IntuiPad**.

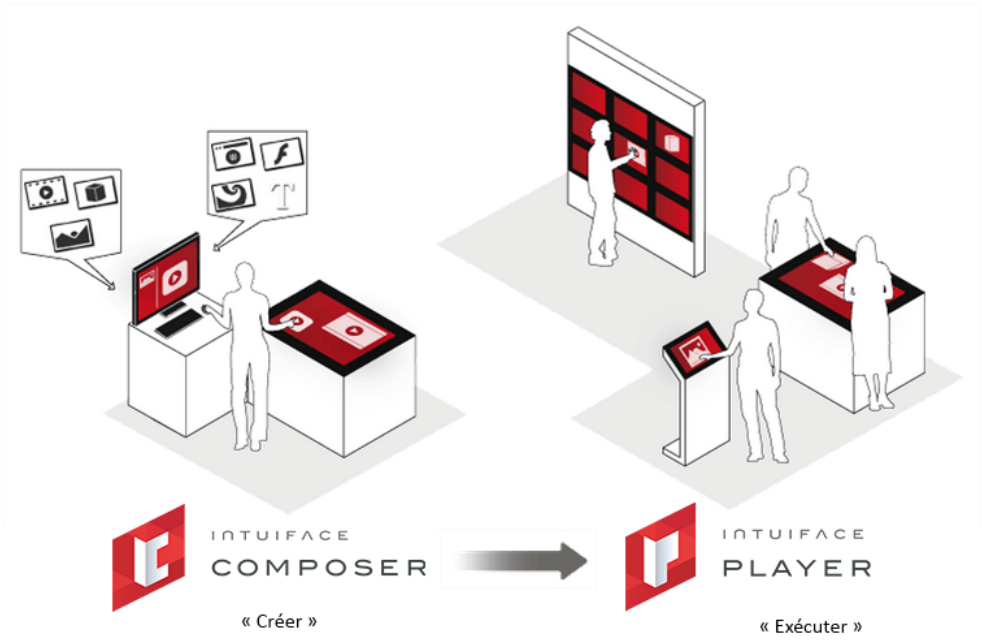

Figure 2 – IntuiFace Composer et IntuiFace Player

2.3. Présentation des langages et formats utilisés à IntuiLab

Le langage de programmation utilisé à IntuiLab et avec lequel j'ai développé le module d'interactions à distance Leap est le langage objet **C#**. Ce langage fait partie du framework .NET de Microsoft. Les interfaces graphiques sont réalisées en WPF (Windows Presentation Foundation). Les bibliothèques de classes C# sont appelées des **DLL** (Dynamic Link Library). Durant mon stage, j'ai donc travaillé sous un environnement Windows et programmé avec l'environnement de développement Microsoft Visual Studio.

Les présentations créées avec le Composer sont stockées dans des fichiers portant l'extension **.ifx** propre à IntuiLab. Ce sont en fait des fichiers XML (Extensible Markup Language), un langage à balise qui facilite l'échange automatisé de contenus complexes entre systèmes d'informations.

Comme indiqué plus haut, les **interface assets** sont un moyen mis en place par la plateforme pour étendre ses fonctionnalités sans imposer de contraintes techniques à cette extension. Néanmoins, le Composer a alors besoin d'un **descripteur** pour gérer l'interface asset et le présenter à l'utilisateur. Ce fichier de description est un fichier JSON (JavaScript Object Notation) portant l'extension spécifique à l'entreprise **.ifd** (pour IntuiFace Descriptor) et permet d'identifier quels services et propriétés seront exposés au Composer. Le JSON est un format de données permettant de représenter de l'information structurée qui offre une alternative au format XML. Il présente l'avantage d'être simple à écrire et facilement lisible par un humain comme par une machine. La syntaxe utilisée par IntuiLab pour ce fichier est basée sur le format Google Discovery Service avec quelques extensions.

Enfin, un dernier type de fichier est utilisé par le Composer, il s'agit des **design accelerators**. Ces fichiers, qui portent l'extension **.ifa**, sont en fait des bouts d'ifx (donc en XML) qui définissent un graphisme et des liaisons triggers-actions par défaut pour les interface assets. Cela donne à l'utilisateur du Composer un exemple d'utilisation pour chaque interface asset, et permet au composant d'être utilisable de suite, dès son ajout à la présentation. Par exemple, lorsque l'on ajoute un chronomètre, ce graphisme, qui peut être personnalisé par la suite, apparaît directement :

Figure 3 – Exemple de design accelerator

Lorsque l'interface asset est glissé dans la présentation, son fichier ifa est copié au bon endroit dans le fichier de la présentation.

2.4. Objectifs du stage

La possibilité d'interagir avec un ordinateur par les gestes permet d'enrichir l'expérience utilisateur en proposant des interfaces homme-machine plus naturelles et intuitives.

Le **Leap Motion** est un nouveau périphérique de détection de mouvement sorti le 22 juillet 2013 et pré-accessible aux développeurs depuis quelques mois. Ainsi, IntuiLab souhaitait intégrer à sa plateforme IntuiFace des interactions gestuelles avec le Leap.

Le but de mon stage a donc été de développer une DLL en C# de **reconnaissance de gestes et de postures** avec le Leap , puis de l'intégrer à IntuiFace sous la forme d'interface assets tout en réfléchissant aux bonnes pratiques ergonomiques à mettre en œuvre pour l'interaction à distance avec l'aide d'un ergonome professionnel. Dans un deuxième temps, je devais développer une solution de **pointage et manipulation directe** avec le Leap, c'est-à-dire un moyen d'utiliser ses doigts comme curseurs et de simuler les touchs propres aux interfaces tactiles.

Bien que le Leap soit un périphérique destiné tout d'abord au pointage, nous avons décidé de commencer par développer la détection de gestes et postures. Ceci s'explique par le fait que le pointage est techniquement plus compliqué à implémenter. Si nous avions commencé par ce dernier, il est probable que nous n'ayons pas eu le temps de travailler sur les gestes et postures. Au début du stage, nous ne savions pas encore comment l'intégrer au Composer. De plus, le même travail était réalisé en parallèle pour le capteur Kinect qui à l'inverse, est plutôt destiné à la reconnaissance de gestes et de postures.

Le même travail a été réalisé par un second stagiaire sur le capteur Kinect de Microsoft.

2.5. Présentation du Leap Motion

Le contrôleur Leap Motion est un petit boîtier de quelques centimètres qui se branche à l'ordinateur par USB, créé par la société du même nom fondée en 2010 et localisée à San Francisco.

Figure 4 – Photo du Leap Motion

Il est composé de 2 caméras et de 3 LEDs infrarouges qui délivrent des images brutes traitées ensuite par le CPU. Il est capable de suivre les mains et les doigts ou les outils similaires de l'utilisateur à une

précision de 0,01 millimètre dans l'espace. Son champ de vision est une pyramide inversée d'environ 1 mètre, bien qu'au-delà de 60 centimètres au-dessus de l'appareil, on perde beaucoup de précision.

La version développeur est livrée avec un kit de développement (SDK) disponible pour plusieurs langages de programmation (Java, C#, C++, Python, JavaScript, ObjectiveC) qui permet de développer des applications compatibles. Grâce au SDK, le Leap fournit toutes sortes d'informations sur les mains et doigts/outils de l'utilisateur, telles que la position 3D de la paume de la main, la position 3D des doigts, la vitesse de ces points, la longueur des doigts, la rotation/translation de la main d'une frame à une autre, etc…

Ces informations sont délivrées si disponibles à chaque frame. Le frame rate du Leap varie au cours du temps selon la complexité des calculs à effectuer sur les frames, et oscille entre 200 et 300 Hertz (frames par seconde).

Le périphérique est également livré avec un visualiseur dont voici une capture d'écran :

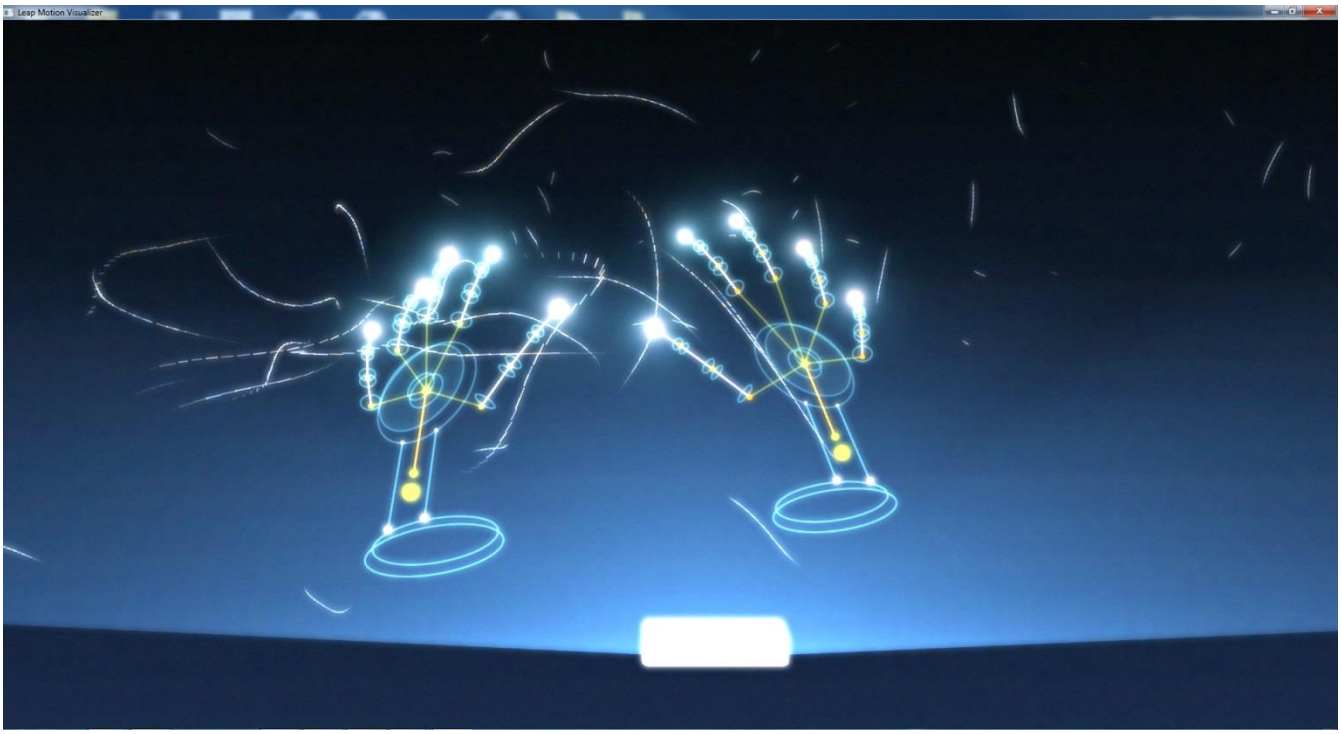

Figure 5 – Capture d'écran du visualiseur Leap Motion

3. Le travail effectué sur la détection de gestes et de postures

3.1. Travail préliminaire

Pour commencer, je me suis familiarisée avec le capteur Leap et j'ai étudié la documentation de l'API fournit par le SDK. L'appareil emploie un système de coordonnées cartésien défini comme suit :

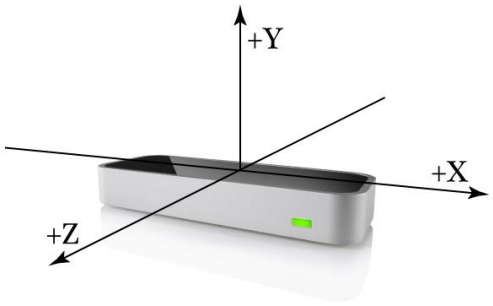

Figure 6 – Système de coordonnées du Leap

Toute application qui utilise le Leap doit dériver de la classe *Listener* qui permet d'écouter les évènements envoyés par le périphérique en implémentant les callbacks *OnInit*, *OnConnect*, *OnDisconnect*, *OnExit* et *OnFrame*. La fonction *OnFrame* est appelée à chaque nouvelle frame disponible, et c'est à l'intérieur que doit être écrit le code qui va récupérer et analyser toutes les informations renvoyées par le capteur.

L'API offre déjà une classe qui détecte quatre types de gestes : les cercles, les déplacements horizontaux (swipes), les key taps qui miment l'appui sur une touche de clavier et les screen taps qui miment l'appui sur un écran tactile.

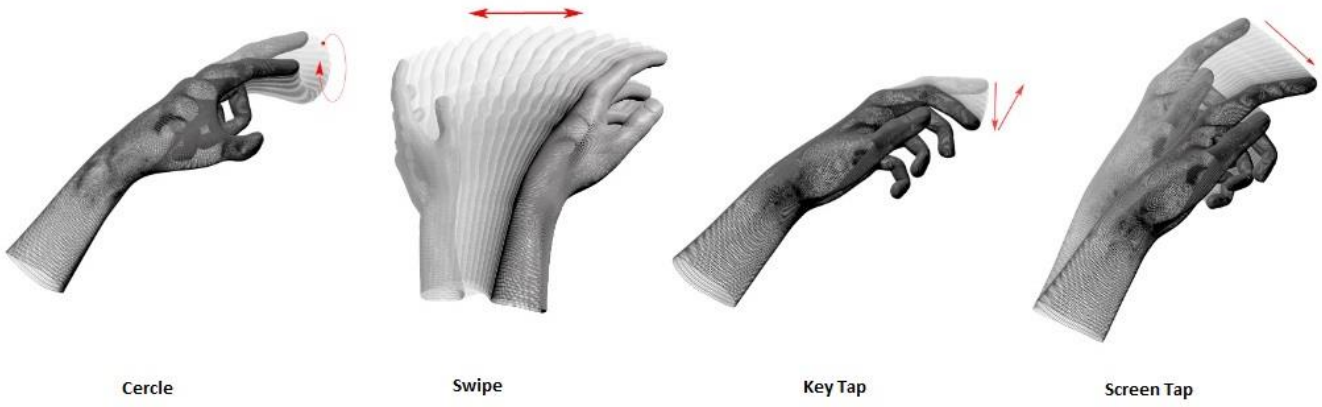

Figure 7 – Les gestes proposés par l'API Leap

Le test de l'algorithme de détection de l'API a été peu concluant. En effet, la détection des gestes est trop sensible, elle révèle beaucoup de faux positifs. Un faux positif est un résultat déclaré positif là où il est en réalité négatif. Par exemple lorsque l'on positionne sa main avant de faire un geste voulu, l'algorithme détecte un mouvement que l'on ne veut pas. J'ai créé une présentation IntuiFace où l'on pouvait naviguer avec ces gestes que j'ai faite essayer aux ergonomes et à mon maître de stage, mais nous avons tous trouvé qu'il y avait trop de détections intempestives. En effet, lorsque l'on se sert de gestes pour naviguer, la main bouge souvent pour se replacer et enchaîner les gestes. Nous avons donc conclu que nous allions plutôt coder notre propre algorithme de reconnaissance.

Dans un deuxième temps, j'ai réalisé un **état de l'art** sur les capteurs de mouvements destinés aux interactions homme-machine ainsi que sur les algorithmes de détection de gestes et postures existants. Vous pouvez retrouver ce document en annexe (annexe A).

On distingue deux types de périphériques de capture, ceux basés sur le contact qui agissent comme des extensions du corps, et ceux basés sur la vision par ordinateurs qui utilisent des caméras pour analyser et interpréter les mouvements en se basant sur le traitement d'images. Le Leap fait partie de cette dernière catégorie. Lors de mes recherches, j'ai pu constater que la plupart des algorithmes de reconnaissance existants nécessitent de l'apprentissage, comme les modèles de Makov cachés ou les réseaux de neurones. Or, l'utilisation du Leap dans la plateforme IntuiFace ne doit pas comporter de phase d'apprentissage, et doit être rapide à prendre en main. Il a donc fallu trouver un algorithme de détection adapté au plus grand nombre d'utilisateurs possible, et qui n'impose pas d'effectuer les gestes d'une manière précise mais naturelle et instinctive, sans phase d'adaptation trop longue.

3.2. Mise en place d'un premier algorithme de détection des gestes et des postures

3.2.1. Reconnaissance de gestes

J'ai tout d'abord déterminé un panel de gestes à coder. En accord avec les ergonomes, nous avons choisi quatre mouvements qui ont la particularité d'être linéaires : le **swipe left**, le **swipe right**, le **tap**, et le **push**.

Swipe Left Swipe Right Tap

Figure 8 – Les gestes retenus

J'ai donc commencé à chercher un algorithme capable de détecter ces gestes chez le plus d'utilisateurs possibles sans apprentissage. J'ai implémenté une solution de reconnaissance dynamique, c'est-à-dire qui utilise plusieurs frames et se sert de données temporelles. La détection est basée sur deux paramètres, la vitesse et la direction du mouvement. De plus, la direction doit être globalement linéaire. Pour calculer les vitesses et les directions, j'ai utilisé la position de l'extrémité du doigt le plus à l'avant, c'est-à-dire celui de plus petit Z. Ainsi, que l'utilisateur fasse le geste avec un seul doigt ou plusieurs, les conditions de reconnaissance restent les mêmes. Voici le principe de l'algorithme, qui prend en paramètre une liste de frames :

```
Algorithme ReconnaissanceDeGestes (frames)
Début
    Si le doigt prioritaire est présent dans toutes les frames alors
        Calcul des directions du mouvement d'une frame à une autre avec
        les deux positions successives du doigt
        Si le mouvement global est linéaire alors
            Calcul de la vitesse globale du mouvement
            Calcul de la direction globale avec les deux positions
            extrêmes du doigt
            Si les conditions du tap sont valides alors
                Lever l'évènement « tap »
            Sinon si les conditions du push sont valides alors
                Lever l'évènement « push »
            Sinon si les conditions du swipe right sont valides alors
                Lever l'évènement « swipe right »
            Sinon si les conditions du swipe left sont valides alors
                Lever l'évènement « swipe left »
            Finsi
        Finsi
    Finsi
Fin
```

Toute la bibliothèque de classes produite fonctionne sur le principe d'**évènements**. Lorsqu'un geste est détecté, un évènement est envoyé et capté par la couche supérieure, puis relayé jusqu'à l'application qui utilise la DLL. Cette fonction est appelée à chaque frame, avec en entrée une liste des dernières frames dont celle actuelle.

Pour déterminer si le mouvement est linéaire, j'ai choisi de comparer les angles entre chaque vecteur direction (vecteur direction = position d'arrivée – position de départ). Ainsi, si j'utilise cinq frames, j'ai cinq positions pour le doigt et donc quatre vecteurs direction. Je calcule les six angles entre ces vecteurs, si tous ces angles sont proches de zéro, je considère que le mouvement décrit dans ces cinq frames est linéaire. Après plusieurs tests, j'ai choisi la valeur $\pi/6$ radian (≈30°) comme borne supérieure pour les angles. Ce schéma illustre ma méthode :

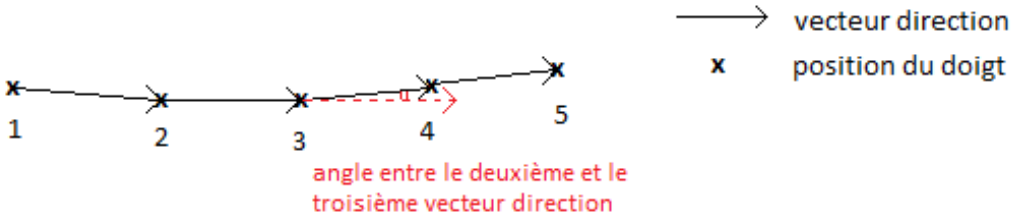

Figure 9 – Schéma d'illustration pour la méthode de test de linéarité

Les conditions sur les gestes concernent la vitesse et la direction globale du mouvement. Pour chaque geste, la vitesse doit être comprise dans un certain intervalle (par exemple entre 1000 et 4000 millimètres/seconde pour les swipes). Les composantes du vecteur direction global, qui représentent en fait le déplacement en X, Y et Z du doigt, doivent respecter certaines règles. Par exemple pour le tap, qui suit approximativement l'axe Y du système de coordonnées du Leap, la variation en X doit être faible (entre -15 et 15 millimètres).

Le problème est que tous ces paramètres qui constituent des bornes pour la vitesse et la direction sont arbitraires, et doivent correspondre aux gestes du plus grand nombre de personne. Au départ, j'ai effectué les réglages avec ma propre façon d'effectuer les gestes, puis j'ai mis en place un processus de tests et de validation des algorithmes où j'ai pu prendre en compte les gestes de tous les employés d'IntuiLab. Cette phase permettra de choisir les meilleurs paramètres de détection, afin que l'algorithme détecte les gestes du plus grand nombre d'utilisateurs possible. Elle permettra aussi de confirmer l'algorithme à chaque fois que des modifications lui seront apportées, et à valider l'algorithme final grâce aux calculs des taux de réussite dans la détection des quatre gestes. Cette phase du projet sera décrite dans la section suivante.

Après avoir codé cet algorithme de reconnaissance de gestes, j'ai élaboré un algorithme de détection de postures.

3.2.2. Reconnaissance de postures

Une posture est une configuration de la main dans laquelle elle se maintient, immobile, durant un certain temps.

Nous avons décidé de coder les postures du jeu pierre-feuille-ciseaux ainsi que des postures de comptage de doigts de zéro à cinq.

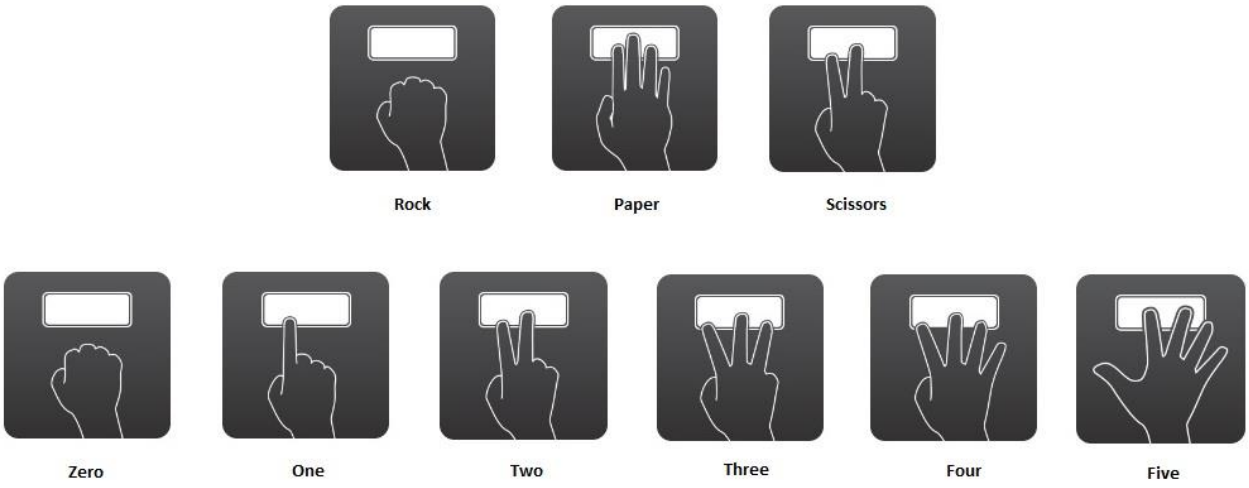

Figure 10 – Les gestes retenus

Ces postures seront divisées dans le Composer en deux interface assets différents, c'est pour cela qu'on implémente les postures rock, paper et scissors bien qu'elles soient les mêmes que les postures zero, five et two.

La reconnaissance est basée sur l'immobilité de la main ainsi que sur le nombre de doigts visibles. L'algorithme prend lui aussi une liste de frames en paramètre, voici son principe :

```
Algorithme ReconnaissanceDePostures (frames)
Début
    Si la main est présente dans toutes les frames alors
        Stockage des positions successives de la paume de la main
        Si la main est immobile et il y a le même nombre de doigts dans
        toutes les frames alors
            Lever l'évènement de la posture correspondante
        Finsi
    Finsi
Fin
```

Pour déterminer si la main est immobile, je calcule les coordonnées minimales et maximales suivant chaque axe de l'ensemble des positions de la paume de la main. Ces coordonnées définissent une sorte de boîte dont les dimensions doivent être faibles pour considérer la main comme immobile. Je calcule donc la distance entre la coordonnée minimale et celle maximale pour chaque axe, qui doit être infime (inférieure à 3 millimètres). La marge correspond au tremblement de la main qui ne sera jamais complètement immobile. Je compte ensuite le nombre de doigts rattachés à cette main, qui doit être constant sur l'ensemble des frames, pour lever l'évènement correspondant à la posture détectée.

A ce stade, mon projet comportait seulement le Listener ainsi qu'une classe de reconnaissance de gestes et une de postures :

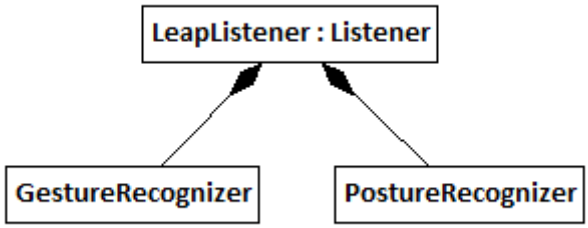

Figure 11 – Architecture initiale

3.3. Création d'un visualiseur et processus de tests

Chaque personne a sa propre façon d'effectuer les gestes. Pour prendre en compte ces différences, et avoir un large panel de gestes afin de tester les algorithmes et définir les meilleurs paramètres, il a fallu trouver un moyen d'analyser les gestes de plusieurs utilisateurs. En effet, avant la mise en place de ce processus de tests, les taux de reconnaissance étaient médiocres, et très hétérogènes suivant les personnes. Grâce à cette partie, on espère pouvoir améliorer l'algorithme de détection afin d'une part, de reconnaître les gestes de plus d'utilisateurs, et d'autre part, d'arriver à des taux de réussite de l'ordre de 75/80%. Pour cela, j'ai créé un visualiseur 3D en **OpenGL** qui permet d'enregistrer à chaque frame les données nécessaires à la reconnaissance gestuelle, puis de rejouer ces frames comme si elles provenaient du Leap et ainsi visualiser les gestes en 3D. Cela m'a permis d'une part de pouvoir analyser visuellement les gestes et d'autre part de pouvoir rejouer, une fois l'algorithme modifié, des gestes non reconnus au début afin d'améliorer le taux de réussite de la détection et limiter les faux positifs. Ce visualiseur ne fait pas partie de la DLL, il se contente de l'utiliser.

3.3.1. Phase 1 : enregistrement des gestes

Tout d'abord, les informations utiles à la reconnaissance sont stockées dans des fichiers **CSV** (Comma-Separated Values), où chaque ligne représente une frame. Vous pouvez trouver en annexe un de ces fichiers (annexe B). L'enregistrement se fait depuis la DLL.

J'ai donc enregistré les gestes d'une quinzaine de personnes afin d'avoir un premier panel de test. Pour guider les testeurs, j'ai utilisé des images représentant des cas d'utilisations pour chaque geste : tourner les pages d'un livre pour les swipes, appuyer sur un bouton pour le push, et appuyer sur une touche de clavier pour le tap. La détection n'était pas activée pendant les enregistrements.

3.3.2. Phase 2 : visualisation et analyse des gestes

Ensuite, le visualiseur lit les fichiers CSV et reconstruit la scène 3D, permettant ainsi de visualiser les gestes dans un repère 3D. La visualisation concerne les positions des extrémités des doigts, ainsi que la vitesse modélisée par la couleur. On peut choisir un affichage discret avec seulement les positions réelles, ou bien un affichage linéaire avec interpolation. On peut aussi choisir d'afficher tous les doigts, ou seulement celui qui est utilisé par l'algorithme. Ceci permet de faire une première analyse visuelle des gestes. Ci-dessous, des captures d'écran pour illustrer mes propos. Le point vert indique la position de départ.

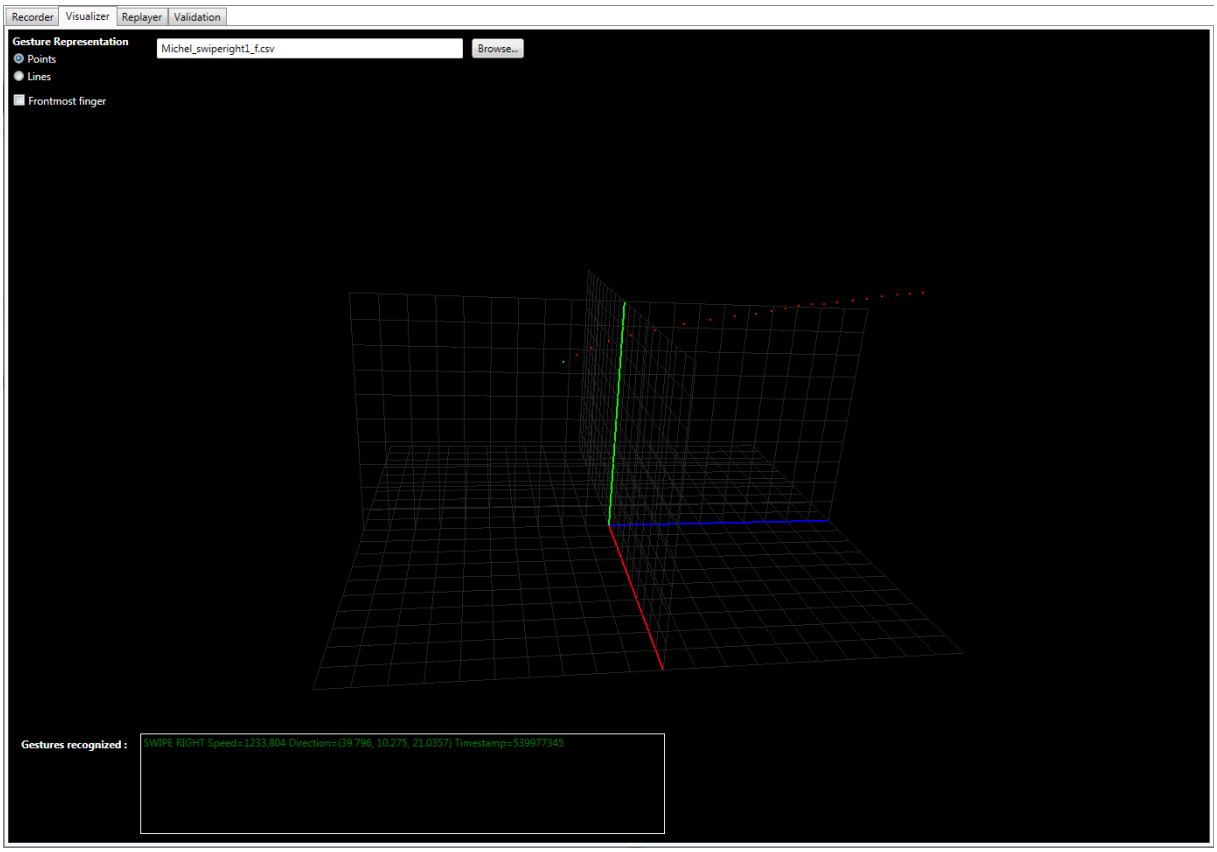

Figure 12 – Visualisation d'un swipe right avec affichage discret

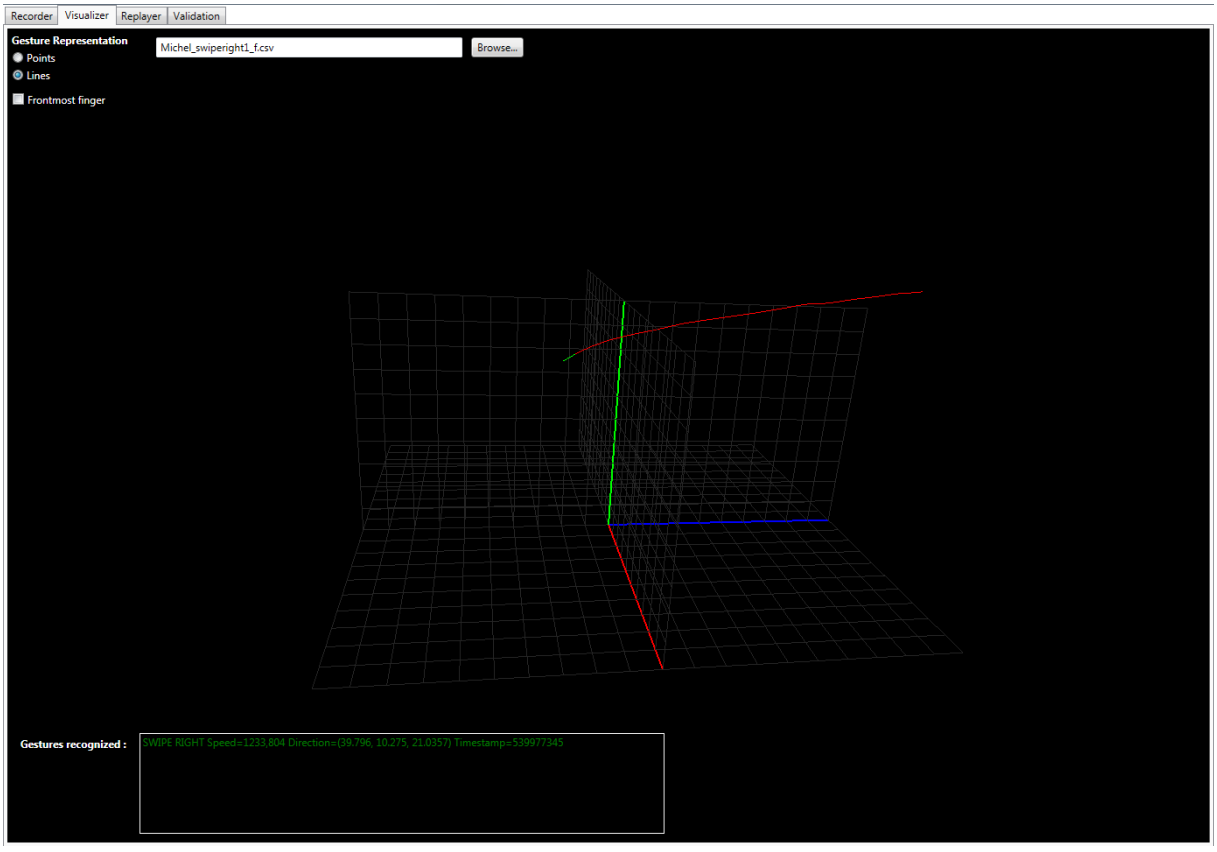

Figure 13 – Visualisation d'un swipe right avec affichage linéaire

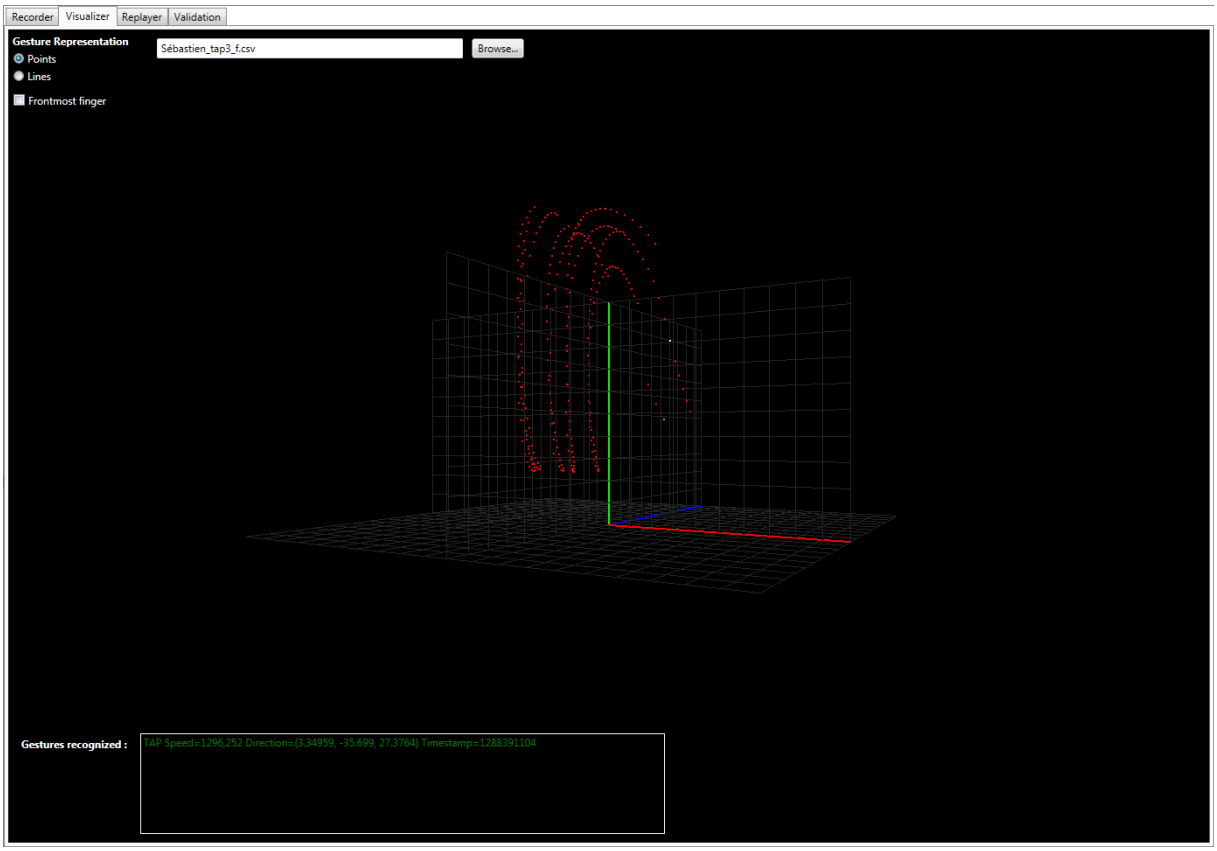

Figure 14 – Visualisation d'un tap avec affichage de tous les doigts

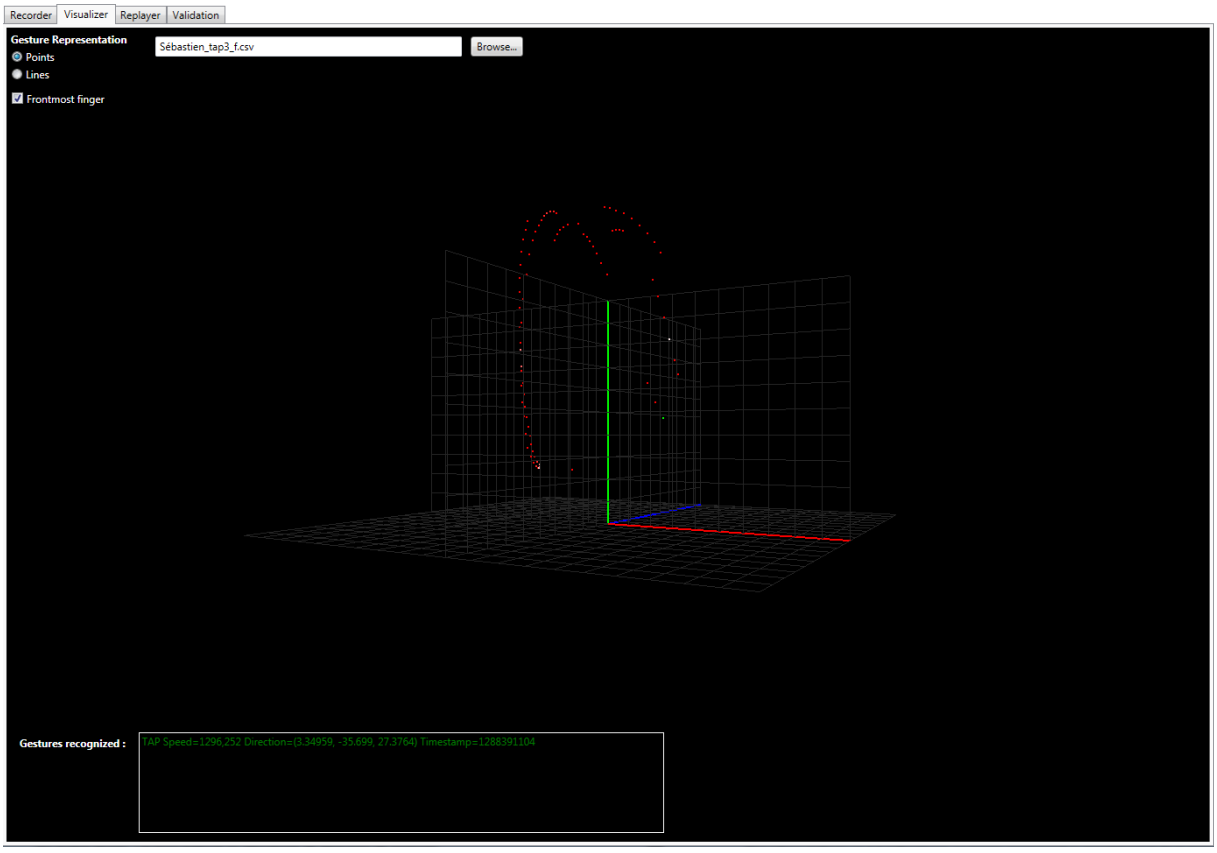

Figure 15 – Visualisation d'un tap avec affichage seulement du doigt le plus en avant

3.3.3. Phase 3 : rejeu des gestes

Les informations contenues dans les fichiers sont en même temps retransmises à la DLL sous forme de frames reconstruites afin de savoir si les gestes sont bien reconnus. Pour réinjecter les données dans l'algorithme de reconnaissance comme si elles provenaient du Leap, j'ai dû écrire ma propre classe de frames, car il est impossible de créer une instance de la classe *Frame* de l'API Leap qui est en lecture seule. Mes algorithmes de détection prennent alors en entrée une *IFrame*, une interface implémentée par *LeapFrame* qui reprend telles quelles les données de la classe *Frame* du Leap, et *ReplayFrame* dont les instances sont construites à partir des données des fichiers CSV. Mon architecture s'enrichit aussi d'une classe *Scheduler* qui s'occupe d'ordonner les informations avant de les envoyer aux moteurs de reconnaissance de gestes et de postures.

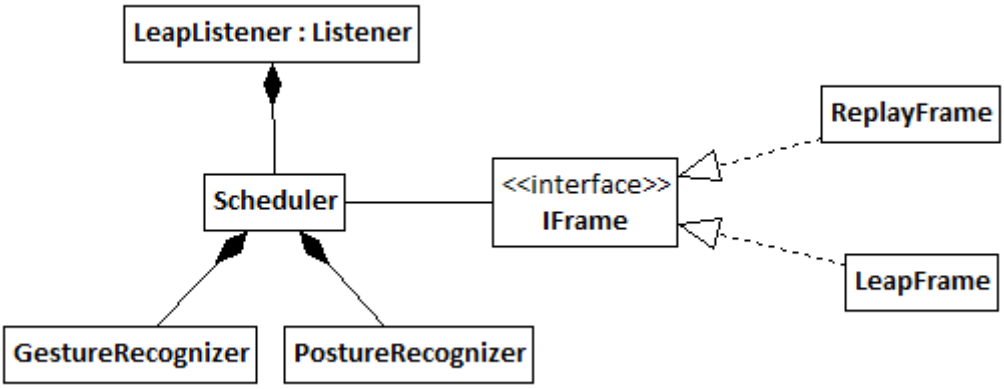

Figure 16 – Architecture après développement du « rejeu »

Grâce à cela, on peut aussi avoir de plus amples informations sur les gestes comme la vitesse et la direction globale, et ainsi ajuster les paramètres de détection et mieux comprendre pourquoi un geste qui devrait être détecté ne l'est pas.

J'ai ensuite développé un programme de statistiques afin de calculer le taux de réussite de mon algorithme de gestes. Ce dernier prend en entrée l'ensemble des enregistrements d'un même geste et calcule le pourcentage de fichiers où le bon geste est reconnu sans faux positifs, qui pour rappel sont les cas où un geste est détecté alors qu'il n'est pas intentionnel et ne devrait donc pas être reconnu. Vous trouverez en annexe un exemple de fichier de statistiques (annexe C), où sont notés les enregistrements non reconnus ainsi que le taux de réussite.

3.3.4. Phase 4 : validation

Pour finir, j'ai créé une partie de validation qui présente les « consignes » des gestes à effectuer et où la détection est activée. Elle permettra de tester l'algorithme de reconnaissance à chaque fois que des modifications lui seront apportées avec le même panel de gestes enregistrés au début, et surtout de valider l'algorithme final afin de pouvoir ensuite l'intégrer au Composer.

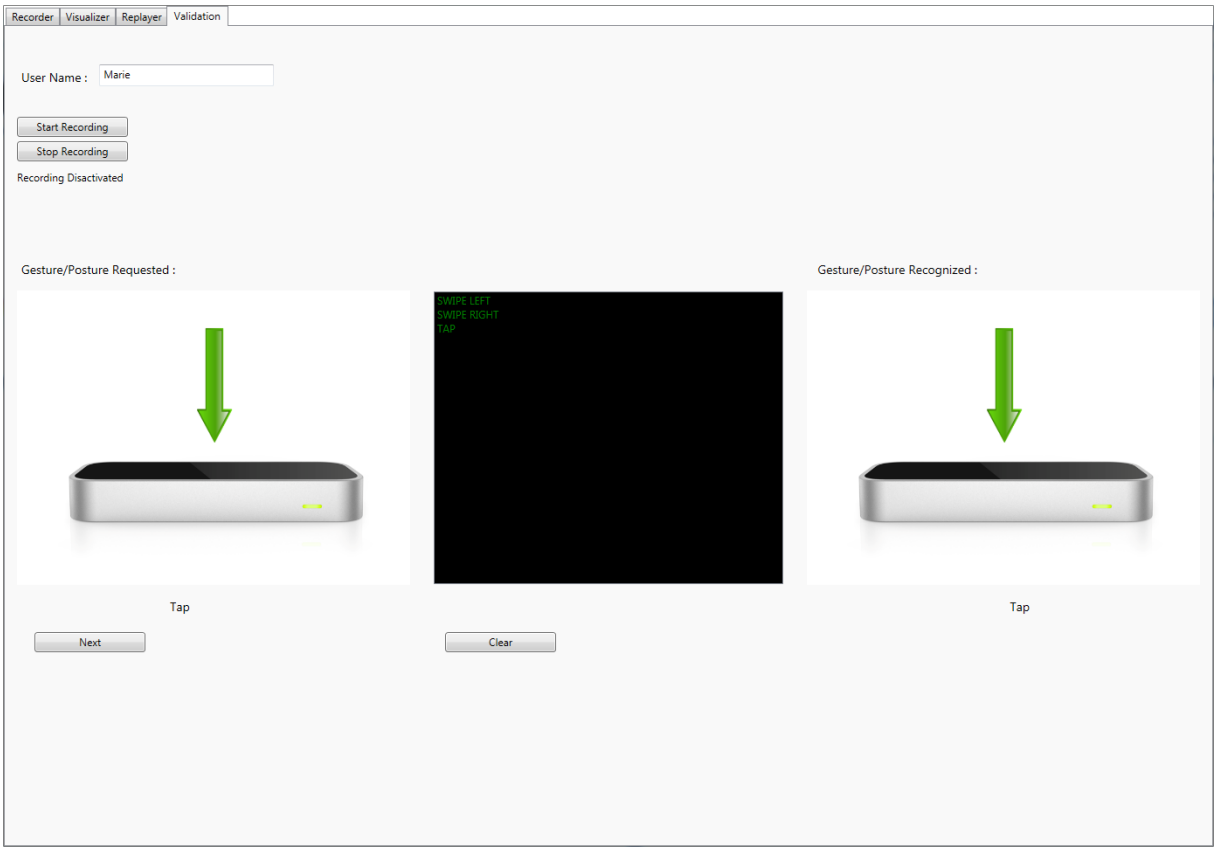

Figure 17 – Partie validation du visualiseur

On peut voir à gauche une image représentant la consigne du geste à effectuer. Sur ces images, on ne voit volontairement pas de doigts pour ne pas biaiser les utilisateurs et les inciter à effectuer les gestes naturellement, avec un doigt ou la main entière. Au milieu se trouve une console où le geste reconnu est affiché, et à droite l'image du geste reconnu.

Je vais maintenant vous expliquer les modifications que j'ai apportées à mes moteurs de détection au vu des résultats recueillis grâce à l'application de tests.

3.4. Amélioration des algorithmes

Grâce au visualiseur 3D, j'ai pu déceler plusieurs problèmes dans mes algorithmes de reconnaissance de gestes et de postures :

- Concernant les gestes, il y avait beaucoup de faux positifs, d'autres gestes étaient reconnus en même temps que celui réellement effectué. Ces faux positifs étaient gênants car ils conduisaient à une détection intempestive, non voulue. Pour résoudre cela, j'ai décidé de mettre en place un système de **buffer**, qui va garder en mémoire les gestes reconnus et que l'on va trier à la fin pour lever l'évènement du geste « prioritaire » (les critères de priorité seront décrits plus bas). De plus, le buffer va permettre de n'envoyer qu'un seul évènement par geste et donc d'éviter les évènements redondants.

- Concernant les postures, j'ai ajouté les évènements « détection en cours », qui possèdent en argument le pourcentage de détection, et celui « détection perdue ». Ainsi, on pourra par exemple lier le pourcentage avec une jauge dans le Composer, et permettre à l'utilisateur de mieux visualiser la détection de postures. Ceci est une bonne pratique ergonomique qui permet à l'utilisateur final de ne pas rester devant un écran vide pendant la détection de la posture.

J'ai aussi pu ajuster les paramètres de détection grâce à la visualisation des gestes pour permettre la détection des mouvements d'un plus ample panel d'utilisateurs.

3.4.1. Reconnaissance de gestes : mise en place d'un buffer de gestes

Là où dans la précédente version de l'algorithme, plusieurs évènements identiques étaient levés lors de l'exécution d'un geste (dès que les conditions étaient vérifiées), la mise en place du buffer demande une notion de début et fin de geste. Pour cela, je garde en mémoire le type du dernier évènement levé ainsi que l'instant où il a été levé (timestamp). Si un nouvel évènement est envoyé et que la différence entre son timestamp et celui du précédent est supérieure au temps entre deux frames, je considère qu'il s'agit du début du geste. Pour envoyer l'évènement de fin de geste, je vérifie qu'un évènement du même type ait été levé à la frame précédente et que la vitesse du mouvement soit inférieure à une vitesse minimum. De plus, si cet évènement n'est pas levé au bout d'un certain temps, un timer s'occupe de « forcer » la fin du geste.

Je vais maintenant vous expliquer le fonctionnement du buffer, que j'ai implémenté dans la classe *Scheduler*. Ce principe requière plusieurs variables :

- Une liste d'évènements : le *buffer*,
- Un dictionnaire qui relie les types de gestes avec un booléen pour savoir si un geste est en cours : le *flag*,
- Un timer qui se lance lorsqu'un geste est détecté et qui cherche la fin de ce geste pour lancer le tri du buffer : le *timesUpTimer* (intervalle de 700 millisecondes),
- Un deuxième timer, lancé si le geste n'est toujours pas terminé lorsque le *timesUpTimer* est écoulé, qui attend la fin du geste et la force si elle tarde à venir : le *delayTimer* (intervalle de 30 millisecondes),
- Un entier qui est un compteur pour le *delayTimer* : le *delayTimerCounter.*

Pour aider à la compréhension, voici un diagramme qui montre le principe des timers, auquel vous pourrez vous référer en lisant les explications :

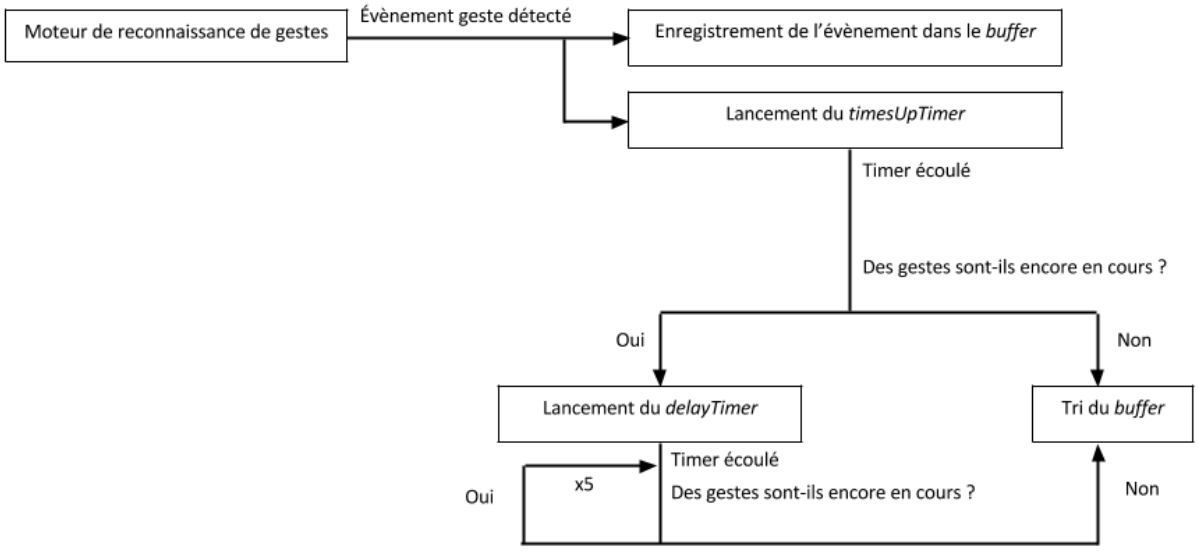

Figure 18 – Diagramme reprenant le principe du buffer

Comme nous avons vu précédemment, le moteur de reconnaissance de gestes envoie maintenant trois types d'évènements :

- Les évènements de début de gestes,
- Les évènements de gestes détectés, qui correspondent aux évènements intermédiaires levés à chaque frame lorsque le geste est en cours,
- Les évènements de fin de gestes.

Lorsque le *Scheduler* reçoit un évènement de début de geste, il se contente de positionner le *flag* correspondant à true. A l'inverse, lorsqu'il reçoit un évènement de fin de geste, il positionne simplement le *flag* à false. Par contre, lorsqu'il reçoit un évènement intermédiaire, il ajoute ce dernier au *buffer*, relance le *timesUpTimer*, et stoppe le *delayTimer*. Dans la plupart des cas, lorsque l'évènement de fin de geste est bien levé, le *delayTimer* n'est pas lancé. Mais imaginons que le geste de l'utilisateur commence par un push parasite, et que le geste voulu commence juste avant les 700 millisecondes du *timesUpTimer*. Sans le *delayTimer*, le buffer serait alors trié directement avec seulement le push. Ce deuxième timer permet de gagner un peu de temps dans le cas où le geste voulu débute juste avant l'écoulement du premier timer. De plus, il permet de forcer la fin d'un geste si l'évènement correspondant n'est pas levé. La valeur de 700 millisecondes pour le *TimesUpTimer* a été choisie après plusieurs tests et constitue le meilleur compromis entre un bon taux de réussite et un temps d'attente acceptable entre le moment où le geste est exécuté et celui où il est reconnu par l'algorithme.

Lorsque le *timesUpTimer* est écoulé, il regarde si un des composants du *flag* est à true. Si c'est le cas, il lance le *delayTimer*, sinon cela veut dire que tous les gestes sont terminés et il se contente de lancer le tri du *buffer*.

Lorsque le *delayTimer* est écoulé, il incrémente son compteur qui va lui permettre de savoir combien de fois il s'est écoulé et regarde si un geste est en cours. Si c'est le cas, il ne fait rien et attend de s'écouler une nouvelle fois. Par contre, si tous les éléments du *flag* sont à false, il lance le tri du

buffer. S'il s'est écoulé cinq fois et qu'un geste est encore en cours, il réinitialise le *flag* à false, forçant ainsi la fin du geste, pour ensuite lancer le tri.

En résumé, voici les deux callbacks des timers appelées lorsqu'ils sont écoulés :

```
Callback OnTimesUpTimerElapsed
Début
	Stopper le timesUpTimer
	Si le flag contient au moins un true alors
		Lancer le delayTimer
	Sinon
		Trier le buffer
	Finsi
Fin

Callback OnDelayTimerElapsed
Début
	Incrémenter le delayTimerCounter
	Si le delayTimerCounter est égal à 5 et le flag contient au moins un
	true alors
		Réinitialiser le flag à false pour tous les gestes
	Finsi
	Si le flag est à false pour tous les gestes alors
		Stopper le delayTimer
		Positionner le delayTimerCounter à 0
		Trier le buffer
	Finsi
Fin
```

Le tri du *buffer* permet de lever l'évènement du geste le plus prioritaire. Pour déterminer le geste prioritaire, je compte le nombre d'occurrences de chaque type d'évènement. Si le tap apparaît au moins une fois, il est prioritaire et c'est lui qui sera levé. Pour les autres gestes, c'est celui qui possède le nombre d'occurrences maximum qui est prioritaire. Si par hasard deux gestes possèdent le plus grand nombre d'occurrences, la priorité est comme suit : swipe left, swipe right, puis push.

La bufferisation des évènements a grandement amélioré le taux de réussite de la reconnaissance et diminué les faux positifs mais elle entraîne un temps de latence entre le fin du geste exécuté par l'utilisateur et la levée de l'évènement.

3.4.2. Reconnaissance de postures : ajout d'évènements détection en cours/détection perdue

Pour mettre en place l'envoi des évènements de détection en cours et détection perdue, le *Scheduler* a besoin que le moteur de reconnaissance de postures lui envoie les évènements « début de posture » et « posture maintenue ». Ce dernier est envoyé à chaque frame tant que l'utilisateur maintient sa posture. Pour cela, j'ai eu besoin d'ajouter au moteur un flag et un timer de 100 millisecondes. Dans l'algorithme de détection, lorsque toutes les conditions sont réunies pour lever un évènement de détection de posture, il n'y a qu'à regarder la valeur du flag correspondant pour

savoir si l'on doit lever l'évènement de début de posture ou de posture maintenue. Lorsqu'on lève l'évènement de début de posture, on positionne le flag à true et on lance le timer, et lorsqu'on lève l'évènement de posture maintenue, on relance le timer. Ainsi, lorsque le timer est écoulé, on peut considérer que la posture a été rompue car cela fait 100 millisecondes que les conditions de posture ne sont plus respectées.

Grâce à ces deux évènements de la part du moteur de reconnaissance, le *Scheduler* peut définir quand envoyer les évènements de détection en cours, détection perdue et détection réussie. Pour cela, il a besoin :

- D'un flag qui va indiquer si la détection d'une posture est en cours (et non pas si la posture est maintenue contrairement au flag du moteur),
- D'un timer qui servira à repérer la perte de détection, le *postureLostTimer* (100 millisecondes),
- D'un compteur pour chaque type de posture. Lorsque ce compteur aura atteint une certaine valeur, définie arbitrairement, on considèrera la posture comme détectée.

Voici un diagramme reprenant le principe pour aider à la compréhension :

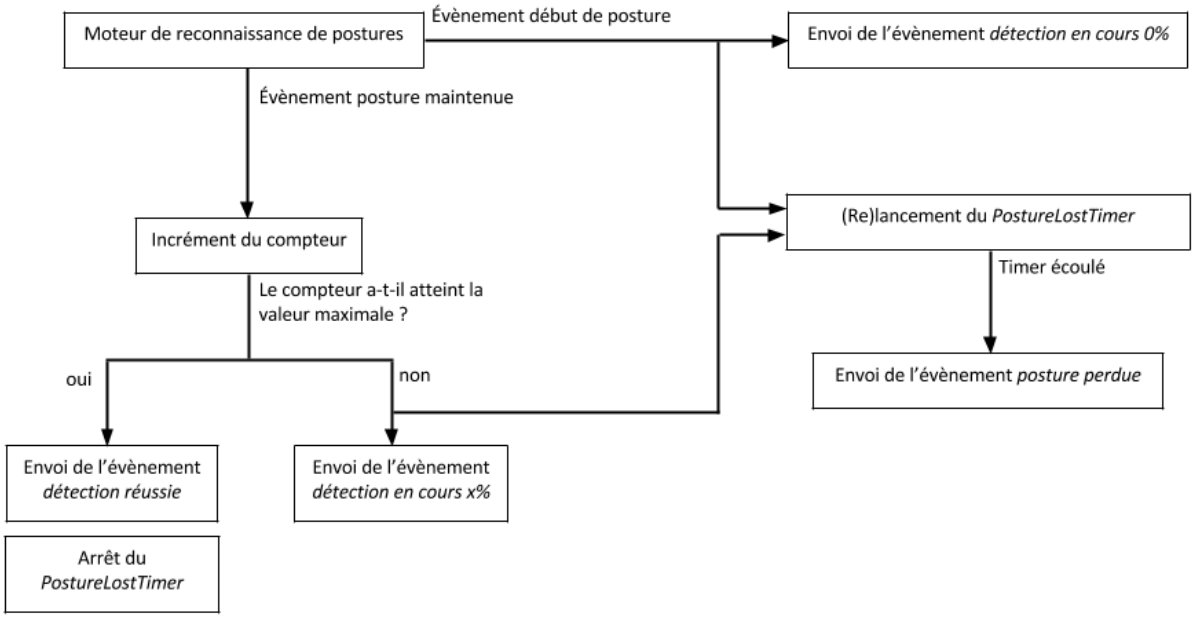

Figure 19 – Diagramme reprenant le principe de la détection de postures

Lorsque le *Scheduler* reçoit l'évènement de début de posture, il positionne son flag et lance le *postureLostTimer* avant de lever le premier évènement de détection en cours avec la valeur 0%. Lorsqu'il reçoit un évènement de posture maintenue, il incrémente le compteur correspondant. Si ce compteur a atteint sa valeur maximale, on considère que la posture est entièrement détectée : le *postureLostTimer* est stoppé et l'évènement de détection réussie peut être levé. Au contraire, si le compteur est inférieur à cette valeur, on se contente de relancer le *postureLostTimer* et lever un évènement « détection en cours » avec le bon pourcentage (donné par ((valeur du compteur * 100) / valeur maximale)). Cette valeur maximale pour le compteur définit en fait le temps durant lequel la posture doit être maintenue afin d'être détectée. Par défaut, ce temps est d'environ une seconde.

Lorsque le *postureLostTimer* est écoulé, on considère que la détection a été perdue, car cela veut dire qu'il n'y a eu aucun évènement « posture maintenue » depuis 100 millisecondes.

Après avoir amélioré les moteurs de reconnaissance de gestes et de postures, j'ai organisé des tests grâce à la partie validation du visualiseur décrite précédemment, en m'aidant aussi des statistiques. J'ai donc pu ajuster les intervalles des timers, les paramètres de vitesse maximale et minimale, etc, en comparant les taux de réussite. Voici ci-dessous une comparaison des taux de réussite pour chaque geste avant et après amélioration de l'algorithme.

	Swipe Left	**Swipe Right**	**Tap**	**Push**
Avant	62,7%	76%	45%	80%
Après	80,4%	83%	76,9%	25,5%

Ces résultats sont un peu faussés, car avant de modifier l'algorithme, je considérais la détection réussie tant qu'au moins ce geste était reconnu, même si il y avait aussi des faux positifs. Tandis que les résultats après modification sont ceux des fichiers où seulement le bon geste est reconnu, sans faux positifs. On peut donc voir que même en ayant compté les fichiers avec faux positifs avant, les résultats sont bien meilleurs après, excepté pour le push. De plus, ce geste était la cause de nombreux échec de reconnaissance sur les trois autres gestes. En effet, il induisait beaucoup de faux positifs. Par exemple, il était souvent reconnu à tort lorsque la main de l'utilisateur entrait dans le champ de vision du Leap avant d'effectuer un geste ou une posture volontaire. Face à ce problème ainsi qu'à son taux de réussite médiocre après l'amélioration de l'algorithme, nous avons décidé de l'éliminer de la liste des gestes qui sortiront dans le Composer.

Lorsque les algorithmes ont été validés, je suis passé à l'intégration de la DLL dans le Composer, que je vais maintenant vous détailler.

3.5. Intégration au Composer

L'intégration au Composer nécessite plusieurs étapes pour lier la DLL au logiciel. Tout d'abord, il faut choisir quels éléments seront exposés au Composer à l'aide du Design Pattern de **Façades**, puis faire le lien grâce à un **descripteur** JSON et pour finir, définir un visuel pour les interface assets sous la forme de **design accelerators**.

3.5.1. La notion de Façade

Une Façade est un patron de conception qui permet de cacher la complexité d'un système en exposant seulement les fonctionnalités nécessaires aux utilisateurs finaux. Ainsi, les Façades vont

permettre de choisir les propriétés et évènements remontés au Composer, et de découper la DLL en plusieurs interface assets. Chaque interface asset possède sa Façade.

Nous avons donc décidé de découper le plugin Leap Motion en quatre interface assets, donc quatre Façades, chacun correspondant à un cas d'usage du capteur :

- Hand presence : on peut recueillir des évènements lorsqu'une main entre ou sort du champ de vision du Leap. Je n'ai pas détaillé plus haut l'algorithme qui détecte la présence d'une main car cela revient seulement à tester si la variable *Hands* renvoyée par le Leap est nulle.
- Gestures : contient les triggers pour le swipe left, le swipe right et le tap ainsi que les propriétés qui permettent d'activer/désactiver la reconnaissance d'un geste.
- Rock-Paper-Scissors Postures : contient les triggers pour les postures rock, paper et scissors (détection en cours, détection réussie et détection perdue) ainsi que les propriétés d'activation/désactivation de la détection.
- Finger Count Postures : idem que pour l'interface asset Rock-Paper-Scissors Postures mais avec les postures de comptage de doigts.

Chaque Façade propage les évènements qui lui correspondent et possède des propriétés pour chaque geste/posture pour activer et désactiver la détection. Cela permet de ne pas recevoir les évènements liés à un geste particulier inutilement si celui-ci n'est pas utilisé dans la présentation, et ainsi alléger la charge envoyée au Player. Ces propriétés sont des booléens qui, lorsqu'ils changent de valeur, vont s'abonner ou se désabonner en chaîne aux évènements des couches inférieures et positionner des variables internes.

Les Façades implémentent le patron de conception **singleton**. Ce patron assure qu'une classe ne pourra être instanciée qu'une unique fois. Cela évite que plusieurs évènements identiques soient levés pour un seul geste effectué, car sinon le Player exécuterait autant de fois l'action associée à ce geste. Ainsi, on ne pourra avoir dans le Composer qu'une seule instance de chaque interface asset. Si l'utilisateur créé une deuxième instance, cette dernière n'est pas viable.

Chaque Façade doit capter les évènements d'une même et unique instance du *LeapListener*. Pour cela, j'ai rajouté une classe intermédiaire qui implémente elle aussi le patron de conception singleton et qui possède l'unique instance du *LeapListener*. Cette classe s'appelle *LeapPlugin*. Ainsi, les Façades s'abonnent aux évènements du *LeapListener* de ce *LeapPlugin*. Lorsqu'une Façade est instanciée, elle regarde si l'instance du *LeapPlugin* est nulle et seulement dans ce cas, elle la créé.

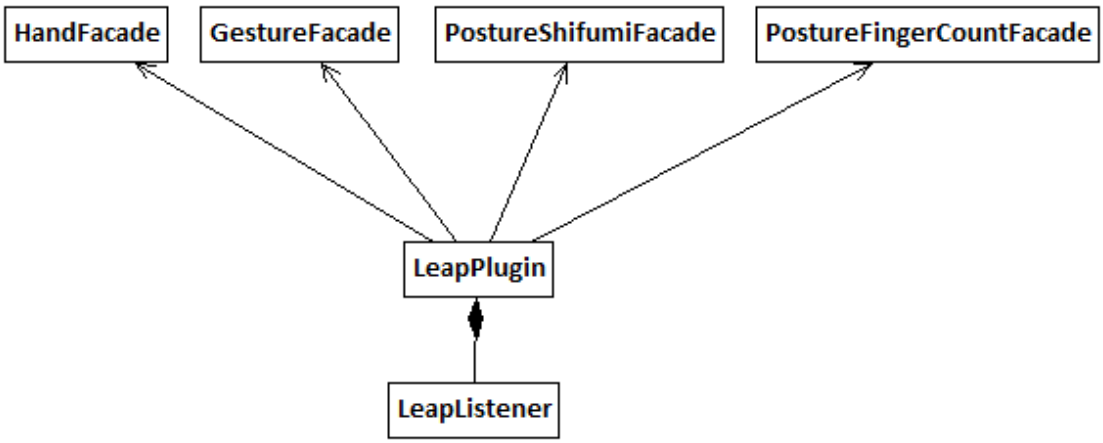

Figure 20 – Le système de Façades

Pour notifier les changements de valeur des propriétés des Façades au Composer, ces dernières doivent implémenter l'interface .NET **INotifyPropertyChanged**. Ainsi, à chaque fois qu'une propriété change de valeur, un évènement « property changed » doit être levé. Le Composer/Player capte cet évènement et réévalue la propriété pour refléter le changement dans son interface graphique. Si une Façade n'implémente pas l'interface INotifyPropertyChanged, le Composer va réévaluer l'ensemble de ses propriétés dès qu'une méthode est appliquée sur son instance. Cela peut poser problème si une propriété est changée par l'instance d'une autre classe (le changement passe inaperçu) ou si la Façade possède beaucoup de propriété (baisse de performance).

3.5.2. Le descripteur JSON

Pour faire le lien avec les Façades, le Composer a besoin d'un fichier JSON appelé **descripteur** basé sur la syntaxe Google Discovery Service, qui va définir les propriétés, méthodes et évènements que l'on veut exposer. Ce fichier contient aussi tout ce qu'il faut au Composer pour utiliser la DLL, tel que ces dépendances. C'est dans le descripteur que l'on va pouvoir écrire les titres et messages qui seront affichés dans l'interface graphique du Composer pour présenter les interface assets, les propriétés et les triggers à l'utilisateur. La phase de choix des titres et descriptions est appelé le **wording** et est menée conjointement avec les ergonomes et le service marketing. Vous trouverez en annexe une version simplifiée du descripteur (annexe D). Dans cette version, il y a seulement la description de la *GestureFacade*. On peut voir quand dans la partie « schemas », on expose les propriétés, et dans la partie « resources » les méthodes et évènements.

3.5.3. Les design accelerators

La dernière étape de l'intégration au Composer consiste à définir des graphismes par défaut aux interface assets. Ainsi, lorsque l'utilisateur glissera un interface asset dans sa présentation, il aura

déjà un exemple d'utilisation, et pourra de suite tester ses fonctionnalités. Ces visuels sont appelés des **design accelerators**. Ce sont des bouts de code XML qui sont copiés dans l'ifx de la présentation lorsqu'on glisse un asset à l'intérieur. Vous pourrez trouver dans la bibliographie un lien vers le support IntuiLab sur comment créer un design accelerator pour un interface asset. Vous pourrez y voir un exemple de design accelerator.

Ces graphismes par défaut doivent respecter la charte graphique du Composer. Pour cela, un infographiste est venu pour nous aiguiller et créer des images représentant les gestes et postures.

Je vais maintenant vous montrer quelques images des interface assets Leap intégrés au Composer, avec entre autres leurs design accelerators. Ces captures d'écran constituent le résultat final de la partie reconnaissance de gestes et postures, c'est-à-dire qu'elles reflètent le produit tel que livré aux clients d'IntuiLab dans la version 4.1 d'IntuiFace.

Tout d'abord, voici le panneau pour ajouter un interface asset, où l'on peut voir la decription de chaque interface :

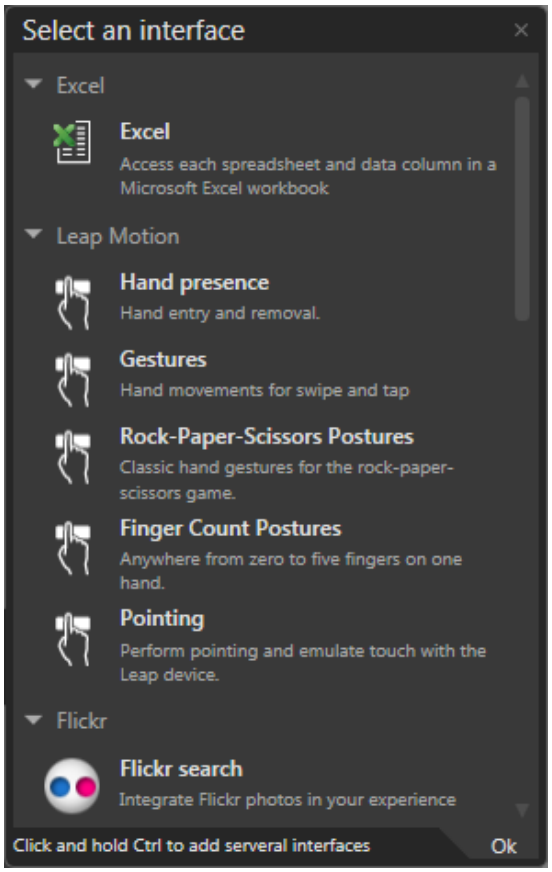

Figure 21 – Panneau d'ajout des interface assets

Ici, vous pouvez voir le panneau des propriétés de l'interface asset Gestures, où l'on peut activer et désactiver un type de geste :

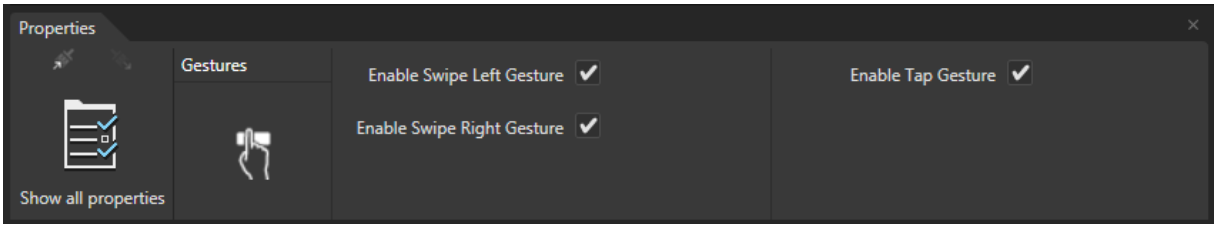

Figure 22 – Panneau des propriétés de l'interface asset Gestures

Je vais maintenant vous montrer le panneau des triggers de l'interface asset Finger Count Postures, où l'on peut lier des évènements à des actions, comme changer la visibilité d'un asset ou encore le contenu d'une image :

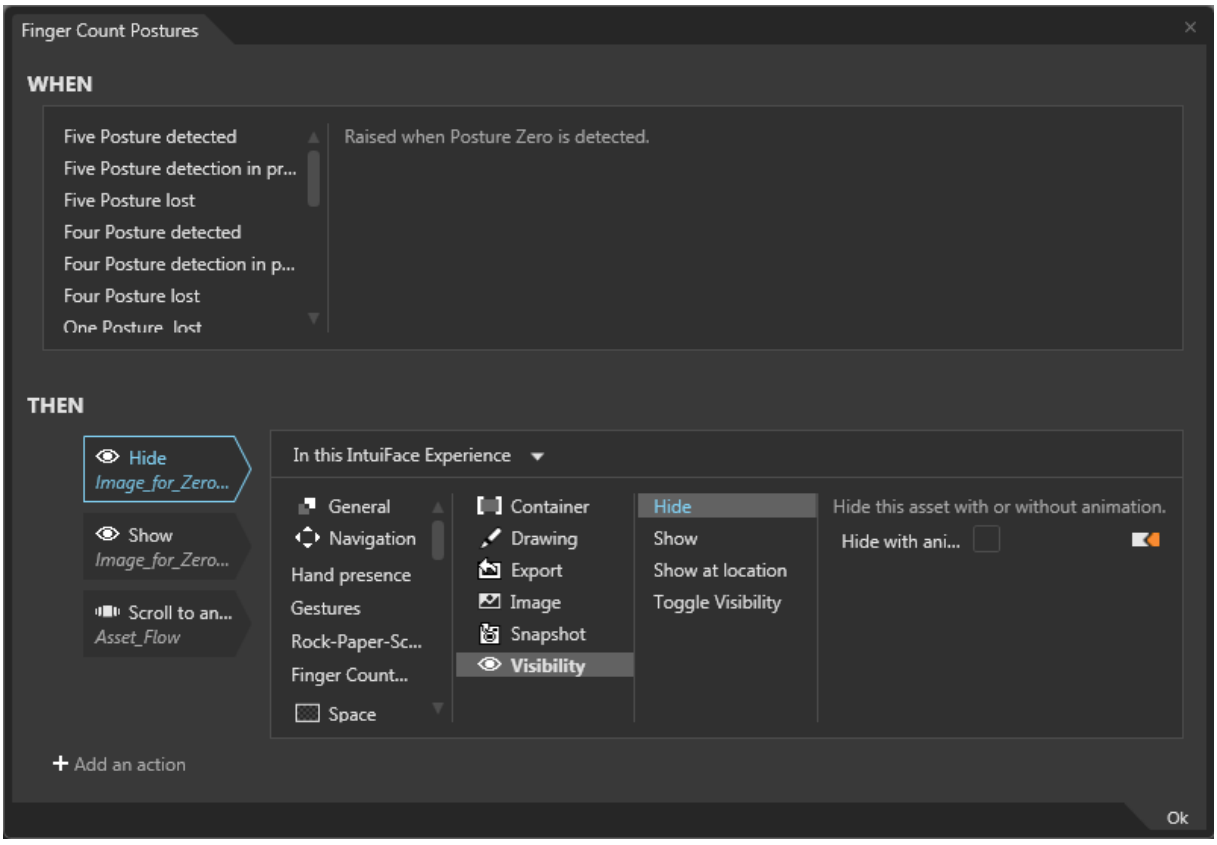

Figure 23 – Panneau des triggers de l'interface asset Finger Count Postures

A présent, voici les design accelerators de chaque interface assets. Je vais commencer par celui du Hand presence :

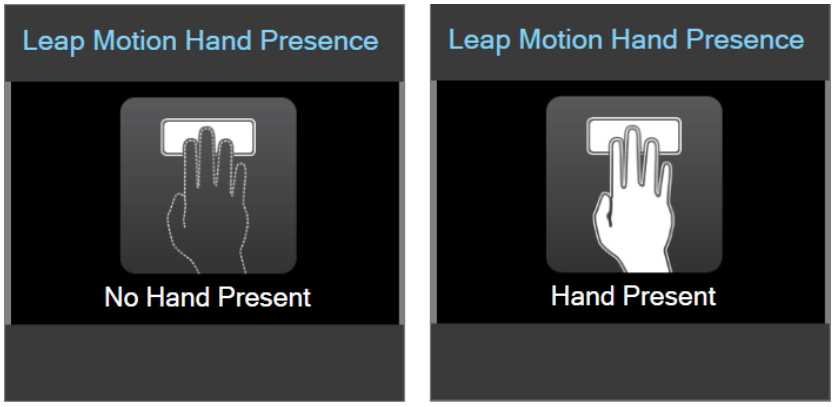

Figure 24 – Design accelerator de l'interface asset Hand presence

Voici maintenant celui des gestes :

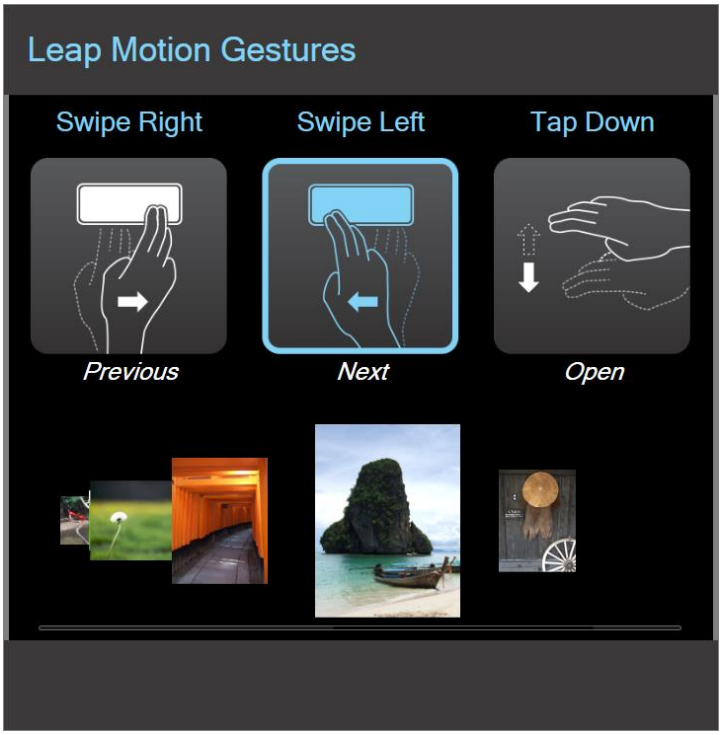

Figure 25 – Design accelerator de l'interface asset Gestures

On peut voir en bas une collection d'images, appelée dans le Composer un asset flow. Nous avons décidé de lier les gestes à la navigation dans cet asset flow pour donner un exemple d'utilisation. Sur cette capture d'écran, un geste swipe left vient d'être détecté, ce qui souligne l'image de sa consigne et fait passer l'asset flow à l'image suivante.

Ci-dessous, le design accelerator de l'interface asset Rock-Paper-Scissors Postures :

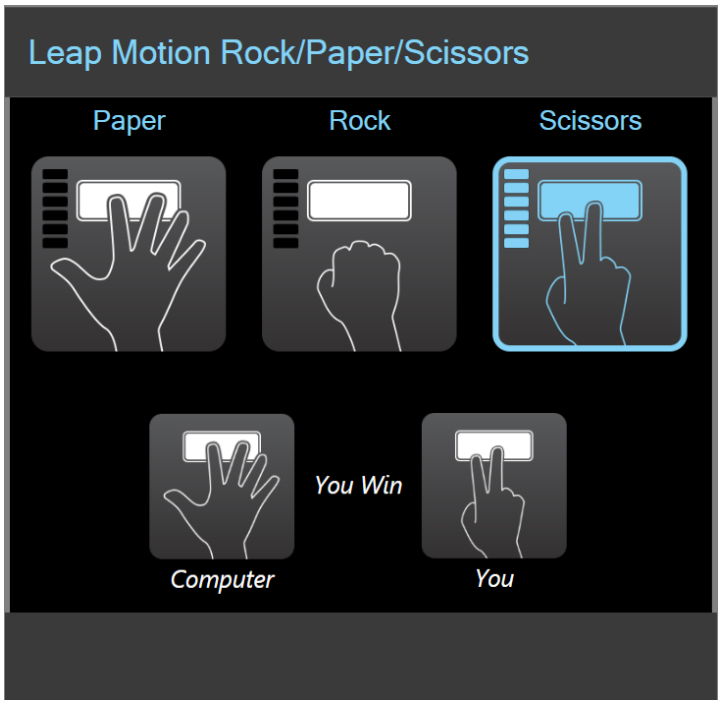

Figure 26 – Design accelerator de l'interface asset Rock-Paper-Scissors Postures

Pour cette interface, nous avons décidé de reprendre le jeu du pierre-feuille-ciseaux. On peut voir que les postures sont liées à des jauges qui indiquent le pourcentage de détection. Lorsque la jauge est pleine, la posture est détectée, comme sur la capture d'écran.

Pour finir, voici le design accelerator de l'interface asset Finger Count Postures :

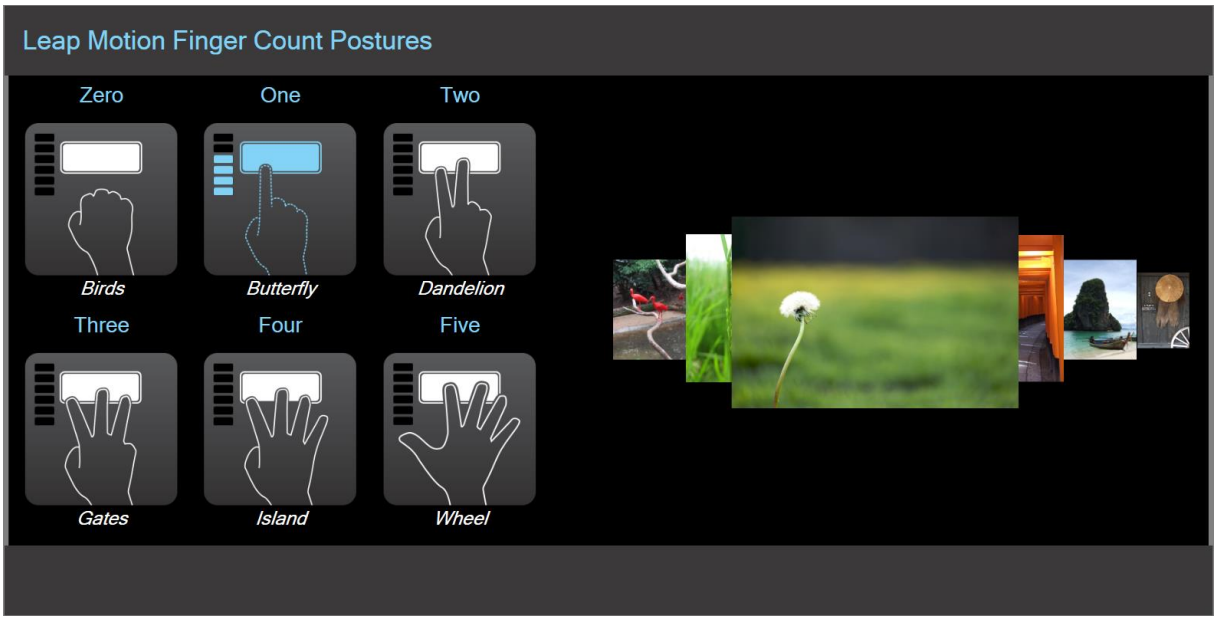

Figure 27 – Design accelerator de l'interface asset Finger Count Postures

Ici, nous avons lié les postures à l'index de l'image à l'avant dans l'asset flow. Ainsi, si on exécute par exemple la posture Four, l'asset flow se déplacera jusqu'à positionner la quatrième image à l'avant. Sur l'image, on peut voir que la détection d'un One est en cours, avec la jauge qui se remplit.

La version 4.1 d'IntuiFace comprenant ces interface assets est sortie le 1er août 2013.

Durant la période restante, j'ai commencé à mettre en place un interface asset de pointage et manipulation d'assets avec le Leap. Dans le chapitre suivant, je vais vous présenter mon travail concernant ce sujet.

4. Le travail effectué sur le pointage et la manipulation directe

4.1. Travail préliminaire

Le **pointage** revient à se servir de ses doigts comme curseurs par l'intermédiaire du Leap, et ainsi simuler des touchs et **manipuler** des éléments dans des présentations. Il y a donc deux parties dans l'implémentation du pointage : il faut afficher des retours visuels sur les positions des doigts, appelés feedbacks, c'est-à-dire des curseurs qui permettent à l'utilisateur de voir où se situent ses doigts à l'écran, et permettre la manipulation des assets dans la présentation.

Dans un premier temps, j'ai réalisé en collaboration avec le deuxième stagiaire un **état de l'art** sur la manipulation directe dans les IHM par le biais d'interactions à distance. Ce document comporte surtout des conseils et des bonnes pratiques à avoir dans la conception d'interface de manipulation par capteurs distants, mais présente aussi plusieurs méthodes pour simuler des touchs. Vous pourrez le retrouver en annexe (annexe E).

Après avoir réfléchi avec les ergonomes et mon maître de stage, voici les concepts retenus pour la mise en place du pointage :

- Le pointage et les gestes ne peuvent pas être actifs en même temps. En effet, le pointage interfère beaucoup trop dans la reconnaissance des gestes, le risque étant d'avoir des détections de gestes intempestives. Par contre, la reconnaissance des postures pourra être activée en même temps que le pointage.
- Dans le cas du Leap, si un interface asset de gestes et celui de pointage sont utilisés en même temps dans une présentation, c'est le pointage qui restera actif par défaut et la détection des gestes sera désactivée, ce dispositif se prêtant plus naturellement au premier qu'au second (à l'inverse de Kinect).
- On pourra passer d'un mode à l'autre (gestes/pointage) par une action configurable, comme par exemple une posture.
- Le pointage se fera avec tous les doigts, et il faut que les dix curseurs correspondant aux doigts apparaissent si l'utilisateur les a tous dans le champ de vision du Leap.
- La simulation des touchs se fera par la méthode du **z-index** ou mur virtuel : il faut que le doigt dépasse un plan virtuel en z pour effectuer un touch. Tant qu'il est derrière ce plan, le doigt reste en touch et il doit le dépasser dans l'autre sens pour désactiver son touch. La valeur du z à dépasser devra être configurable dans le Composer.
- La taille de la zone d'interaction doit pouvoir être configurable depuis le Composer.
- La manipulation s'appuiera sur le protocole **TUIO** qui est déjà implémenté dans le Composer et que je vais maintenant vous détailler.

IntuiLab a développé et breveté un module embarqué dans IntuiFace appelé le **Multitouch Gestures Recognition Engine (MGRE)**. Ce module gère toutes les interactions tactiles faites sur une présentation et applique des transformations géométriques aux objets si nécessaire, le tout de manière continue.

Le MGRE prend en entrée les positions des doigts sur l'écran, leur vitesse et accélération. Ces informations sont transmises du périphérique vers le MGRE par un protocole appelé **TUIO**. La plupart des périphériques multitouch utilisant ce protocole, le MGRE permet aux applications développées par IntuiLab de fonctionner sur ces derniers sans modification. Nous avons donc décidé d'utiliser le MGRE pour la manipulation avec Leap. Ainsi, lorsqu'un doigt passe le z-index du mur virtuel, il est considéré comme en touch : ses positions seront envoyées au MGRE grâce à TUIO et il sera traité comme un évènement tactile. C'est donc le lui qui s'occupera de toutes la physique de la manipulation et je n'aurai qu'à lui envoyer les données pour que le comportement soit identique au tactile.

TUIO est un framework open source qui définit un protocole et une API pour les surfaces multitouch tangibles. Il permet la transmission d'une description abstraite de surfaces interactives, incluant les évènements de touch et les états d'objet tangible. Le périphérique qui capte les positions des doigts est un serveur TUIO qui envoie des trames contenant ces données par un port précis et l'application qui affiche la présentation tactile est le client TUIO qui reçoit ces trames. Voici un exemple d'utilisation du protocole TUIO :

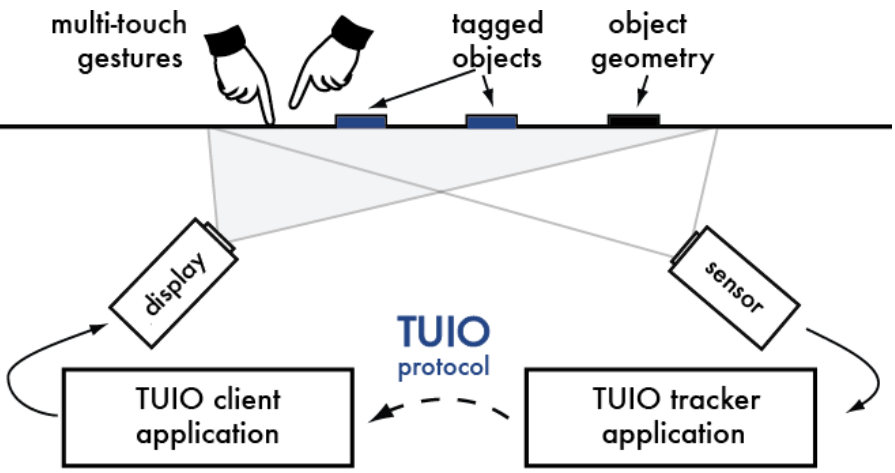

Figure 28 – Le protocole TUIO

Le MGRE est donc un client TUIO, et le plugin Leap devra implémenter une partie serveur TUIO. Une trame TUIO typique est constituée d'un message ALIVE qui identifie les IDs des doigts présents sur la surface, un ou plusieurs messages SET utilisés pour envoyer les données d'un doigt et un message FSEQ qui envoie l'ID de la trame. Elle a la structure suivante :

```
/tuio/2Dcur alive s_id0 ... s_idN
/tuio/2Dcur set s_id x_pos y_pos x_vel y_vel m_accel
/tuio/2Dcur fseq f_id
```

Où `s_id0 ... s_idN` représentent les IDs des doigts présents, `x_pos` et `y_pos` sont la position du doigt, `x_vel` et `y_vel` sa vitesse et `m_accel` son accélération. `f_id` représente l'ID de la trame. Les positions doivent être normalisées entre 0 et 1 avant d'être envoyées via TUIO, et c'est le MGRE qui s'occupera de les mettre à l'échelle de la présentation.

Je vais maintenant vous détailler la première version de mon algorithme de pointage qui prend en compte les spécifications citées plus haut.

4.2. Mise en place d'un premier algorithme de pointage

Cet algorithme devra lever cinq types d'évènements :

- Un évènement lorsqu'un doigt apparaît dans le champ de vision du Leap, qui devra contenir l'ID du doigt et les positions x, y et z de son extrémité.
- Un évènement lorsqu'un doigt qui est déjà dans le champ de vision bouge, avec les même paramètres plus un booléen qui indique si le doigt est en touch. C'est ce type d'évènement qui permettra de rafraichir la position du doigt sur l'écran. En résumé, cet évènement sera lancé à chaque frame pour les doigts présents dans le champ.
- Un évènement lorsqu'un doigt quitte le champ de vision du Leap,
- Un pour notifier qu'un doigt vient de dépasser le plan virtuel en z et est donc en touch (touch down),
- A l'inverse, un pour notifier qu'un doigt n'est plus en touch (touch up).

Pour cela, je récupère la liste des doigts présents dans la frame courante : *fingers*. Pour savoir si un doigt vient juste d'apparaître ou non, je vais utiliser une liste où je vais garder en mémoire les IDs des doigts présents à la frame précédente : *lastFrameFingers*.

Tout d'abord, je regarde si des doigts ont disparus : pour chaque ID présent dans cette liste, je vais voir si un doigt avec l'ID correspondant est présent dans *fingers*. Si ce n'est pas le cas, je lève l'évènement « doigt retiré » pour ce doigt.

Puis, pour chaque doigt dans *fingers*, je vais calculer ses coordonnées x et y normalisées. Pour cela, je défini une zone d'interaction qui est un plan en x et y et dont les bornes sont arbitraires. Ces bornes seront remontées au Composer pour être configurables. Si le doigt est dans la zone, ses coordonnées seront comprises entre 0 et 1 et il apparaitra à l'écran. Sinon, je borne ses coordonnées à 0 ou 1, comme ça le curseur ne pourra pas sortir de l'écran.

Ensuite, je regarde si le doigt est présent dans la liste *lastFrameFingers*. Si c'est le cas, je lève un évènement « doigt en mouvement », sinon je lève l'évènement « un doigt apparaît ». Pour finir, je teste si la coordonnée z du doigt est inférieure ou supérieure à la limite du touch et si le doigt est déjà en touch ou pas pour savoir si je lève un des évènements touch down ou touch up.

Les coordonnées x et y que je remonte dans les évènements sont comprises entre 0 et 1 selon la zone d'interaction choisie. Pour la coordonnée z, je renvoie celle dans le repère du Leap translaté en z pour que le z qui définit le touch soit l'origine de l'axe.

Concernant le protocole TUIO, le serveur garde en mémoire une liste des curseurs présents avec leurs positions. Un thread à part va se charger d'envoyer une trame à intervalle de temps régulier, en se basant sur cette liste pour remplir les messages. Du côté de l'algorithme de pointage, il suffit donc d'ajouter un curseur à la liste lorsqu'un doigt exécute un touch down, de mettre à jour sa position lorsqu'il bouge alors qu'il est en touch puis de le retirer lorsqu'il exécute un touch up.

Pour une meilleure compréhension, voici le principe de l'algorithme de pointage :

```
Algorithme Pointage (frame courante)
Début
	Récupérer la liste des doigts présents dans la frame courante : fingers
	Pour chaque ID dans lastFrameFingers faire
		Si fingers ne contient pas de doigt ayant le même ID alors
			Lever l'évènement « doigt retiré » pour ce doigt
		Finsi
	Finpour
	Pour chaque doigt dans fingers faire
		Calculer les coordonnées x et y normalisées
		Si le doigt n'est pas dans lastFrameFingers alors
			Lever l'évènement « un doigt est apparu » pour ce doigt
		Sinon
			Lever l'évènement « doigt en mouvement » pour ce doigt
			Si le doigt est en touch alors
				Mettre à jour la position du curseur correspondant
				à ce doigt dans le serveur TUIO
			Finsi
		Finsi
		Si le z de ce doigt est supérieur à la limite du touch et que
		le doigt n'est pas en touch alors
			Lever l'évènement « touch down » pour ce doigt
			Ajouter un curseur pour ce doigt au serveur TUIO
			Positionner touching à true pour ce doigt
		Sinon si le z de ce doigt est inférieur à la limite du touch et
		que le doigt est en touch alors
			Lever l'évènement « touch up » pour ce doigt
			Enlever le curseur correspondant à ce doigt dans le
			serveur TUIO
			Positionner touching à false pour ce doigt
		Finsi
	Finpour
Fin
```

Avant d'intégrer le pointage et la manipulation au Composer, un dernier principe doit être mis en place : celui de la **compression d'évènements**. En effet, le frame rate du Leap étant très élevé (200 Hz), le Composer ne sait pas traiter autant d'évènements à la fois, surtout lorsque beaucoup de doigts sont présents (à chaque frame, si 5 doigts sont visibles, c'est 5 évènement levés pour rafraichir la position, à multiplier par le frame rate du Leap d'environ 200 frames/s = 1000 évènements par seconde, c'est le crash assuré !). Pour cela, il faut envoyer les évènements « doigt en mouvement » à

une certaine fréquence. Les quatre autres types d'évènements sont levés normalement, car ils n'interviennent pas à chaque frame. La fréquence retenue est de 35Hz.

Pour implémenter ce principe, j'ai utilisé un timer qui possède un intervalle de temps de (1000 / fréquence) millisecondes, et qui s'écoule donc 35 fois par seconde. Au lieu de lever directement l'évènement « doigt en mouvement » dans l'algorithme, je le stocke dans un buffer, soit en l'ajoutant si aucun évènement avec l'ID de ce doigt n'est déjà stocké, soit en le mettant à jour si le buffer contient déjà un évènement en attente pour ce doigt. Puis c'est dans la callback appelée lorsque le timer s'écoule que je lève les évènements présents dans le buffer. Si un doigt est retiré, je prends soin d'enlever l'évènement correspondant dans le buffer, pour ne pas le voir réapparaître au prochain écoulement du timer. Les évènements qui devraient être levés entre deux écoulements du timer sont tout simplement supprimés.

Maintenant que ce premier algorithme de pointage et manipulation est mis en place, je vais vous expliquer comment je l'ai intégré au Composer.

4.3. Intégration au Composer

Tout comme les interface assets précédents, celui-ci a besoin d'une Façade pour être exposé dans le Composer. Cette Façade possède une propriété pour activer ou désactiver le pointage, ainsi qu'une méthode qui inverse cette propriété, pour pouvoir passer du mode pointage au mode gestes par une action comme une posture. C'est la Façade qui s'occupe de ramener les coordonnées x et y renvoyées par l'algorithme à la taille de la présentation IntuiFace. Pour cela, j'utilise un web service qui questionne le Player sur la taille de la présentation qu'il est en train de jouer, puis je n'ai plus qu'à multiplier les coordonnées normalisées par ces valeurs. La Façade expose aussi la limite en z pour le touch ainsi que les bornes du plan d'interaction, pour les rendre configurables dans le Composer.

J'ai ensuite ajouté cette Façade dans le descripteur, puis il a fallu trouver un moyen d'afficher les feedbacks pour les doigts, en passant par un design accelerator. Pour qu'un curseur suive le doigt auquel il correspond, il a fallu lier l'ID du doigt à celui du curseur. Pour cela, j'ai utilisé la notion de trigger conditionnel : à chaque évènement (trigger) reçu, je teste pour chaque curseur si son ID est égal à celui du doigt (condition), lorsque c'est le cas je lie les coordonnées x et y du curseur à celles de l'évènement pour déplacer le curseur. Les dix curseurs sont tous présents dans la présentation mais si les doigts auxquels ils correspondent ne sont pas présents, ils sont invisibles. Comme les IDs des doigts donnés par le Leap sont tirés au hasard, j'ai dû les ramener entre 1 et 10 pour qu'ils soient égaux à ceux des curseurs. Cette approche est nécessaire pour déplacer le bon curseur, mais les ifx conditionnels sont très lourds pour le Player. Ainsi, on ne peut pas créer de trop grosses présentations, auquel cas le Player n'arriverait pas à les lire correctement.

Pour bénéficier de la notion de profondeur qu'offre le Leap, les curseurs sont en fait des jauges en forme de cercle dont le remplissage est lié à la coordonnée z du doigt. Ainsi, plus le doigt s'approche du plan de touch, plus la jauge se remplie. Lorsque la jauge est pleine, le doigt est en touch.

Voici des captures d'écran pour illustrer le pointage et la manipulation dans IntuiFace :

Figure 29 – Les curseurs pour le pointage

Figure 30 – Un exemple de manipulation avec le Leap : le zoom avec deux doigts sur une image

4.4. Premiers tests

Pour effectuer les premiers tests sur le pointage et la manipulation avec le Leap, j'ai imaginé deux petits jeux chronométrés : dans le premier, il faut appuyer sur des boutons avant qu'ils ne disparaissent et dans le second, il y a au milieu de l'écran un tas de formes rouges et bleues superposées et il faut mettre toutes les formes bleues d'un côté et toutes les rouges de l'autre. Les testeurs sont invités à exécuter ces deux jeux, avant de passer à une présentation où ils vont pouvoir s'entraîner à manipuler des images, un asset flow, et passer du mode pointage au mode gestes en effectuant la posture Five. Ils sont ensuite invités à faire part de leurs impressions avant de refaire les

deux jeux pour voir si leurs temps s'améliorent après s'être entrainé sur la présentation précédente. Ceci permet de voir si après un petit temps d'apprentissage, les utilisateurs arrivent à mieux prendre en main le pointage, ainsi que de comparer les résultats des utilisateurs novices et de ceux habitués au capteur. Ces tests permettent aussi de recueillir les observations des testeurs afin d'améliorer le pointage et la manipulation directe, ainsi que la façon de simuler les touchs.

Voici des captures d'écran des jeux :

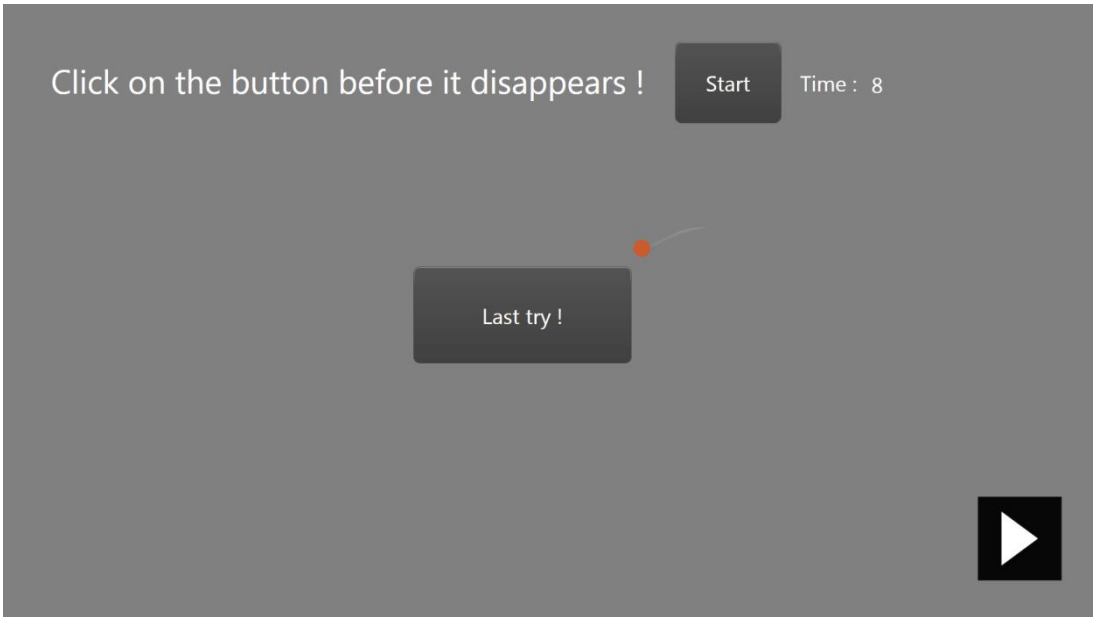

Figure 31 – Premier jeu : cliquer sur les boutons avant qu'ils ne disparaissent

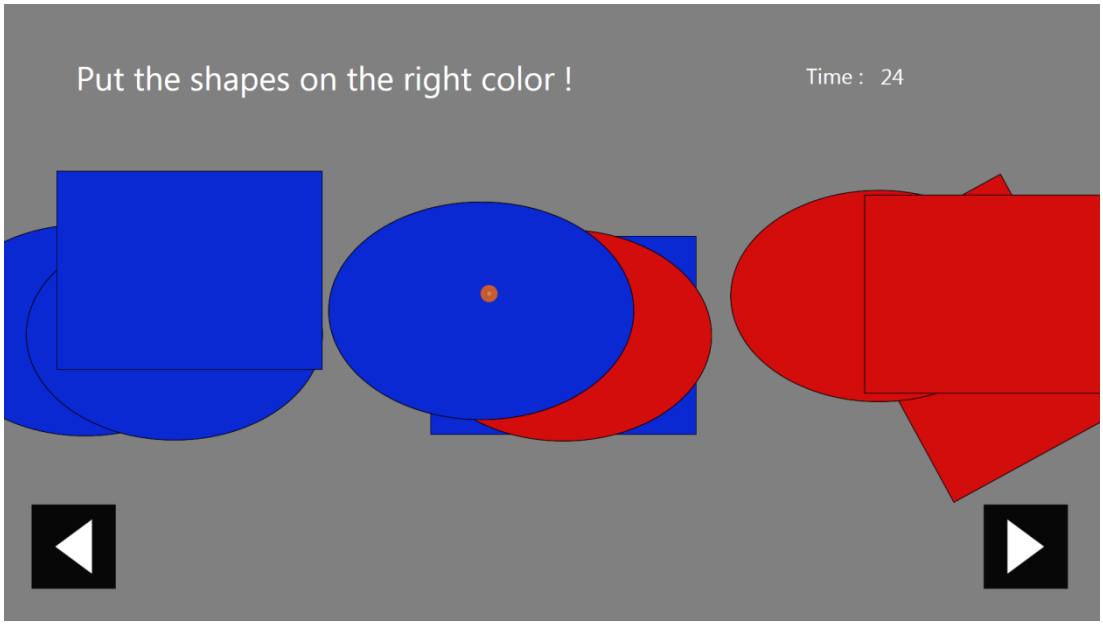

Figure 32 – Deuxième jeu : déplacer les formes du bon côté

A la suite de ces tests réalisés sur une dizaine de personnes, j'ai pu constater que la méthode du z-index pour simuler les touchs n'est pas pertinente avec le Leap. En effet, le Leap est un périphérique très précis (peut-être même trop !), ce qui fait qu'il est dur de garder la position en x et y tout en

avançant en z. Le testeur se positionne sur un bouton, mais lorsqu'il avance le doigt pour faire un touch il perd sa position et se retrouve à cliquer en dehors du bouton. Pour réussir, il faut se concentrer ou s'entraîner et cela n'est plus naturel.

De plus, les doigts bougent imperceptiblement et ce tremblement se répercute à l'écran. Il faudrait stabiliser les positions renvoyées par l'algorithme pour avoir un pointage moins sensible et plus précis.

La dernière observation concerne une sensation d'inconfort à devoir garder la main au-dessus du capteur, mais ceci est lié à l'utilisation du Leap en lui-même et aucune solution n'est possible.

J'ai maintenant terminé de vous exposer le travail que j'ai réalisé durant mes six mois de stage. Nous allons donc voir quelles sont les améliorations futures à apporter au projet.

5. Ce qu'il reste à faire

5.1. Sur la détection de gestes et postures

Tout d'abord, la première étape serait de limiter l'attente entre le moment où le geste est effectué par l'utilisateur et le moment où il est effectivement détecté. Cette attente est induite par l'utilisation du buffer de gestes, plus précisément par l'utilisation du *TimesUpTimer* de 700 millisecondes. Le problème est qu'en diminuant l'intervalle de ce timer, le taux de réussite de la détection est aussi diminué. La valeur de 700 millisecondes a été adoptée d'après les tests et constitue le meilleur compromis entre un bon taux de réussite et un temps d'attente acceptable. Pour pallier ce problème, il faudrait réfléchir à une autre façon de limiter les faux positifs et de bufferiser les gestes.

Dans un deuxième temps, on pourrait réfléchir à de nouveaux gestes et de nouvelles postures. On pourrait par exemple implémenter le cercle mais il faudrait mettre en place un tout nouvel algorithme étant donné que ce mouvement n'est pas linéaire.

5.2. Sur le pointage

Concernant le pointage, il faudrait en premier lieu stabiliser les positions des doigts, en utilisant un filtre de stabilisation sur les coordonnées, comme par exemple un filtre moyenneur.

De plus, il faudrait trouver une autre façon de simuler les touchs, à cause des problèmes cités plus haut. Une des idées proposées par les testeurs est la suivante : éliminer le plan de touch commun à tous les doigts et considérer un doigt comme ayant l'intention de faire un touch lorsqu'il effectue en z une distance delta à une certaine vitesse. Ainsi, lorsqu'on décèle une intention de touch, on pourrait verrouiller les coordonnées x et y jusqu'à ce que le doigt est assez avancé pour lever l'évènement de touch. Ceci éviterait le problème de la stabilité lorsque l'utilisateur vise le point qu'il veut toucher. Une fois le doigt en touch, on déverrouillerait les coordonnées x et y pour permettre la manipulation. Si finalement le doigt ne passe pas en touch, on déverrouillerait les coordonnées au bout d'un certain laps de temps.

Enfin, lorsque le module sera satisfaisant, il faudra finaliser son intégration au Composer en créant de meilleurs graphismes pour les curseurs, en faisant du wording et plus de tests avant de pouvoir sortir une version au client.

6. Les difficultés rencontrées

Tout d'abord, le Leap étant un périphérique très récent, la communauté de développeur Leap est très restreinte et il n'y avait quasiment pas d'exemple de code sur internet, peu d'explications et peu de personnes pour répondre sur les forums. Il m'a fallu un peu de temps pour m'adapter au SDK.

Ensuite, une des difficultés fut de gérer les différences entre les gestes des utilisateurs. Pour produire un algorithme qui détecte les gestes du plus grand nombre de personnes, il a fallu faire des tests avec un large panel de gens : certains qui connaissaient la technologie, d'autres novices en interactions à distance.

Un des problèmes majeurs de la détection fut la gestion des faux positifs, limités en grande partie grâce à l'utilisation du buffer de gestes.

D'un point de vue technique, il a fallu que j'apprenne l'utilisation des évènements en C# et surtout la gestion de la mémoire afin d'éviter les fuites mémoire, ce qui ne fut pas toujours évident.

Concernant les tests, la plus grosse difficulté fut de réussir à réinjecter les données dans la DLL comme si elles provenaient du Leap, à la fois à cause du fait que la classe *Frame* soit en lecture seule, mais surtout car il a fallu simuler le frame rate du Leap. De plus, pour calculer les statistiques sur les fichiers CSV d'un même geste, il fallait lire tous les fichiers les uns après les autres mais en faisant bien attention à ce que l'intégralité du fichier soit passée par le moteur de reconnaissance avant d'en lancer un autre. Sinon, les évènements n'avaient pas le temps d'être reçus et traités, et les gestes détectés se mélangeaient d'un fichier à l'autre. Pour cela, j'ai utilisé des évènements, levés par la classe de rejeu, qui informaient la classe de statistiques lorsqu'un fichier avait été totalement rejoué.

Le frame rate élevé et variable du Leap a aussi constitué une difficulté majeure. Par exemple, il a fallu compresser les évènements pour le pointage afin de ne pas saturer le Composer. Cela a aussi posé problème pour simuler le frame rate lors du rejeu. J'ai utilisé un timer dont l'intervalle était positionné après chaque envoie de frame. En effet, dans les fichiers CSV était aussi stocké le timestamp (horodatage) des frames, ainsi en donnant à l'intervalle du timer la différence entre le timestamp de la prochaine frame à envoyer et celui de la frame qui vient d'être envoyée, on peut simuler le frame rate.

Enfin, la dernière difficulté fut d'intégrer la DLL au Composer sous la forme d'interface assets. En effet, le développement du module Leap Motion était complètement indépendant des autres projets d'IntuiLab alors que la phase d'intégration à IntuiFace nécessitait de connaître les méthodes de l'entreprise, telles que l'utilisation des descripteurs, comment créer un design accelerator et quelques notions sur le fonctionnement interne du Composer et du Player.

7. Les bénéfices de cette expérience

Cette expérience fut très enrichissante. Tout d'abord, j'ai eu la possibilité de travailler sur un capteur à la pointe de la technologie, très innovant, qui n'était même pas encore sorti au grand public au début de mon stage, et j'en suis très contente. J'ai pu découvrir l'ambiance de travail dans une entreprise de taille moyenne et cela m'a beaucoup plu. Les employés sont tous très accueillants et disposés à fournir de l'aide lorsqu'un collègue en a besoin. J'ai aussi découvert le travail en collaboration avec un ergonome, les contraintes à respecter pour satisfaire les utilisateurs.

De plus, j'ai réalisé un projet concret de A à Z, qui impliquait des contraintes de temps et nécessitait de prédire les attentes du client. Mon maître de stage m'a laissé une grande autonomie et des responsabilités, cela m'a permis de gagner de la confiance en mon travail en voyant que j'ai réussi à mener le projet à bout. Le module Leap Motion est sorti dans la version 4.1 d'IntuiFace, cela fut gratifiant de le voir intégré au Composer et utilisable par les clients d'IntuiLab.

Ce projet m'a aussi permis d'approfondir mes connaissances en C#, notamment sur la notion d'évènements sur laquelle repose ma DLL. J'ai aussi pu mieux me familiariser avec svn, le logiciel de gestion de versions utilisé à IntuiLab. Enfin, j'ai découvert et approfondi la programmation avec le périphérique Leap.

En résumé, je suis vraiment très satisfaite du déroulement de mon stage et du projet que j'ai eu l'opportunité de réaliser, cela m'a beaucoup apporté en matière de compétences informatiques et de méthodes de travail.

8. Conclusion

Après ces six mois de stage, je pense avoir dans l'ensemble réussi ma mission, bien qu'il y ait quelques parties à améliorer. Une première version de la reconnaissance de gestes et de postures est désormais intégrée à la solution IntuiFace, et le visualiseur 3D, qui pourra être réutilisé pour le développement de nouveaux gestes, est fonctionnel. J'ai pu avancer sur le pointage, qui devra être amélioré par la suite.

C'était la première fois que je travaillais dans une entreprise spécialisée en interactions homme-machine et cela m'a beaucoup plu, c'est une branche de l'informatique où j'aimerais travailler. A l'aube d'un monde où ces interactions font partie du quotidien, je suis fière d'avoir pu contribuer à l'extension d'IntuiFace aux interactions à distance. Je garde de mon stage un excellent souvenir, il constitue désormais une expérience professionnelle valorisante et très enrichissante.

Webographie

Site web d'IntuiLab:
http://www.intuilab.com

Support IntuiFace:
http://support.intuilab.com/kb

Site web de Leap Motion:
https://www.leapmotion.com

Le format JSON :
http://www.json.org/

Facade design pattern :
http://en.wikipedia.org/wiki/Facade_pattern
http://www.codeproject.com/Articles/481297/UnderstandingplusandplusImplementingplusFacadeplus

Singleton pattern :
http://en.wikipedia.org/wiki/Singleton_pattern
http://jlambert.developpez.com/tutoriels/dotnet/implementation-pattern-singleton-csharp/

Créer un descripteur d'interface asset pour les DLL C# :
http://support.intuilab.com/kb/interface-assets-new-in-40/design-an-interface-asset-descriptor-for-a-c-net-dll

Créer un design accelerator pour un interface asset :
http://support.intuilab.com/kb/interface-assets-new-in-40/creating-a-skin-for-your-custom-made-interface-asset

TUIO :
http://www.tuio.org/

Documentation IntuiFace sur les interface assets Leap Motion :
http://support.intuilab.com/kb/how-to/use-leap-motion-with-your-experiences

Annexes

A. Etat de l'art scientifique et technologique sur la reconnaissance de gestes

Reconnaissance de gestes
Etat de l'art scientifique et technologique

Marie Pietrowski

27 avril 2013

1 Introduction

La possibilité d'interagir avec un ordinateur par les gestes permet d'enrichir l'expérience utilisateur en proposant des interfaces homme-machine plus naturelles et intuitives. La reconnaissance de gestes a pour but d'analyser et d'interpréter les gestes de l'homme par des algorithmes mathématiques. L'utilisation d'interfaces gestuelles apporte plusieurs avantages [1] :

- *Une interaction naturelle :* la communication gestuelle est un moyen naturel de communication pour l'homme. De plus, certains mouvements sont plus ou moins universels et se rapprochent de la manipulation directe des objets dans le monde réel.
- *Une interaction simple et puissante :* grâce à l'aspect dynamique du mouvement, l'utilisateur peut exprimer en une seule action à la fois une commande et ses paramètres.
- *Une interaction directe :* le corps utilisé comme périphérique d'entrée permet l'élimination d'objets intermédiaires entre l'utilisateur et la machine.

Les travaux de reconnaissance de gestes sont motivés par la possibilité de manipulation d'objets 3D, le pointage et la sélection de menus, la reconnaissance de gestes pour les commandes gestuelles, ou encore la reconnaissance de la langue des signes.

2 Les périphériques d'acquisition

Il existe deux grandes catégories de périphériques de capture dans le domaine de la reconnaissance des gestes :

- *Ceux basés sur la Vision par Ordinateurs :* ces appareils utilisent des caméras pour analyser et interpréter les mouvements depuis les séquences vidéo enregistrées en s'appuyant sur le traitement d'images.
- *Ceux basés sur le contact :* des appareils tels que des télécommandes où des gants peuvent agir comme des extensions du corps et ainsi capter ses mouvements.

Les premiers périphériques d'acquisition font partis de la deuxième catégorie. L'apparition des gants numériques avec le *DataGlove* de VPL en 1987 marque le début des travaux de recherche en reconnaissance automatique des gestes. Le DataGlove utilise 10 capteurs, marche à 60 Hz et possède une précision de l'ordre de 5-10 degrés. Parmi ces périphériques, on peut aussi noter les *CyberGloves* d'Immersion [2] qui sont des gants sans fil qui possèdent 18 à 22 capteurs placés à des points critiques permettant de mesurer la posture de la main. Ces gants peuvent aller jusqu'à 149Hz et ont une précision d'à peu près 1 degré. L'inconvénient majeur des gants numériques est leur caractère intrusif.

La *VisionWand* (baguette magique visuelle) [3] est une baguette dont les deux extrémités sont de couleur vive pour permettre aisément leur détection et qui est utilisée pour l'interaction avec des objets 2D ainsi que la sélection de menus. Le système peut reconnaître 9 commandes gestuelles.

Plus récemment, en 2006, est sortie la *Wiimote* qui sert de périphérique d'entrée à la console de jeu Wii de Nintendo. Cet appareil est équipé d'accéléromètres qui permettent la détection des mouvements et rotations en trois dimensions [4]. La manette est en plus munie d'une caméra sensible à une certaine longueur d'onde émise par une barre de LED placée à côté de l'écran, ce qui lui permet de calculer sa position par rapport à cette barre et ainsi pointer des objets sur l'écran.

En 2010, Microsoft a sorti un périphérique de reconnaissance de gestes initialement prévu pour contrôler sa console de jeu Xbox 360, appelé la *Kinect*. Cet appareil permet aussi d'interagir par commandes vocales. Kinect a connu un grand succès auprès du grand public et des scientifiques, surtout lorsque Microsoft a élargi ses possibilités en sortant une version pour Windows en 2012. Cette version pour PC inclut

un SDK pour permettre aux développeurs de programmer la Kinect. Ce périphérique possède un système de détection de profondeur 3D composé d'un émetteur laser infrarouge et d'une caméra infrarouge [5, 6]. L'émetteur IR projette une lumière IR structurée à 830 nm, et la caméra interprète ces données grâce aux techniques de stéréoscopie. La lumière émise par la Kinect permet de donner une texture à l'ensemble de la scène filmée, contournant ainsi les limitations de la stéréoscopie dans les zones dépourvues de texture. En plus de ce dispositif, la Kinect possède une caméra RGB ainsi qu'un micro. Le SDK Kinect permet de récupérer trois différents flux : celui de la caméra de profondeur, celui de la caméra RGB, ainsi que le squelette de l'utilisateur. Ce squelette est composé de 20 jointures dont les positions sont récupérées à chaque frame. Ainsi, le développeur peut travailler sur cette information pour coder la reconnaissance de ses propres gestes ou postures, ainsi que sur l'image vidéo pour détecter par exemple un poing fermé grâce aux techniques de traitement d'images.

Le dernier périphérique de capture de mouvements en date est le contrôleur *Leap* de la société *Leap Motion* [7]. C'est une petit boîtier de quelques centimètres qui se branche par port USB à l'ordinateur. Contrairement à la Kinect qui traque l'ensemble du corps, le Leap localise seulement les mains et leurs doigts ou les outils qu'elles portent, et ce au centième de millimètre près. La date de sortie de cet appareil est prévue pour le 13 Mai 2013 mais il est déjà accessible aux développeurs qui en ont fait la demande, accompagné de son SDK disponible pour plusieurs langages de programmation. Malheureusement, le SDK ne permet pas l'accès aux données brutes. D'après des recherches effectuées à Intuilab, le Leap est apparement composé de deux caméras IR séparées d'une distance fixe qui délivrent des images brutes qui sont ensuite traitées directement dans le CPU de l'ordinateur auquel est branché le périphérique.

Le prochain détecteur de mouvements attendu est un brassard qui se porte à l'avant-bras baptisé *Myo* [8]. La société Thalmic Labs a annoncé sa sortie en quantité limitée pour la fin de l'année 2013. L'appareil est doté de capteurs, d'un gyroscope et d'un processeur ARM qui traite l'ensemble des données. Myo analyse l'activité électrique des muscles de l'avant-bras ainsi que sa position dans l'espace pour permettre la détection des mouvements de l'utilisateur et interagir avec un ordinateur.

3 Les algorithmes de reconnaissance de gestes

Avant de pouvoir passer à l'étape de reconnaissance à proprement parler, un système de reconnaissance de gestes doit passer par une phase en amont afin de préparer les données brutes à être analysées [9]. Pour cela, les périphériques basés sur des caméras utilisent les techniques de traitement d'images. La reconnaissance se décompose donc en quatre étapes :

- *L'initialisation :* Cette étape permet de s'assurer que le système possède une interprétation correcte de la scène.
- *Le tracking :* Tout d'abord, le système effectue la segmentation du sujet ou des objets d'intérêts afin de les séparer de l'arrière-plan. Ensuite, l'image est transformée afin de ne garder que les informations utiles pour la suite, puis, le système définit une méthode pour pouvoir suivre le sujet d'une frame à l'autre.
- *L'estimation de pose :* Il y a ensuite une étape d'évaluation de configuration du corps du sujet.
- *La reconnaissance :* Ce n'est qu'à ce moment que les données pourront être analysées afin de reconnaître les actions effectuées par l'utilisateur.

C'est cette dernière étape de reconnaissance que nous allons approfondir. En effet, lorsque les périphériques de capture sont livrés aux développeurs avec leur SDK, comme la Kinect et le Leap, les trois premières étapes sont déjà implémentées et le travail du développeur consiste donc à utiliser les données délivrées par le SDK pour construire la reconnaissance de ses propres gestes.

On distingue deux types de reconnaissance :

- La reconnaissance statique, basée sur une seule frame : elle utilise les données spatiales de la frame et compare des informations pré-enregistrées avec l'image courante.
- La reconnaissance dynamique, qui utilise plusieurs frames et se sert de données temporelles.

Plusieurs algorithmes et techniques sont utilisés pour analyser et reconnaître les mouvements [10]. Nous allons voir ici une description de quelques-unes de ces méthodes.

1. **Analyse en composantes principales :** L'ACP est une technique statistique d'analyse des données qui consiste à transformer des variables liées entre elles en variables décorrélées. Cette méthode permet de diminuer le nombre de variables. Appliquée à la reconnaissance de gestes 3D utilisant une base de données, elle permet de réduire la taille de la base tout en gardant la capacité de reconnaître un grand nombre de gestes. Comme expliqué dans [11], les coordonnées 3D d'un nuage de points représentant un geste changent après application d'une rotation ou translation dans le repère

caméra, tandis qu'elles restent inchangées dans le repère ACP. Ainsi, l'ACP permet de trouver une description particulière d'un nuage de points indépendante de sa position ou son orientation dans l'espace 3D.

2. **Méthodes Bayésiennes :** Dans [12], il est démontré comment l'utilisation d'un classifieur bayésien peut permettre de reconnaître des gestes prédéfinis. Le principe est d'extraire des images vidéo brutes une *motion gradient orientation image* pour former un vecteur caractéristique du mouvement. Ce vecteur est ensuite classé par un classifieur bayésien qui permet de le relier au geste qu'il représente.
3. **Modèles de Markov Cachés :** Ce sont des modèles statistiques définis par des collections d'états finis connectés par des transitions. Chaque état est caractérisé par une probabilité de transition et une probabilité de sortie. L'utilisation des MMC nécessite un apprentissage. Cette technique est très utilisée à la base en reconnaissance de parole et depuis une vingtaine d'années en reconnaissance de gestes, notamment dans [13], [14] et [15]. Les MMC permettent d'atteindre un taux d'erreur très faible.
4. **Réseaux de neurones :** Ce sont des modèles de calcul inspirés du fonctionnement biologique des neurones. Ils sont constitués de plusieurs neurones formels. Chaque neurone est une unité qui transforme ses entrées selon une règle précise qui lui est propre pour par exemple effectuer une classification. Les neurones sont connectés pour former un réseau complexe. Tout comme les modèles de Markov cachés, les réseaux de neurones nécessitent un apprentissage, qui permet de moduler le poids de chaque connexion pour aboutir à une classification optimale. Avec les MMC, les réseaux de neurones sont les deux techniques les plus employées dans la reconnaissance de gestes [16, 17].
5. **Dynamic Time Warping :** C'est un algorithme qui permet de calculer la similitude entre deux séquences qui peuvent varier au cours du temps. Ainsi, il peut reconnaître deux séquences d'un même mouvement même si la vitesse varie [18].

4 Conclusion

Le désir de l'homme de pouvoir communiquer avec les ordinateurs de manière plus naturelle, comme il le ferait avec ses semblables, fait de la recherche en reconnaissance de gestes mais aussi de paroles une quête importante de nos jours. Dans ce document, nous avons d'abord décrit les différents appareils de capture, des plus anciens à ceux dont on attend encore la sortie. Puis, nous avons parlé des différentes étapes de la reconnaissance de gestes ainsi que de certains algorithmes utilisés.

Au vu de l'évolution des périphériques d'acquisition qui aspirent à être de plus en plus précis et rapides dans le suivi du corps, et des algorithmes de plus en plus fiables, on peut se demander si les interactions gestuelles supplanterons les interactions classiques du clavier et de la souris.

Références

[1] T. Baudel and M. Beaudouin-Lafon. Charade : Remote control of objects using free-hand gestures. *Communications of the ACM*, pages 28–35, July 1993.

[2] G. Drew Kessler, Larry F. Hodges, and Neff Walker. Evaluation of the cyberglove as a whole hand input device.

[3] X. Cao and R. Balakrishnan. Visionwand : Interaction techniques for large displays using a passive wand tracked in 3d. *Symposium on User Interface Software and Technology (UIST)*, page 173–182, 2003.

[4] T. Schlömer, B. Poppinga, N. Henze, and S. Boll. Gesture recognition with a wii controller. *Proceedings of the Second International Conference on Tangible and Embedded Interaction*, Feb 2008.

[5] J. Kramer, N. Burrus, D. Herrera C., F. Echtler, and M. Parker. *Hacking the Kinect*. Apress, 2012.

[6] A. Lejeune, S. Piérard, M. Van Roogenbroeck, and J. Verly. Utilisation de la kinect. *Linux Magazine France*, pages 16–29, Juillet-Août 2012.

[7] Leap Motion Official Website. https ://www.leapmotion.com/.

[8] Myo Official Website. https ://getmyo.com/.

[9] J. Thomet. Une vue d'ensemble de la reconnaissance de gestes. 2009.

[10] G. R. McMillan. The technology and applications of gesture-based control. *RTO EN-3*, October 1998.

[11] O. Ben-Henia and S. Bouakaz. Utilisation de l'acp pour la reconnaissance des gestes 3d de la main. *ORASIS - Congrès des jeunes chercheurs en vision par ordinateur*, Mai 2011.

[12] S.-F. Wong and R. Cipolla. Real-time interpretation of hand motions using a sparse bayesian classifier on motion gradient orientation images. 2005.

[13] J. Yang and Y. Xu. Hidden markov model for gesture recognition. 1994.

[14] C. Keskin, A. Erkan, and L. Akarun. Real time hand tracking and 3d gesture recognition for interactive interfaces using hmm.

[15] C. Joslin, A. El-Sawah, Q. Chen, and N. Georganas. Dynamic gesture recognition. *IMTC 2005 – Instrumentation and Measurement Technology Conference*, 2005.

[16] K. Murakami and H. Taguchi. Gesture recognition using recurrent neural networks. *CHI'91*, pages 237 242, 1991.

[17] K. Symeonidis. Hand gesture recognition using neural networks. 2000.

[18] G.A. ten Holt, M.J.T. Reinders, and E.A. Hendriks. Multi-dimensional dynamic time warping for gesture recognition. *Thirteenth annual conference of the Advanced School for Computing and Imaging*, June 2007.

B. Exemple de fichier CSV d'enregistrement de geste

Frame id;Finger id;Associated hand id;Is frontmost;Direction;Tip position;Tip velocity;Length;Width;Timestamp

173201;4;-1;True;(0.552769, 0.261241, -0.791328);(115.043, 116.483, 6.4827);(-1296.3, 172.942, -659.359);87,77987;18,62399;857284920

173203;4;-1;True;(0.51246, 0.268291, -0.815724);(100.059, 118.342, -0.374727);(-1601.47, 198.741, -732.944);86,17216;18,55474;857294276

173205;4;-1;True;(0.470207, 0.272345, -0.839484);(80.3302, 120.637, -7.52373);(-2108.72, 245.352, -764.109);84,5639;18,62233;857303632

173207;4;-1;True;(0.425499, 0.280096, -0.860521);(49.8731, 123.786, -15.4876);(-3255.7, 336.54, -851.291);83,23297;18,68582;857312987

173209;4;-1;True;(0.376261, 0.286638, -0.88106);(14.7907, 127.516, -23.1528);(-3749.73, 398.704, -819.285);82,02968;18,62429;857322343

173213;4;-1;True;(0.312366, 0.301083, -0.900986);(-30.7528, 131.439, -27.8624);(-2434.05, 209.635, -251.705);82,02968;18,62429;857341054

173219;4;-1;True;(0.292081, 0.264363, -0.919131);(-56.1391, 129.612, -17.1255);(-904.488, -65.0929, 382.548);82,02968;18,62429;857369121

173221;4;-1;True;(0.276517, 0.235353, -0.931744);(-82.0309, 128.765, -9.46903);(-2767.7, -90.5297, 818.435);75,49521;18,23138;857378476

173225;4;-1;True;(0.240973, 0.220079, -0.94525);(-122.163, 129.261, -0.529972);(-2144.71, 26.5308, 477.718);70,41193;17,92158;857397188

173227;4;-1;True;(0.22577, 0.18713, -0.956039);(-160.866, 130.687, 10.525);(-4137.15, 152.419, 1181.72);66,7078;17,98193;857406543

C. Exemple de fichier de statistiques

Anthony_Swipeleft2_f.csv
Joanna_swipeleft1_f.csv
Joanna_swipeleft2_f.csv
Louis_swipeleft1_f.csv
Louis_swipeleft2_f.csv
Louis_swipeleft5_f.csv
Michel_swipeleft1_f.csv
Michel_swipeleft2_f.csv
Mikael_swipeleft1_f.csv
Mikael_swipeleft3_f.csv
Roland_swipeleft2_f.csv
Roland_swipeleft3_f.csv
Rémy_swipeleft3_f.csv
Nb de fichiers non reconnus : 13 sur 51
Taux de réussite : 74,5098%

minTimeBetweenGestureDetections = 0.7s
nbMinFrames = 5
minSpeed = 1000
maxSpeed = 4000

Anthony_Swipeleft2_f.csv
Joanna_swipeleft1_f.csv
Joanna_swipeleft2_f.csv
Louis_swipeleft2_f.csv
Louis_swipeleft5_f.csv
Michel_swipeleft1_f.csv
Mikael_swipeleft1_f.csv
Roland_swipeleft2_f.csv
Roland_swipeleft3_f.csv
Rémy_swipeleft3_f.csv
Nb de fichiers non reconnus : 10 sur 51
Taux de réussite : 80,39216%

minTimeAfterPushDetections = 0.4s
minTimeAfterSwipeDetections = 0.7s
nbMinFrames = 5
minSpeed = 1000
maxSpeed = 4000

D. Extrait du descripteur

```
{
    "kind": "discovery#restDescription",
    "discoveryVersion": "v1",
    "id": "IntuiLab.Leap",
    "name": "Leap Motion",
    "version": "1.0",
    "title": "Leap Motion DLL",
    "protocol": "dll",
    "basePath": "IntuiLab.Leap",
    "dependencies": [
        "IntuiLab.Leap.dll",
        "LeapCSharp.NET4.0.dll",
            "Bespoke.Common.Osc.dll"
    ],
    "schemas": {
        "GestureFacade": {
            "id": "GestureFacade",
            "type": "object",
            "title": "Gestures",
            "properties": {
                "EnableSwipeLeftGesture": {
                    "type": "boolean",
                    "title": "Enable Swipe Left Gesture"
                },
                "EnableSwipeRightGesture": {
                    "type": "boolean",
                    "title": "Enable Swipe Right Gesture"
                },
                "EnableTapGesture": {
                    "type": "boolean",
                    "title": "Enable Tap Gesture"
                }
            }
        }
    },
    "resources": {
        "GestureFacade": {
            "id": "GestureFacade",
            "title": "Gestures",
            "description" : "Hand movements for swipe and tap",
            "isExternalAsset": "true",
            "templateDesignAccelerator" : "IntuiLab.Leap.Gestures.ifa",
            "methods": {},
            "events": {
                "SwipeLeftGestureDetected": {
                    "id": "SwipeLeftGestureDetected",
                    "title": "Swipe Left Gesture detected",
                    "description": "Raised when a Swipe Left gesture is
                    detected."
                },
```

```
                "SwipeRightGestureDetected": {
                    "id": "SwipeRightGestureDetected",
                    "title": "Swipe Right Gesture detected",
                    "description": "Raised when a Swipe Right gesture is
                    detected."
                },
                "TapGestureDetected": {
                    "id": "TapGestureDetected",
                    "title": "Tap Down Gesture detected",
                    "description": "Raised when a Tap Down gesture is
                    detected."
                }
            }
        }
    }
}
```

E. Etat de l'art sur la manipulation directe dans une IHM avec des interactions à distance

Le concept de la manipulation directe en IHM est utilisé depuis longtemps et doit respecter certaines règles. Avant d'expliciter les bonnes manières pour effectuer de l'interaction à distance, il semble bon de les rappeler.

Règles générales pour la conception d'interfaces à manipulation directe

- Les actions doivent être initialisées par l'utilisateur, le système est en attente des entrées utilisateur.
- L'entrée utilisateur est basée sur la reconnaissance et pointage.
- L'interface doit être simple, elle ne doit pas être surchargée par trop d'éléments éventuellement complexes.
- Les systèmes à manipulation directe opèrent généralement dans un mode principal unique. Si le système est dans un mode spécifique, le mode courant doit être identifiable de façon visuelle (forme de curseur par exemple).
- La réversibilité doit être de règle. Si elle n'est pas possible, notamment pour des opérations complexes, cela doit être indiqué.

Attention à la « prévisibilité »

Un dialogue est ambigu si l'action de l'utilisateur peut avoir des effets différents qui ne peuvent pas être prédits par l'utilisateur selon l'état courant de l'affichage. (Ceci indique des modes cachés.)

Objets

- Les interfaces utilisateur devraient proposer des objets de manipulation simples et des actions également simples et génériques.
- Les objets manipulables doivent être représentés visuellement sur l'écran et accessibles directement.
- La sélection d'un objet devrait être implicitement combinée avec l'exécution d'une action.
- La sortie d'information devrait également être affichée comme un objet directement manipulable.

Actions

- Les interfaces utilisateur doivent respecter un style d'interaction (syntaxe).

- Dans la manipulation directe, la syntaxe « objet -> action » est plus naturelle pour l'utilisateur.
- Les actions génériques devraient être proposées pour les fonctions sémantiquement similaires.
- Les accélérateurs devraient être disponibles pour les utilisateurs experts pour augmenter leur performance.
- Après chaque action un retour visuel devrait être présenté à l'écran.
- Toute action simple devrait être réversible.

Bonnes pratiques à avoir avec les interactions à distance

Dans une IHM, l'utilisation d'interactions à distance via des capteurs (Kinect, Leap Motion, Mio, etc…) apporte de l'instabilité au niveau du curseur de l'IHM. Prenons par exemple le capteur Kinect, il est très difficile de ne pas faire bouger la main sur l'IHM. De ce fait, certaine pratique sont bonnes à prendre lorsque l'on utilise ce type d'interactions :

- Utiliser des résolutions types HD (1920*1080), cela permet de moins voir ces fluctuations de position qui se feront ressentir de plus en plus pendant une utilisation prolongée.
- Les objets doivent avoir une taille minimale raisonnable (220*220)
- Lorsque l'utilisateur passe son curseur sur un objet où une interaction est possible, mettre en place un feedback permettant de lui signifier qu'une action lui est possible.
- L'utilisateur doit savoir, par le biais de feedbacks comment le capteur le perçoit. Dans le cas d'une utilisation de l'IHM en multi-user, la différenciation des deux feedbacks doit être visible (cas du Kinect : deux user => deux squelettes de couleurs différentes.

Multi – User

L'interaction à distance permet d'ouvrir la porte aux sessions multi-user sur une IHM. Plusieurs possibilités peuvent être utilisées :

- Collaboratif :
    - Un seul user contrôle l'IHM (le driver) le second reste spectateur et pourra interagir avec l'IHM seulement à des moments bien précis.

Pratique facile à mettre en place, l'IHM doit seulement posséder une logique de sélection du driver.

Tous les users peuvent interagir simultanément. Concept plus complexe à utiliser et l'IHM doit être robuste à différent cas comme la manipulation d'un objet par deux users différents.

- Non – Collaboratif : l'écran de l'IHM est splitté, chaque user effectue l'interaction qu'il souhaite sur sa partie de l'IHM. Pas de communication entre les deux users.

Zone d'interaction

Il est recommandé de définir des zones d'interaction pour permettre à l'utilisateur de bien comprendre le déplacement du curseur sur l'IHM. Ces zones sont surtout utiles avec les capteurs qui utilisent le corps entier d'un user, cela permet de lui éviter de s'étirer de façon trop excessive pour atteindre une extrémité de l'IHM.

Interactions

Pour effectuer de la manipulation directe dans une IHM au travers d'interactions à distances, des points précis sont traqués par les capteurs, et en majeure partie ce sont les mains. Sur un user, il est préférable de traquer une seule main à la fois pour effectuer des interactions. La raison est que pour l'ensemble des capteurs, le rapprochement des mains peut donner des comportements (graphiques) déroutants pour l'utilisateur.

Voici différents types d'interactions que l'on peut mettre en place pour effectuer de la manipulation directe.

- *Locking Hand Closed :*

Ce type d'interaction est basé sur l'état de la main, ouverte ou fermée. Lorsque la main est ouverte le curseur effectue seulement du pointage, il renseigne sur la position de l'utilisateur dans l'IHM. Lorsque la main se ferme le curseur agit comme un touch (dans le monde du tactile).

http://www.youtube.com/watch?v=WUSwVsyc3S4

https://www.leapmotion.com/ (à la fin de la vidéo)

- *Hand Z-Index :*

Ce type d'interaction consiste à mettre un mur imaginaire devant l'utilisateur. Lorsque la main de l'utilisateur est derrière ce mur le curseur effectue seulement du pointage, il renseigne sur la position de l'utilisateur dans l'IHM. Lorsque la main franchit le mur le curseur agit comme un touch.

On peut voir ce principe d'interaction utilisé dans ces vidéos :

http://blog.leapmotion.com/post/50933421954/windows-os-video

http://www.youtube.com/watch?v=vZSEEnMP6pg (+ utilisation du Kinect avec TUIO)

- *Hand Pointing / Hand Action :*

Ce type d'interaction est basé sur l'identification d'une main primaire et une main secondaire. La main primaire est la première à entrer dans la zone de pointage, ce sera cette main qui permettra de représenter le curseur et la main secondaire permettra de déclencher un touch en entrant dans la zone de pointage, à l'endroit où la main primaire se trouve.

http://www.youtube.com/watch?v=_AkUh3TpP-A

Une évolution de ce type d'interaction, on peut imaginer que la main secondaire puisse déclencher des actions particulières en effectuant des gestes spécifiques.

L'utilisation de ces différents types d'interactions peut se faire sous forme de combo, par exemple on peut se servir du *Hand Pointing / Hand Action* et l'enrichir du *Hand Z-Index* sur la main secondaire pour déclencher le touch.

F. Diagramme de classe simplifié de la DLL