POLYTECH
LILLE

Rapport de projet :

Outils de Maquettage Virtuel

IMA5 2013-2014

Matthias De Bie - Pierre-Jean Petitprez

Table des matières

Table des figures

Dans le cadre de la dernière année de cycle ingénieur à Polytech Lille, spécialité IMA, nous réalisons un projet de fin d'étude. Nous avons choisi le sujet "Outils de maquettage virtuel" proposé par l'équipe MINT du LIFL. Dans l'optique d'une collaboration avec une artiste (Mathilde Lavenne), le projet s'inscrit dans le domaine des intéractions homme-machine. Ce projet en suit un autre réalisé en stage dans les mêmes conditions. Le but de cet autre projet était de créer une expérience d'exploration et de manipulation dans un univers 3D (utilisant le moteur de jeu Unity) grâce à une caméra Kinect et à un logiciel de reconnaissance de geste (*ThreeGear*). Pour ce faire, le stagiaire a réalisé plusieurs scripts et une scène grâce à Unity : le résultat est un univers 3D simpliste qui permet de déplacer des cylindres par détection de collision ou de les saisir grâce à une action "pincée" de la main.

La première étape de notre projet était de développer une brique logicielle permettant d'utiliser n'importe quel type de capteur gestuel en entrée et d'en sortir un flux générique et réutilisable dans plusieurs applications 3D. En nous inspirant de ce qui a été fait précédemment, nous utilisons la plateforme Unity comme interface visuelle et pour recevoir les informations.

La deuxième étape consistait alors à créer une première scène Unity capable :

- de recevoir les informations de notre premier logiciel pour traiter les gestes de l'utilisateur
- de recevoir les informations d'une Kinect (différente de celle pour la detection des gestes) pour détecter principalement la tête de l'utilisateur et ainsi construire une vue en perspective donnant l'impression d'une 3D quelle que soit la position de l'utilisateur.

La troisième étape consistait à créer une deuxième scène Unity capable de synchroniser la position de sa caméra principale en fonction de la position d'un objet se déplaçant dans la première scène.
Ce projet s'est déroulé de septembre 2013 à février 2014. Dans ce rapport, nous exposons le cahier des charges du projet, le travail déjà réalisé et enfin les perspectives d'évolution.

1 Travail préparatoire

Avant d'aborder la première partie il est important de rappeler les logiciels que nous utilisons :

- ThreeGear[1] : Logiciel de captation utilisant une Kinect et qui émet diverses informations, notamment la géométrie des mains, sur le réseau via le port 1988.
- Unity[2] : Moteur de jeu 3D temps réel utilisé dans notre cas pour la visualisation de la scène 3D par l'utilisateur.

1.1 Cahier des charges

Étant donné les évolutions récentes du projet il nous a semblé utile de revenir sur le cahier des charges.

Il a été décidé au début du projet de créer un programme qui soit le plus adaptable possible aux données d'entrée (capteurs) et de sortie (applications graphiques), pour ce faire deux parties constituaient le programme que nous avons développé :

- La récupération des informations provenant des différents capteurs, incluant généralisation et normalisation des coordonnées.
- L'interprétation des gestes et l'envoi des messages à l'application utilisateur (sous-entendu l'application graphique) via un protocole TCP. Le protocole d'envoi des messages était laissé libre.

Il nous a également été demandé de développer une application graphique sous Unity utilisant l'outil précédemment décrit. Un exemple qui s'inscrivait dans le cadre de la collaboration avec l'artiste serait une scène 3D dans laquelle il est possible d'attraper une bague ou un curseur, permettant ainsi de lancer le visionnage d'une vidéo (ou tout autre contenu artistique) en fonction de la position de ce curseur sur un chemin prédéfini symbolisant par exemple la ligne temporelle de la vidéo.

En addition des précédents besoins et au fur et à mesure de la définition de l'oeuvre finale, il nous a été demandé d'ajouter une fonctionnalité à notre scène Unity. En effet cette première scène doit être visualisée sur une large table tactile. Pour donner l'impression à l'utilisateur qu'il observe réellement un univers 3D, il a fallu utiliser une autre caméra Kinect pour suivre la position de la tête de l'utilisateur et adapter le rendu de la perspective en temps réel. Nous aborderons le côté technique plus loin dans le rapport.

Enfin, il a fallu créer une autre scène Unity qui puisse récupérer les données de la première scène pour visualiser une œuvre mélangeant 2D et 3D.
Le projet se découpe en plusieurs grandes phases. Notre gestion du projet est visible via le diagramme Gantt de l'annexe A.

1. http ://threegear.com/
2. http ://unity3d.com/

1.2 Travail d'analyse

L'équipe MINT travaille en collaboration avec une artiste qui souhaite utiliser cette technologie pour une œuvre interactive : l'utilisateur pourrait déplacer un curseur le long d'une ligne temporelle, et selon la direction et la position choisies sur cette ligne, une partie de la vidéo créée par l'artiste serait diffusée.

La vidéo en question est disponible à l'adresse : `http://vimeo.com/76143490`

Comme dit précedemment dans le cahier des charges, l'oeuvre finale est composée d'une table-écran, d'un écran standard, d'au moins une Kinect et bien sûr d'un utilisateur. En plus de ces conditions et tel que le projet est prévu pour le moment, il faudra deux ordinateurs reliés en réseau et un dispositif de suivi des mains (cela peut être une Kinect seule, un autre dispositif tel que des gants ou une combinaison de plusieurs de ces dispositifs).

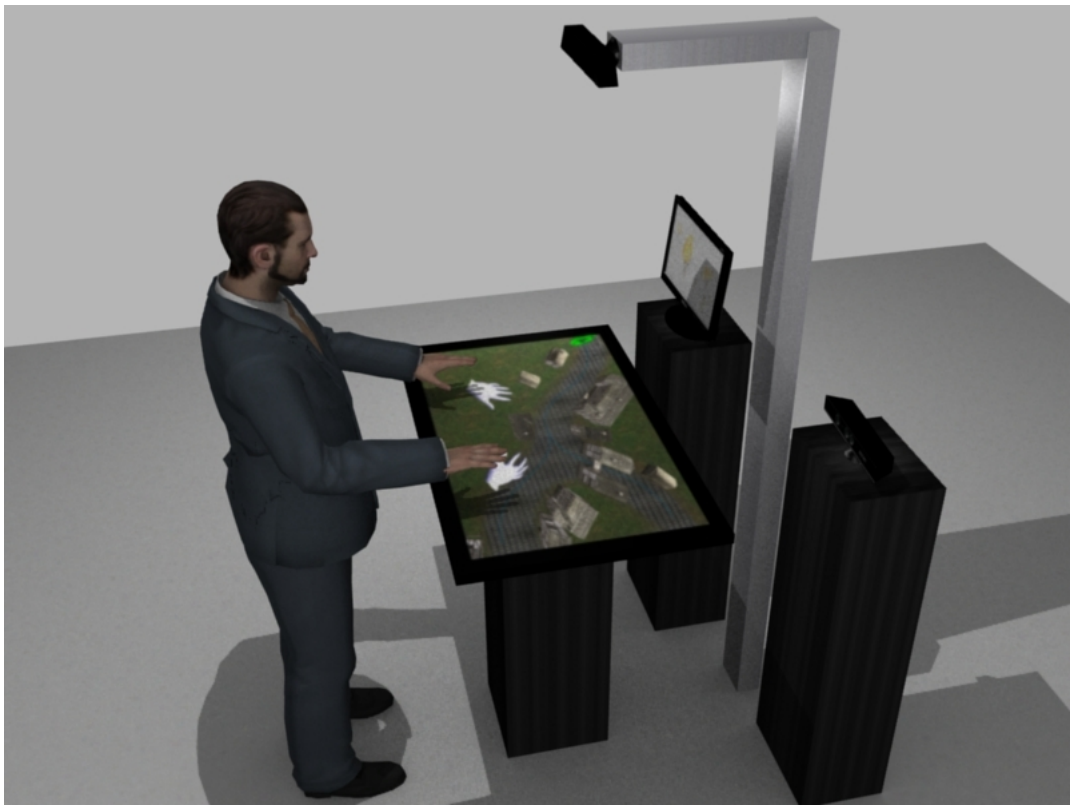

FIGURE 1.2.1 – Rendu 3D de l'installation finale telle que pensée au début du projet

La présence de deux ordinateurs est obligatoire car il n'est pas possible de brancher deux capteurs de type Kinect en raison d'une limitation dans la configuration de l'utilitaire ThreeGear qui ne permet pas de choisir manuellement la caméra à utiliser, et il est actuellement impossible d'avoir deux fenêtres Unity actives en même temps (quand l'une est active l'autre est en pause). Ce problème pourrait être en partie résolu en utilisant par exemple une machine virtuelle mais l'implémentation finale de la solution ne nous revient pas et ne sera donc pas abordée ici plus en détail.
De plus, les exigences et limitations des technologies utilisées imposent l'utilisation de PC en architecture 64 bits et équipées d'un système d'exploitation Microsoft Windows 7.

1.2.1 Choix du C++

La technologie utilisée pour la première partie du projet était laissée libre. Cependant, au vu des fonctionnalités demandées, notamment une gestion multithreadée, il nous a semblé judicieux de choisir le C++. Ce langage s'appuie en effet sur des notions que nous connaissions déjà (les bases issues du C, auxquelles s'ajoutent les notions de programmation orientée objet) et permet de travailler aussi bien à bas niveau qu'à haut niveau. Le choix du langage a ainsi orienté le choix de l'outil de développement : pour développer un logiciel d'une taille déjà conséquente, un environnement de développement intégré était nécessaire ; nous nous sommes tournés vers l'IDE Visual Studio 2010 de Microsoft.

1.2.2 Choix du C♯

Unity offre la possibilité d'écrire des scripts qui permettent de définir le comportement des objets de la scène. Deux langages de programmation sont disponibles : javascript et C♯. Etant donné que de nombreux scripts, comme la récuprération de la représentation 3D des mains, étaient déjà disponibles et écrits en C♯, nous avons continué avec ce langage. De plus, il est assez proche d'un langage que nous avons déjà pratiqué, à savoir Java.

1.3 Analyse bibliographique

Étant donné l'aspect relativement novateur et original de ce projet, une grande partie du travail réalisé repose sur nos propres expérimentations et sur une évolution empirique de nos algorithmes. Néanmoins on peut citer le travail de thèse de Bruno De Araújo [3], chercheur postdoctoral dans l'équipe MINT, qui nous a permis de créer une perspective modifiée en fonction de la position de l'utilisateur et lui donnant une impression de 3D immersive et temps réel. Nous utilisons également le protocole TUIO dans une application Unity pour recevoir et mettre à jour des objets (au sens programmation orientée objet) [4].

D'autre part, l'utilisation de nouveaux langages (C++ et C♯) ainsi que de nouveaux logiciels (environnement Visual Studio, Unity) nous a mené à utiliser de manière intensive les documentations respectives de ces langages et logiciels, en particulier concernant les fonctions C♯ propres à l'environnement Unity, notamment pour les calculs mathématiques et la manipulation des objets 3D.

3. http://web.ist.utl.pt/bruno.araujo/
4. Voir : http://www.tuio.org/

2 Travail effectué

2.1 Approche technique : C++

Dans une optique de simplicité, nous avons décomposé l'approche technique du projet C++ en 3 parties, abordant respectivement la réception et le traitement des informations reçues, le stockage des données et enfin la reconnaissance de geste et la routine d'envoi des données.

2.1.1 Première étape : Réception et traitement des données

L'intérêt principal de cette application C++ est la modularité, en effet elle a été conçue pour pouvoir accueillir la plus grande diversité de capteurs en faisant le moins d'adaptation possible sur le code source.
Nous sommes partis du principe que la plupart des capteurs utilisables possèdent un kit de développement logiciel (*Software Development Kit - SDK*) et donc une base pour le développement déjà opérationnelle. En partant de ce constat nous avons mis à profit les singularités du langage C++, en particulier l'héritage multiple, pour ne pas avoir à développer de code redondant.
Pour chaque SDK il suffit de créer une classe dérivée à la fois de notre propre classe d'observateur et de celle propre au SDK. De cette manière la classe fille a accès aux méthodes de traitement d'information propre à chaque SDK et aux méthodes de mise à jour de la structure de données de notre programme.

On peut observer sur le diagramme UML ci-après que *ThreeGearReceiver* est dérivée de *HandTrackingListener* qui est une classe virtuelle d'observateur du SDK ThreeGear, et également de *Receiver* qui lui permet d'avoir accès à la structure de données *UserInformation*. Il est possible de conjuguer plusieurs SDK en même temps, pour ce faire nous utilisons une table de correspondance entre chaque point de la structure des mains et un SDK. Cette table doit être renseignée par l'utilisateur dans un fichier texte externe (*matchingJoints.txt*) avant d'utiliser le programme. Dans ce fichier chaque ligne fait correspondre le nom d'une jointure de main (sous-entendu un *HandPart* comprenant les coordonnées spatiales du point) et le nom d'un SDK. À noter que le protocole utilisé pour la réception n'a pas d'influence car il est complètement intégré au SDK et nous n'avons pas à nous en préoccuper. Dans le cas de ThreeGear par exemple, il s'agit de TCP.
Ainsi pour ajouter une nouvelle classe de réception il faut suivre les étapes suivantes :

- Ajouter un type dans l'énumération *SDK*
- Créer une classe qui hérite de *Receiver* et de la classe propre au SDK
- Ajouter dans la classe *Echo* la routine de lancement du thread client du SDK. La classe *Echo* est responsable du lancement des threads client de chaque SDK et d'instancier les classes "Receiver". Notons que les classes sont instanciées seulement si *Echo* est parvenue à établir une connexion. Dans le cas contraire, il est inutile de continuer car aucune donnée ne pourra être reçue.
- Ne pas oublier de stopper le thread client à l'arrêt du programme.
- Ajouter dans la liste *matchingJoints.txt* les SDK correspondants aux jointures des mains.

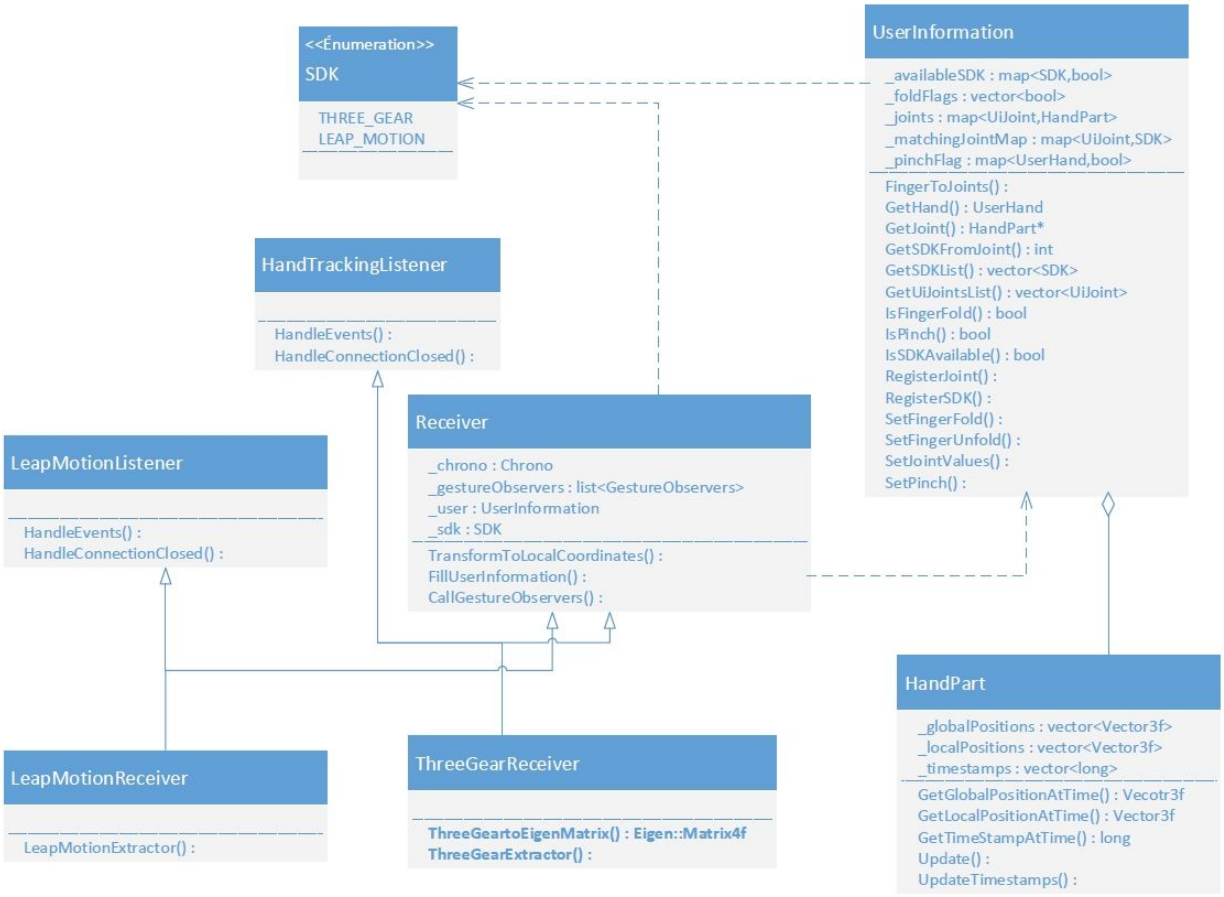

FIGURE 2.1.1 – Diagramme UML partiel : partie réception

2.1.2 Deuxième étape : Stockage des données

Le stockage des données est assuré par une classe instanciée une fois au démarrage, il s'agit de *UserInformation*. Elle contient une table de correspondance constituée d'autant de *HandPart* qu'il y a de jointures dans les deux mains c'est à dire 44 pour le moment[5]. Ce chiffre est susceptible d'être modifié mais pour le moment ThreeGear est le SDK qui met à notre disposition le squelette avec le plus de jointures.

Chaque *HandPart* permet de stocker les données de 3 types :

- Position globale, toutes les coordonnées sont définies dans le repère global du capteur qui les a reçues
- Position locale, les coordonnées des points de la main droite (respectivement gauche) sont données de manière relative à la main droite (respectivement gauche). En réalité les coordonnées sont translatées et tournées de manière à ce que le dos de la main forme un plan fixe dans l'espace quelle que soit la position de la main à chaque instant. Ainsi seuls les mouvements des doigts par rapport au dos de la main sont captés.
- Temps entre chaque évènement capteur, ceci afin de pouvoir déterminer la vitesse d'un mouvement par exemple.

5. Ces jointures correspondent aux poignets, aux différentes articulations des phalanges et aux bouts des doigts.

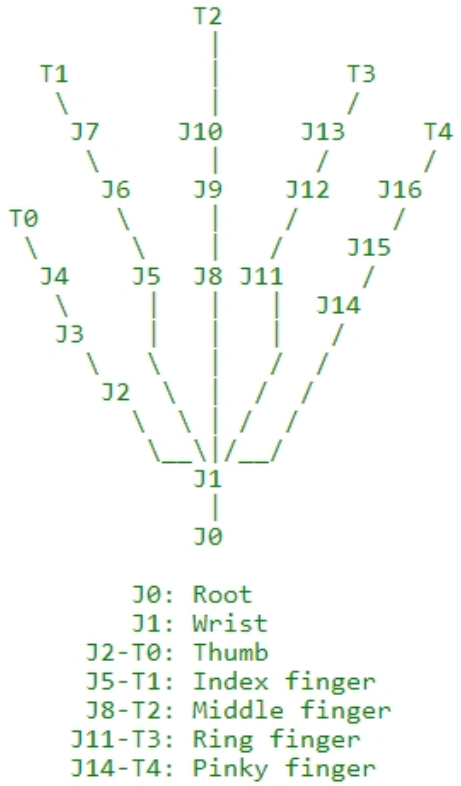

FIGURE 2.1.2 – Représentation des jointures d'une main selon ThreeGear

Il est important de souligner que ces trois types de données sont contenus dans un vecteur de taille 10 par défaut, donc les 10 dernières valeurs sont sauvegardées et mises à jour à chaque évènement. En fonction des gestes qui seraient implémentés à l'avenir, cette taille pourrait être revue mais jusqu'à présent, un historique de 10 valeurs est suffisant.

Mis à part la liste de tous les *HandPart*, la classe *UserInformation* contient beaucoup d'informations, certaines sont mises à jour au début du programme :

- ***_availableSDK*** : une table de correspondance entre les noms des SDK et leur disponibilité. Les SDK sont considérés disponibles lorsque le client associé a établi une connexion.
- ***_matchingJointsMap*** : une table de correspondance entre chaque jointure et le SDK qui doit lui fournir ses informations.

D'autres informations sont mises à jour par les observateurs de gestes ou *GestureObserver* :

- ***_foldFlags*** : un vecteur contenant un booléen pour chaque doigt, de cette manière nous indiquons si ce doigt est dans une posture pliée ou non. Ces booléens peuvent ensuite être utilisés pour déterminer une pose plus complexe de la main comme le fait de pointer de l'index (index tendu mais autres doigts pliés).
- ***_pinchFlag*** : Une table de correspondance qui permet d'indiquer si une main est en posture "pincée" (*pinched* en anglais) ou non. Ce drapeau est pour le moment uniquement utilisé si le SDK ThreeGear est utilisé. Dans ce cas la détection du *pinch* utilise les messages *Pinch* provenant de ThreeGear. En effet les détections de *pinch* de ThreeGear sont plus précises que celles que nous effectuons pour le moment.

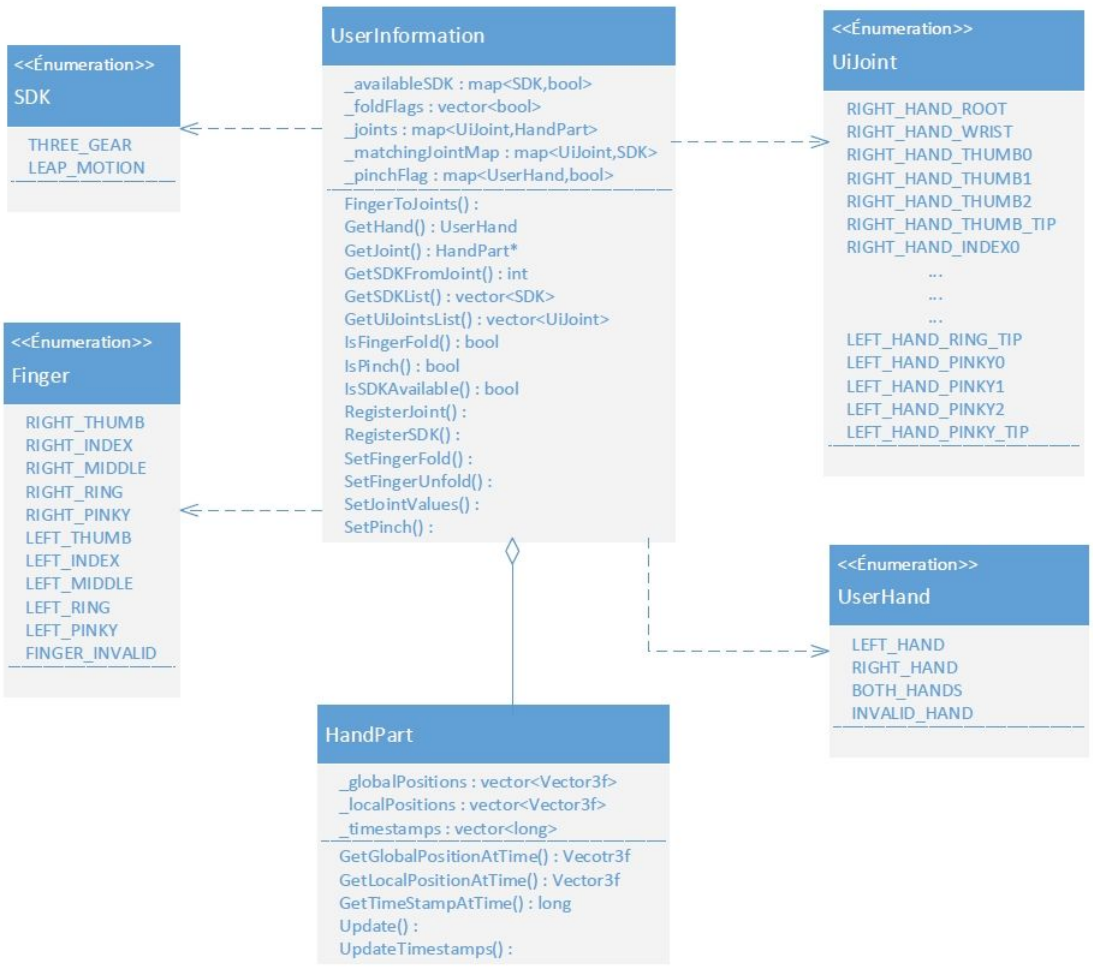

FIGURE 2.1.3 – Diagramme UML partiel : partie stockage des données

2.1.3 Troisième étape : reconnaissance de geste et routine d'envoi des données

Une fois les données de positions renseignées dans la structure de données, la détection de geste peut s'effectuer. Pour chaque geste à observer une classe d'observateur existe et utilise les données à sa portée ; si plus de gestes doivent être gérés, alors il suffit de construire une nouvelle classe dérivée de la classe virtuelle *GestureObserver*.

On peut constater sur le diagramme UML ci-après qu'il y a deux types d'observateurs de gestes :

- Des observateurs finals qui peuvent directement créer des messages et les insérer dans la queue de message pour être envoyés. C'est le cas de ***TapObserver***, ***PointingObserver*** et ***PinchObserver***.
- Des observateurs intermédiaires qui n'émettent aucun message mais mettent à jour des valeurs de la structure de données. C'est le cas de ***FoldObserver*** qui permet d'ajouter la notion de doigt plié ou déplié, ensuite utilisable dans d'autres observateurs.

Comme dit précédemment, les observateurs finals peuvent créer et insérer des objets de type ***Message*** dans une ***MessageQueue***, ces messages peuvent être de deux types :

- ***CommonMessage*** : permettent de transmettre une chaine de caractère simple sans argument. par exemple la chaine “TAP”, interprétée dans Unity comme un mouvement rapide de l'index vers le bas.
- ***ArgumentsMessage*** : comme son nom l'indique, permet de transmettre un ou plusieurs arguments à la suite de la chaine. Ainsi, *PointingObserver* peut créer un Message de type “POINTING 0.0 0.0 0.0” qui indique que l'utilisateur pointe les coordonnées globales x=0, y=0 et z=0.

Pour envoyer les messages à l'application cliente, nous avons mis en place un serveur TCP. Sa structure même ne nous permet pas d'implémenter d'observateur, de ce fait nous avons une méthode de scrutation de la file de messages. Les difficultés et limitations rencontrées lors du développement de ce serveur sont expliquées en partie 3.2.2.

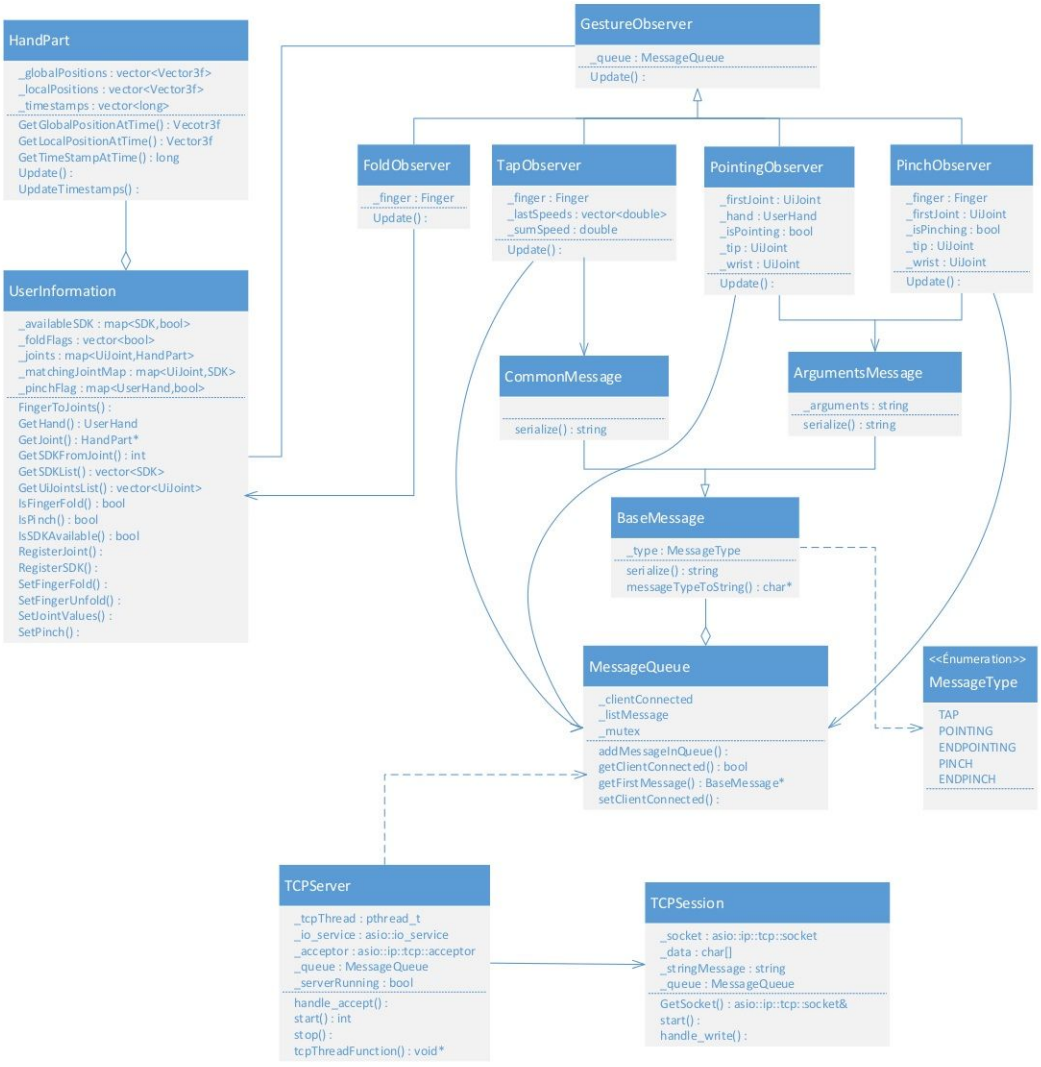

FIGURE 2.1.4 – Diagramme UML partiel : partie reconnaissance de geste et routine d'envoi des données

2.2 Approche technique : C♯ et Unity

De même que pour la partie C++, la partie développée sous Unity se décompose en plusieurs étapes. La particularité de la programmation sous Unity réside dans le fait que nous programmons des *scripts* décrivant le comportement des objets de la scène (les *GameObjects*) et non des classes pouvant s'appeler entre elles comme nous le faisons pour la partie écrite en C++. Chaque script est une classe dérivant d'une classe virtuelle propre à Unity, nommée *MonoBehaviour*, qui implémente une méthode *Start()*, appelée au lancement de l'application, et une méthode *Update()* appelée à chaque *frame*. D'autres méthodes implémentées dans *MonoBehaviour* sont également disponibles, elles sont toutes adaptées au développement d'une application graphique, ce qui simplifie grandement le développement, notamment pour des personnes dont la profession est avant tout artistique et qui ont des connaissances limitées en programmation.
Il est également possible d'écrire des classes "normales", n'héritant pas de *MonoBehaviour*, et dans ce cas elles représentent des objets classiques comme nous pourrions les définir en dehors de Unity. Cependant ces scripts ne font pas partie de la collection de scripts liables à un *GameObject* et n'implémentent qu'un objet soumis aux contraintes de la programmation orientée objet habituelle, sans aucun lien avec les particularités de Unity.
Par exemple, nous avons choisi de programmer la création et la sauvegarde du chemin sous forme de courbe de Bézier (voir section 2.2.3) dans une classe normale, *PathCreator*, ce qui est logique car il s'agit d'un objet non unique, on pourrait imaginer créer plusieurs chemins dans la même application. En revanche, cet objet *PathCreator* est créé dans un script hérité de MonoBehaviour se nommant *BallBehaviour* car il décrit le comportement de la balle présente dans la scène et dont le but est de se déplacer le long dudit chemin. Ce comportement est unique, lié à un GameObject particulier, et doit être mis à jour à chaque frame, c'est pourquoi il s'agit d'un script *MonoBehaviour*.

En réalité, il existe deux applications Unity sur deux PC différents car il est impossible de faire tourner deux applications Unity sur le même poste : dès que Unity perd le focus, l'application se met en pause.

2.2.1 Première étape : Communication et traitement des données

Puisque nous envoyons des messages depuis la brique C++, il faut les recevoir et les interpréter. Plusieurs canaux sont nécessaires pour recevoir toutes les informations :

- Un client TCP permettant de recevoir les informations sur les mains et leurs représentations 3D provenant de ThreeGear (ou d'un autre système le cas échéant)
- Un second client TCP, écoutant sur un autre port, permettant de recevoir les messages que nous envoyons depuis la brique C++. Ces messages sont actuellement : "*TAP*", "*POINTING*", "*ENDPOINTING*", "*PINCH*", "*ENDPINCH*". A chaque nouveau mouvement implémenté, un nouveau type de message est alors également à développer. Il est important de noter que contrairement à la partie développée en C++, dont le but est d'être très abstraite et de pouvoir être utilisée pour le plus d'applications possible, l'application développée sous Unity est développée dans un but unique. Autrement dit, mis à part l'algorithme d'analyse

syntaxique des messages, le code ici n'est pas particulièrement générique et est adapté à l'objectif de l'application.

- Un client UDP qui utilise le protocole *TUIO*[6]. Ce client est utilisé pour recevoir les informations d'une deuxième Kinect (si la première est celle pour le *tracking* des mains) qui envoie, elle, positions et rotations de la tête et des mains de l'utilisateur. Ces données sont utiles pour adapter l'effet de perspective à l'utilisateur. Le protocole TUIO, à la base utilisé pour transmettre les données tactiles des tablettes, a ici été détourné pour acquérir 7 données décimales : les 3 positions (x,y,z) et le quaternion de rotation (x,y,z,w). L'avantage de ce protocole est de pouvoir envoyer et mettre à jour des objets au complet et à distance.

Enfin, une communication existe aussi entre les deux applications Unity, celle-ci s'effectue grâce à la classe *Network* (spécifique à Unity) et qui implémente de façon implicite des moyens de communication réseau (ainsi on peut décider pendant l'exécution de passer d'un protocole UDP à TCP). Dans notre cas seule la position et la rotation du "curseur" (représenté par l'objet se déplaçant le long de la courbe de Bézier) de la première scène doit être transmise à la seconde, nous utilisons donc les composants *NetworkView* qui permettent de synchroniser en temps réel les positions et rotations de plusieurs objets bénéficiant du même identifiant.

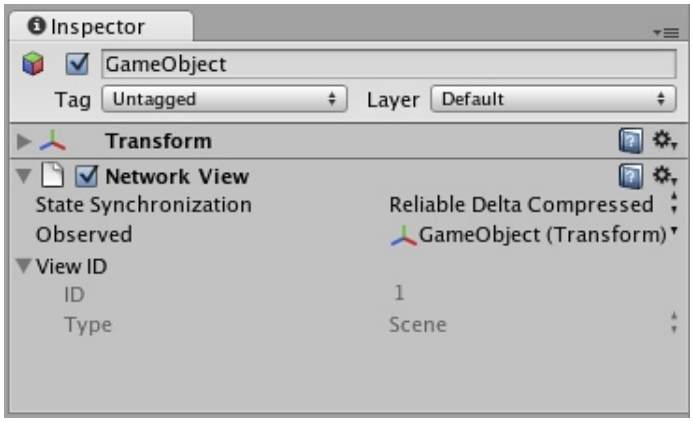

FIGURE 2.2.1 – Composant *NetworkView* attaché à un GameObject

2.2.2 Deuxième étape : Transformation des repères

Cette partie s'appuie sur le travail de thèse de Bruno De Araùjo.

Les différentes informations provenant de sources indépendantes, il est nécessaire de réaliser des transformations afin de travailler dans un repère cohérent et normalisé. Nous savons que Unity utilise la convention de repère main gauche ou *Left Handed*, ce qui n'est pas le cas de toutes les sources d'informations, ceci nous oblige à inverser les données des axes x, y et z avant la plupart des manipulations.

De plus, en raison de l'effet de perspective adaptée à l'utilisateur, il nous a fallu effectuer une transformation supplémentaire pour chaque élément de manière à les adapter au nouveau repère

6. Tangible User Interface Objects

“Table”. Le terme de repère “Table” fait référence au repère assigné par l'utilitaire de Bruno De Araùjo qui place un repère à l'emplacement physique de la table-écran. En effet, l'espace virtuel et l'espace réel sont corrélés, la table virtuelle au-dessus de laquelle la scène existe est directement associée à la table réelle qui affiche cette scène aux yeux de l'utilisateur.

Afin de modifier la perspective tout en gardant le "sol" de la scène fixe, il nous faut transformer le *frustum* de la caméra. En informatique graphique, le *frustum* correspond au champ de vision de la caméra. En utilisation normale, ce frustum est une pyramide tronquée, généralement symétrique où les plans délimitant la profondeur affichable (les plans *zNear* délimitant la profondeur minimale et *zFar* délimitant la profondeur maximale) sont parallèles.

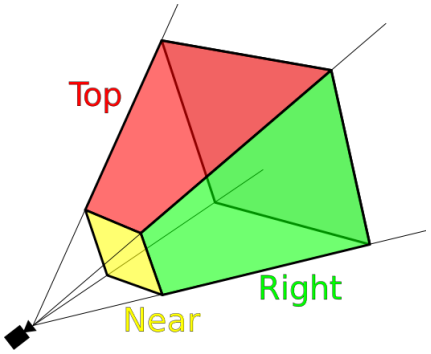

FIGURE 2.2.2 – Schéma explicatif du frustum d'une caméra

Dans notre cas, nous devons déformer ce frustum car le plan de profondeur maximale *zFar* doit toujours rester identique et correspondre au sol de la scène, alors que le reste du frustum doit bouger en fonction de la position de l'utilisateur.

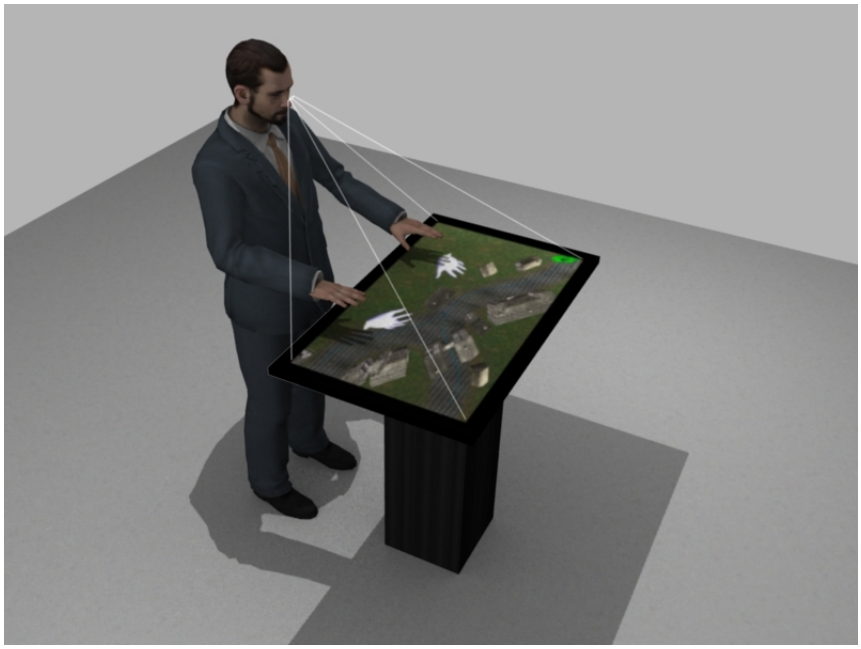

FIGURE 2.2.3 – Rendu 3D d'un utilisateur devant la table : les lignes blanches montrent la déformation à appliquer au frustum de la caméra.

2.2.3 Troisième étape : Création de la courbe de Bézier

Une courbe de Bézier est une courbe paramétrique définie uniquement par un point de départ, un point d'arrivée et un ou plusieurs points de contrôle.

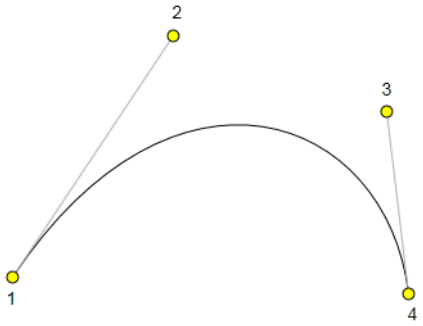

FIGURE 2.2.4 – Courbe de Bézier cubique : les 4 points de contrôle suffisent à la définir.

Dans notre projet, nous utilisons des courbes de Bézier pour définir le chemin que doit suivre le curseur à déplacer dans la scène. Nous avons choisi d'utiliser une courbe d'ordre 3 (courbe cubique), qui dispose donc de deux points de contrôle en supplément des points de départ et d'arrivée, car elles permettent d'obtenir une forme complexe avec un nombre raisonnablement limité de points.

Les seuls paramètres à passer au programme sont les coordonnées X, Y, Z des quatre points de contrôle. Ils sont à écrire dans le fichier *BezierPoints.txt* lu par l'application Unity. A partir de là, le reste des points est calculé et stocké dans une liste. La taille de l'échantillonage est également définissable. La liste de ces points permet d'effectuer tous les calculs par la suite, notamment la détection de proximité lors du déplacement de l'anneau le long du chemin. Mais ils servent également à créer la représentation graphique du chemin : en effet, à l'écran le chemin est représenté par une succession de petits cylindres dont le nombre dépend du nombre d'échantillons stockés dans la liste. Le grand nombre de cylindres et leur diamètre de taille paramétrable donnent l'impression d'une courbe continue.

3 Résultats obtenus

3.1 Résultats

3.1.1 Installation finale

Suite aux nombreux tests, nous avons pu définir un prototype d'installation.

En utilisant une table tactile, deux caméras et deux PC, nous sommes parvenus à créer la situation requise. Afin d'être dans les meilleures conditions, il est important d'avoir les deux caméras à une certaine hauteur : la caméra destinée au *tracking* des mains doit être attachée à un mât à environ 1,20 m au-dessus de la table et alignée avec le milieu de celle-ci, afin d'avoir un champ de vision assez large pour éviter de perdre le suivi des mains si l'utilisateur s'approche trop près des bords de la table. Le pied du mât doit être excentré pour ne pas bloquer le champ

de vision de la deuxième caméra. Cette dernière doit se situer à environ 1,50 m derrière la table, ainsi qu'à une hauteur d'environ 1 m au-dessus de la table, car il faut détecter correctement les 4 coins de la table ainsi que la tête de l'utilisateur.

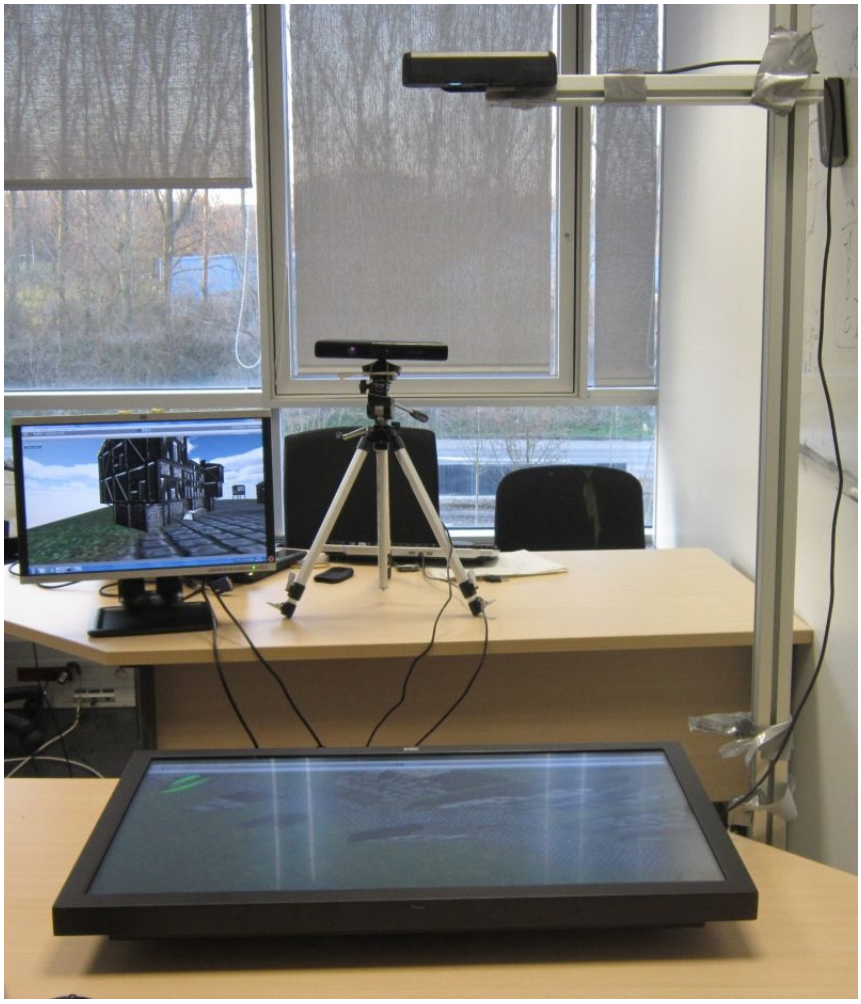

FIGURE 3.1.1 – Prototype d'installation

3.1.2 Les gestes reconnus et actions qui y sont liées

Actuellement, seule la configuration essentielle est disponible, c'est-à-dire que notre scène Unity reconnait correctement le geste de *Pointing* sur l'une ou l'autre main. Ce geste permet de déplacer un curseur le long d'une courbe paramétrique dans la scène. Les gestes de *Pinch* et *Tap* sont néanmoins opérationnels et utilisables à tout moment dans la scène. On pourrait par exemple imaginer que le *Pinch* simultané des deux mains permette de changer l'échelle de la scène pour avoir une vue d'ensemble ou une vue plus spécifique d'une zone.

3.1.3 Doxygen

Pour mettre en place une documentation structurée et utilisable pour les personnes qui travailleront plus tard sur notre projet, nous avons décidé d'utiliser la convention Doxygen. Cette

convention permet d'auto-générer une documentation (en html ou PDF) à partir de commentaires suivant une notation précise présents dans le code source.

3.2 Limitations et difficultés

3.2.1 Fuites de mémoire

Contrairement à d'autres langage comme Java ou C♯ qui disposent du principe du ramasse-miettes (*garbage collector*), autorisant le programmeur de laisser à la machine le soin de s'occuper de la gestion de la mémoire, programmer en C++ nécessite de toujours avoir à l'esprit la notion de libération de la mémoire, en particulier lorsque nous créons des objets avec l'opérateur **new** permettant d'allouer la mémoire sur le tas, qui doit toujours être accompagné d'un **delete** autre part dans le code. Dans notre cas nous avons toujours tenté de suivre cette règle afin d'éviter les fuites de mémoire. Pourtant, nous avons remarqué en utilisant des outils de détection de fuites de mémoire (en particulier *Visual Leak Detector*, car malheureusement *valgrind* n'est pas disponible sur Windows), qu'à chaque exécution de notre programme, quelques octets voire kilo-octets ne sont pas proprement libérés. La cause nous est inconnue car nous libérons, ou en tout cas nous pensons libérer la mémoire proprement. Par manque de temps il nous est impossible de pousser plus loin les investigations. Bien que la taille de la fuite ne semble pas importante comparativement aux systèmes actuels, le problème s'avère suffisamment important pour qu'il soit indiqué ici. Cette fuite reste non négligeable et est le témoin d'un programme non terminé.

3.2.2 Serveur TCP

Afin de pouvoir envoyer nos messages à l'application graphique, il nous faut mettre en place un serveur pour lequel l'application graphique se comporte comme un client. Le choix du protocole étant laissé libre, nous avons préféré garder le protocole déjà utilisé par ThreeGear et dont le script de réception était déjà écrit dans une ancienne application Unity : nous avons donc décidé de développer un serveur TCP, en utilisant la bibliothèque *Asio*. Cependant, contrairement un protocole UDP qui ne nécessite pas de connexion, le protocole TCP nécessite l'établissement d'une connexion, ce qui est plus complexe à gérer au niveau serveur car il faut garder en mémoire la session. En contrepartie, le protocole TCP est plus fiable et nous sommes assurés que le client reçoit tous les messages émis par le serveur.
La difficulté que nous avons rencontrée se situe sur le fait que les méthodes de la bibliothèque Asio sont plutôt adaptées pour implémenter un serveur répondant à une requête d'un client, et non pour envoyer des messages dont la fréquence est variable et non prévisible. De plus Asio recommande d'utiliser des sockets asynchrones, autrement dits non bloquantes : l'appel de la fonction d'écriture sur la socket est accompagné d'une fonction de *bind* (fonction attachée, appelée à la suite de la fonction d'écriture). Ces deux constatations combinées, il nous a fallu trouver un moyen de placer la socket en état d'attente d'un message à envoyer. Nous avons implémenté la manière la plus simple : une boucle d'attente qui vérifie si un nouveau message est disponible ; si oui, il est envoyé, et dans le cas contraire la fonction est mise en pause pendant un certain temps (fonction *sleep()*) avant de revérifier la disponibilité d'un message. Cependant si pendant un temps assez

long aucun message n'est à envoyer, et si dans cette période le client s'est déconnecté, nous nous retrouvions coincés dans la boucle sans pouvoir vérifier que le client est toujours disponible. C'est pourquoi nous avons ajouté un message envoyé à période fixe, qui simule un "battement de cœur" (*heartbeat*) afin de vérifier que le client est toujours connecté. Ainsi on peut détruire la socket et se mettre en attente d'un autre client si le précédent s'est déconnecté.

3.2.3 Utilisation de Unity

Unity est un outil de développement de jeux, avant tout pensé pour aider des créateurs de jeux dont le domaine est plus l'art que la programmation. C'est pourquoi il dispose d'un moteur graphique et d'un moteur physique plutôt complets et simples d'utilisation, et allie une interface de programmation de scripts avec une interface *WYSIWYG*[7] simplifiant la création des scènes 3D et la hiérarchisation des objets de la scène. Cependant, cette interface *WYSIWYG* est limitante pour nous qui sommes avant tout programmeurs : le principe des scripts décrivant le comportement des GameObjects sans possibilité d'avoir une "vue d'ensemble" de la scène complète rend plus difficile le développement de scripts complexes, comme par exemple les nombreux calculs de changement de repère que nous effectuons. Ainsi, cette interface est très pratique pour développer en quelques clics une scène simple avec des interactions basiques, mais devient une limitation lorsque l'on a des objets dont les comportements sont complexes et dépendants entre eux, ou que l'on souhaite avoir accès au cœur de l'application, ce qui est quasiment impossible car la grande majorité des fonctionnalités des moteurs graphique et physique est cachée, Unity fonctionnant sur le principe de la boîte noire.

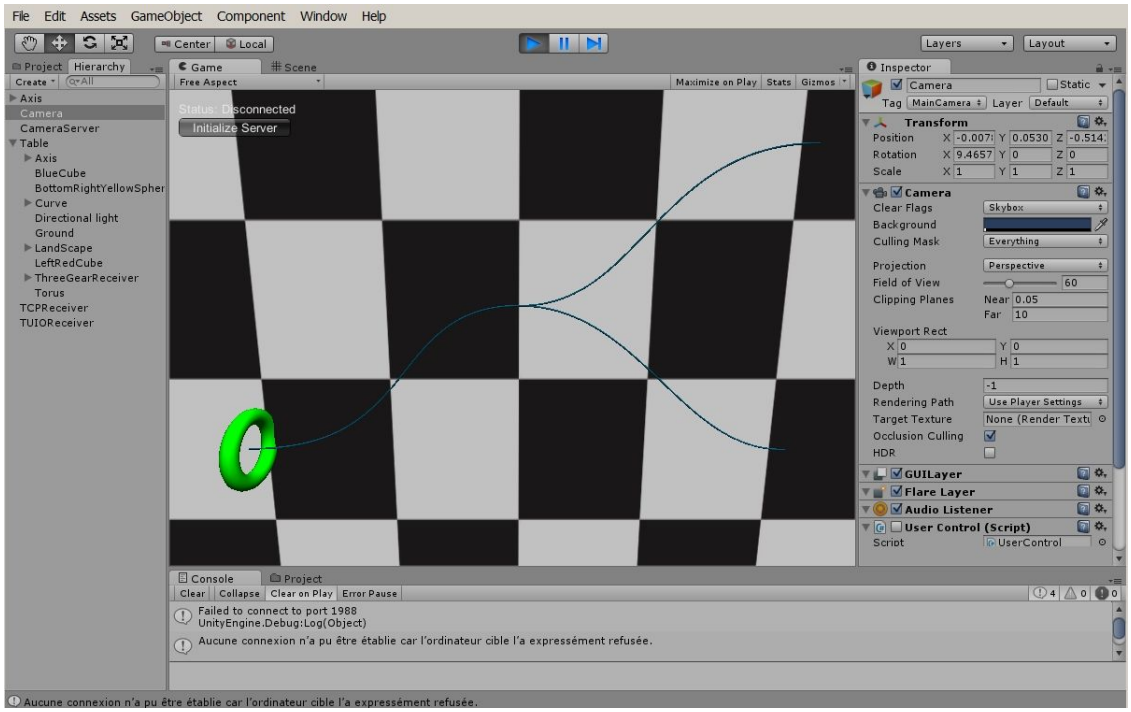

FIGURE 3.2.1 – Interface utilisateur de Unity en mode "développement"

7. What You See Is What You Get, ou interface permettant de travailler en ayant directement une idée du résultat final.

3.2.4 Limitations techniques

Les gestes

Comme nous l'avons déjà expliqué, toutes les reconnaissances de gestes prévues à l'origine n'ont pas pu être intégrées. Le nombre de gestes est ainsi pour l'instant limité, et si l'on veut utiliser notre application pour un autre projet, il serait nécessaire de développer de nouvelles reconnaissances.

Ordre d'exécution des scripts

Unity fonctionnant sur le principe de la boîte noire, nous n'avons que très peu d'accès aux threads créés par le logiciel pour pouvoir gérer l'exécution du grand nombre de scripts développés. Ces scripts partagent plusieurs ressources, en particulier les matrices de transformation du repère global au repère local à la table tactile. Il faut donc un moyen pour indiquer quel script doit être exécuté en premier, car les scripts doivent récupérer des informations dépendantes du résultat d'autres scripts. Il faut donc définir un ordre d'exécution. Pour cela, Unity dispose d'un gestionnaire d'ordre d'exécution des scripts. La difficulté revient alors à déterminer cet ordre de manière à éviter tout conflit tout en gardant un système rapide.

Installation à plusieurs caméras

Étant donné que nous utilisons deux caméras Kinect différentes permettant de récupérer deux types d'informations différents, il a fallu vérifier que les deux caméras ne se parasitent pas. Or, la détection de la profondeur se fait par infrarouge. Nous avons remarqué que la Kinect prévue pour suivre les mains de l'utilisateur pollue la détection de la seconde Kinect à cause de ces infrarouges : la Kinect prévue pour suivre les mains empêche l'autre de détecter convenablement la table tactile, les infrarouges de la première reflètent sur la table et faussent complètement la phase de calibration. Il a donc fallu convenir d'un protocole de calibration empêchant ce phénomène :

- **[Indispensable]** - Calibrer la Kinect de détection de la tête avant d'alimenter la caméra de détection des mains.
- **[Recommandé]** - Démarrer la calibration de ThreeGear avec la Kinect de détection de la tête éteinte.

3.3 Prévisions

3.3.1 Gestes prévus

Initialement, notre projet devait porter sur le développement de plus de gestes. Mais étant donnée la place qu'a prise l'application Unity, nous n'avons pu gérer que les gestes indispensables. C'est pourquoi, si notre application est réutilisée pour un autre projet, il sera nécessaire de reconnaitre d'autres gestes. Ainsi, étaient prévus :

- la reconnaissance d'un "tap" de la main, en supplément du "tap" du doigt, qui pourrait servir à la validation d'un menu ou d'une action.
- la reconnaissance d'un mouvement de balayage de la main, horizontalement ou verticalement, ce qui serait utile pour la navigation dans un menu par exemple.

En supplément de ces gestes initialement prévus, on peut imaginer toute une multitude d'autres gestes, à définir en fonction des interactions prévues. Pour voir plus loin, on peut également penser à des algorithmes qui requièrent plus de difficultés : reconnaissance de formes géométriques comme un cercle par exemple, qui nécessite l'utilisation et la comparaison d'un historique de points. Un tel algorithme est complexe car contrairement à une machine, un utilisateur humain est incapable d'effectuer un cercle parfait, ainsi il devient difficile de comparer la forme dessinée avec une forme géométrique parfaite.

3.3.2 Capteurs prévus

Pour le moment seule une Kinect avec le logiciel ThreeGear peuvent communiquer avec notre application, nous avions prévu d'ajouter un module *Leap Motion* aux choix disponibles, mais faute de temps ce ne sera pas possible. Pour rappel, le Leap Motion est un petit appareil qui utilise la technologie infrarouge pour détecter les mains et les doigts d'un utilisateur dans un faible rayon d'action (moins de 60 cm).

L'ajout de ce type de capteur n'est pas prévu pour l'utilisation grand public et n'aurait servi qu'à nous assurer du bon fonctionnement de l'application dans un environnement différent.

Une alternative utilisable pour une application grand public serait peut-être le *Gametrack*, petit dispositif qui relie le doigt de l'utilisateur avec un capteur fixe et qui permet de définir, avec une bonne précision, la position d'un point dans l'espace. En soutien d'une Kinect, cette solution serait plus précise et donc plus fiable mais le contact avec le public pendant une longue durée pourrait l'endommager.

Enfin, pour gagner de la précision sans changer drastiquement les moyens de captation nous pourrions utiliser la seconde génération de Kinect qui possède une plus haute définition.

3.3.3 Améliorations

Pour que l'effet de 3D sans stéréoscopie soit optimal, il faut effectuer un calibrage de la seconde Kinect de manière à définir les bords de la table-écran dans l'espace virtuel. Ce calibrage n'est pas encore au point, pour le moment nous adaptons les coordonnées des bords à ce que nous nous attendons à avoir, mais la marge d'erreur fait que le rendu n'est pas parfaitement authentique. Dans un premier temps il faudrait régler ce problème, dans un deuxième temps il faudrait parvenir à faire fonctionner tous les appareils sur le même ordinateur, ce qui améliorerait sans doute la réactivité, et simplifierait l'installation.

La mise en place d'interactions homme-machine est loin d'être une tâche à prendre à la légère, ceci est d'autant plus vrai lorsqu'il s'agit de réalité virtuelle n'impliquant aucun périphérique standard (clavier, souris ...).

Pendant ces six derniers mois nous avons développé une application C++ en partant pratiquement de zéro, nous avons appris à utiliser le logiciel Unity et avons acquis des compétences en matière de manipulation géométrique 3D et en structuration de code. Le but était de pouvoir proposer à un public un outil intuitif, fiable et robuste qui lui permette de manipuler et explorer un univers 3D.

Aujourd'hui la solution que nous proposons n'est pas une application finale, un certain nombre de gestes peuvent être ajoutés et d'autres systèmes de captation peuvent être intégrés. Cependant, le but même de cette application était aussi de permettre une adaptabilité aux futures améliorations. S'appuyant sur le principe de la modularité, notre solution peut facilement être reprise et augmentée, afin de l'adapter à un autre type d'application.

Quant à l'application graphique développée sous Unity, nous proposons les technologies et les solutions nécessaires à la réalisation du projet lancé en collaboration avec une artiste, en nous appuyant sur une partie du travail de thèse de Bruno De Araùjo, chercheur postdoctoral dans l'équipe MINT.

Annexe A : Diagramme Gantt du projet