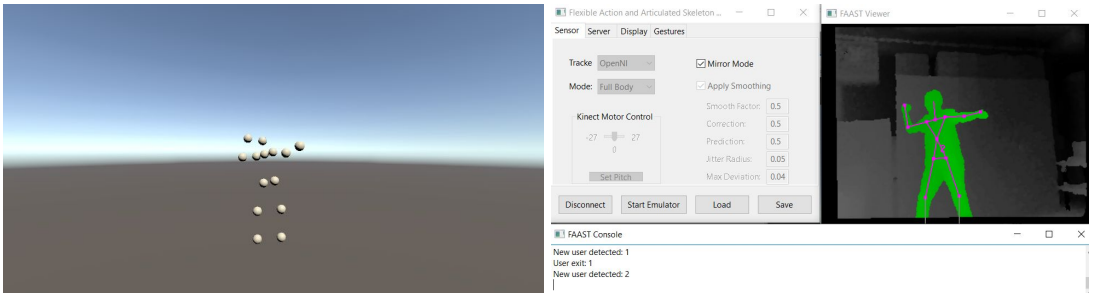
Ali Zahidi p1713059 (a.alizahidi@gmail.com)
Robin Donnay p1510329 (donnay.robin@gmail.com)

Rapport Technologies Embarquées

HTC Vive et Asus Xtion
KiVe Project (https://github.com/Aros69/Kive)

Concept initial

Intégrer le squelette du joueur dans une application de réalité virtuelle pour rajouter du réalisme.



A gauche : les points du squelette détectés par FAAST et envoyés sur Unity
A droite : le logiciel FAAST et sa représentation de la zone de jeu avec le squelette du joueur

Récupération du squelette

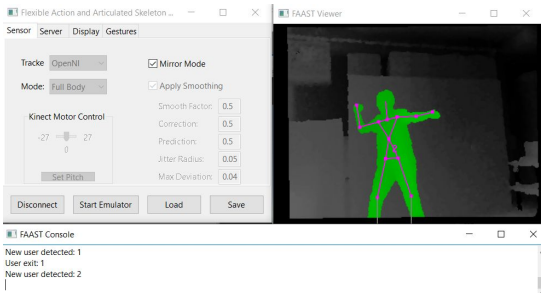
La récupération de données, notamment le squelette, via un système Kinect / Asus Xtion a déjà été traitée par de nombreux développeurs. Plusieurs SDK sont déja disponibles comme OpenNI (https://structure.io/openni) et le SDK Kinect (https://developer.microsoft.com/fr-fr/windows/kinect/). Ces SDK donnent un accès bas niveau aux données renvoyées par les dispositifs tel que le Asus Xtion. On obtiendra notamment une carte de profondeur grâce au capteur infrarouge du matériel. Nos besoins étant assez génériques (récupération du squelette) nous avons pu utiliser un middleware développé par des chercheurs de l'université de Californie du Sud. Flexible Action and Articulated Skeleton Toolkit : FAAST (http://projects.ict.usc.edu/mxr/faast/) permet de récupérer plusieurs squelettes humain présents face à la caméra et de les diffuser via un réseau VRPN afin que d'autres applications puissent s'en servir. Sa mise en place est très facile et fonctionne aussi bien avec Kinect qu'avec un Asus Xtion. Le résultat obtenu est décrit par les images ci-dessus.

Transfert du squelette vers l'application principale

L'une des raisons principales de l'utilisation de FAAST est son VRPN (Virtual-Reality Peripheral Network). Celui-ci permet de transmettre par réseau ce qu'il capte, à savoir un ensemble de points clés formant le squelette d'une personne. Il est donc très facile de récupérer ces points sur un autre logiciel si celui-ci possède un réseau VRPN. De plus, un plugin Unity permettant de communiquer avec un serveur VRPN existe déjà (https://github.com/hendrik-schulte/UVRPN). Le lien avec ces deux applications étant fait, il faut savoir quels points correspondent à quelles "articulations" dans le squelette.

Voici les points normalement capté par FAAST :

Sensor	Joint	Sensor	Joint	Sensor	Joint	Sensor	Joint
0	Head	6	Left Elbow	12	Right Elbow	18	Left Ankle
1	Neck	7	Left Wrist	13	Right Wrist	19	Left Foot
2	Torso	8	Left Hand	14	Right Hand	20	Right Hip
3	Waist	9	Left Fingertip	15	Right Fingertip	21	Right Knee
4	Left Collar	10	Right Collar	16	Left Hip	22	Right Ankle
5	Left Shoulder	11	Right Shoulder	17	Left Knee	23	Right Foot



Sur la photo ci dessus, on peut voir qu'il y a moins de points qu'annoncé par la documentation FAAST. Nous pensons que les articulations détectés peuvent différer selon la caméra utilisée. Après quelques tests sur Unity avec le capteur Asus Xtion, nous avons pu déduire le tableau de correspondance suivant :

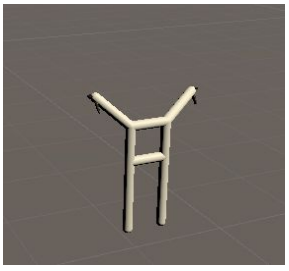
Sensor	Joint	Sensor	Joint	Sensor	Joint	Sensor	Joint
0	Head	7	Left Hand	12	Left Hip	17	Right Knee
1	Neck	8	Right Shoulder	13	Left Knee	18	Right Foot
4	Left Shoulder	9	Right Elbow	14	Left Foot		
6	Left Elbow	11	Right Hand	16	Right Hip		

Le transfert entre application est pour le moment local mais il est possible d'utiliser le VRPN sur d'autres machines et donc de séparer sur deux machines différentes l'application de réalité virtuel et la captation du squelette. Nous y reviendrons dans la section de problèmes rencontrés mais nous n'avons pas réussis à utiliser le VRPN sur deux machines distinctes. Nous avons donc utilisé Mirror (https://github.com/vis2k/Mirror) combiné au VRPN pour diffuser l'information entre le PC captant le squelette et l'ordinateur utilisant Unity. Nous expliquerons cela dans la partie "Structure du projet".

Reconstruction

Reconstruction Naïve

Actuellement les jeux vidéos et les applications VR utilisent la cinématique inverse pour reconstruire le corps du joueur dans l'environnement virtuel. Cette méthode est simple et efficace mais nécessite un certains temps d'implémentation. Notre implémentation naïve est extrêmement simple et "fausse". Aucune articulation du corps ne bouge (à part la tête et les mains) mais cela permet de conserver la physique et la représentation graphique du corps (sans la tête et le cou pour éviter les problèmes d'affichage avec la caméra VR) lorsque la reconstruction n'est pas possible.



Reconstruction naïve du corps

Transformation linéaire

La première approche que nous avons décidé d'essayer est le calcul de la transformation linéaire qui permet de passer du repère de la caméra Kinect, au repère du HTC Vive. Pour cela nous avons utilisé les positions associés au casque et aux manettes HTC Vive, et les positions des articulations de la tête et des mains. Ces positions étant approximativement les mêmes. Nous avons ensuite résolu le système d'équations linéaires sous forme matricielle.

$$K \,.\, T \;=\; H$$

K : Matrice des positions Kinect

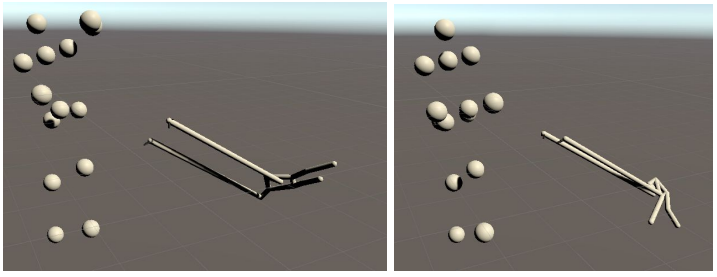
T : Transformation linéaire

H : Matrice des positions HTC Vive

La matrice de transformation T peut donc être obtenu en multipliant H par l'inverse de K :

$$T \;=\; K^{-1} \,.\, H$$

Il suffit ensuite d'appliquer la transformation aux positions des articulations Kinect pour obtenir les positions dans le repère d'Unity. Les résultats obtenus avec cette méthode sont cependant erronées comme le montre les images ci-dessous.

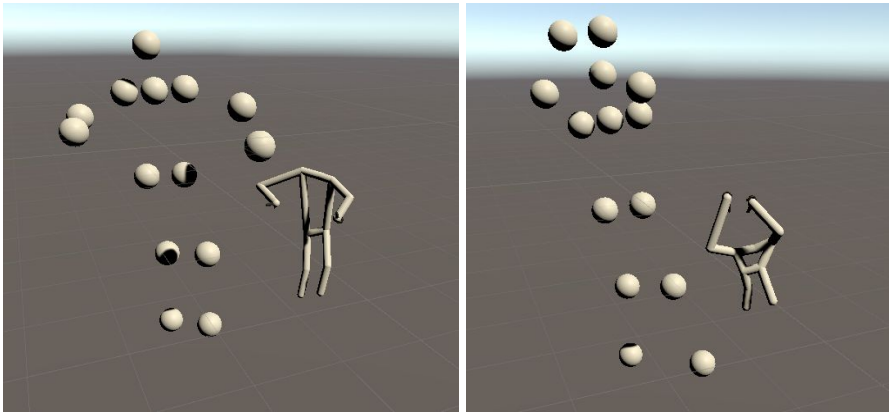


Représentation de la captation Asus (gauche de chaque image) et de la reconstruction linéaire (droite de chaque image)

Calculs direct

Pour cette méthode nous utilisons la distance entre le casque et la manette droite HTC ainsi que la distance entre l'articulation de la tête et de la main droite pour calculer le rapport permettant de convertir une distance dans le repère Kinect vers le repère HTC. Nous utilisons ensuite les directions/distances que l'on récupère dans le repère K et que l'on applique ensuite au repère H.

Nous faisons ensuite la somme pondérée qui nous donnera la position de l'articulation voulue. Les résultats obtenus sont plus intéressants que les résultats de la premières méthode. Néanmoins quelques ajustements restent à faire, notamment l'orientation et la conservation de la taille du corps.



Représentation de la captation Asus (gauche de chaque image) et de la reconstruction par calcul direct (droite de chaque image)

Structure du projet

Le projet, en plus du logiciel FAAST, utilise 2 packages importants : UVRPN (pour récupérer les données produite par FAAST envoyé par le VRPN) et SteamVR (package simplifiant grandement la VR pour le HTC Vive sur Unity). De plus, pour synchroniser la communication entre les machines on utilise Mirror qui est une API réseau pour Unity.

Prefabs clés

- SkeletonContainer : L'objet auquel sont parenté les joints (les articulations) qui vont être transmis par le réseau, c'est à partir de ce prefab que l'on pourra visualiser le squelette récupérer par la caméra Kinect sur le client et le serveur.
- SkeletonData : Contient des fonctions utilitaires permettant de récupérer les joints sur l'ordinateur connecté à la caméra kinect. Ce prefab s'occupe aussi de l'initialisation et de la mise à jour des positions des joints grâce au package UVRPN.
- Network Manager : ce prefab gère entièrement la couche réseau entre les machines, de la recherche d'autres joueurs à la création des instances des joueurs… Ce prefab est essentiel dans toute les scenes. Le seul point sur lequel il faut être attentif sur ce prefab est que le component "Network Manager (Script)" doit avoir comme "Player Prefab" le prefab "Skeleton Data" sinon les données du squelette ne seront pas envoyées vers le PC connecté au casque VR.

Les autres prefabs présents dans le projet ne représentent que des prefabs de gameplay, de visualisation pour les tests

Scripts clés

- KiveSkeleton représente le squelette du joueur. On peut y trouver la structure du squelette (os et articulations) mais aussi les positions du casque et des manettes dans l'espace. A chaque instant il met à jour tout le squelette avec ou sans l'aide du Kinect. S'il y a un lien avec une caméra Kinect alors il fera une des deux reconstructions présentées plus tôt sinon il fera la reconstruction naïve.
    - La fonction `asICanSkeletonUpdate` correspond à la reconstruction très naïve et n'a besoin que des positions du HTC Vive. Son seul intérêt est d'éviter des problèmes en cas de pertes de connexion avec Kinect.

    - La fonction `kinectSkeletonUpdate` utilise à l'heure actuelle le reconstruction par calcul direct mais calcul aussi la matrice de passage de la reconstruction par transformation linéaire pour faciliter le passage d'une méthode de reconstruction à une autre.
- Calibrator correspond à la reconstruction par transformation linéaire ou par calcul direct.
    - La fonction `computeKive` correspond au calcul de la matrice passage du repère Kinect au repère Vive. Les trois paramètres sont la position dans le repère Vive du casque et des manettes, les trois suivants sont les positions dans le repère Kinect de ces trois points. La matrice de passage est stockée dans la matrice `KiveMat`
    - La fonction `getViveJoint` prend en paramètre un point dans l'espace Kinect et calcul grâce aux positions du casque, des manettes, de la tête et des mains afin de définir sa position dans le repère Vive. On notera qu'il faut au préalable avoir mis à jour dans le calibrator les positions de chacun des six points énoncés précédemment pour que le calcul direct soit lui aussi mis à jour.
- SkeletonContainer est le script permettant la communication entre les deux ordinateurs. Ce script a un comportement différent selon si la machine est le serveur (ordinateur branché à la caméra Kinect) ou le client (ordinateur branché au casque VR). Dans le premier cas on initialise tous les joints avec une position et une rotation et on met à jour la position des joints à chaque frame en fonction des données reçu via la VRPN et envoyé par FAAST. Dans le cas du client, on ne fait que mettre à jour le tableau des joints locaux avec les positions communiquées par le réseau Mirror.

Les autres scripts présents dans le projet ne représentent que des scripts de gameplay, de visualisation pour les tests et aussi des fonctionnalités abandonnées (comme l'intégration du squelette sur un vrai modèle humanoïde 3D tiré du site Mixamo)

Étapes pour tester le projet

Pour tester le projet, il est fortement conseillé d'utiliser deux ordinateurs, nous n'avons pas réalisé de test avec uniquement un ordinateur. Il est très important que les deux ordinateurs soit sur le même réseau. Nous vous conseillons même d'utiliser un réseau créé en local ou un partage de connexion (via un smartphone) afin d'éviter que les ports de communication entre les deux appareils soit bloqués (Mirror utilise les ports 7777 et 47777).

Il faut au préalable lancer l'application FAAST sur le premier ordinateur et connecter la caméra Kinect à ce dernier. Il faut ensuite, sur la fenêtre de FAAST, choisir le tracker "OpenNI" et appuyer sur "Connect". Si le programme détecte la caméra correctement, le FAAST Viewer va afficher les images de profondeurs capturées par la caméra, ainsi que les squelettes des personnes se trouvant dans le champs de vision de la caméra.

L'ordinateur utilisant FAAST doit avoir un build du projet et ne doit donc pas lancer le jeu via l'éditeur Unity sinon Mirror ne trouvera pas l'autre ordinateur. De plus il doit lancer sa partie en mode "Server Only", en cliquant sur le bouton "Start Server", car son seul objectif est d'envoyer les informations à l'autre ordinateur. Par conséquent, peu importe la scène buildé du moment qu'elle possède un network manager (la scène "vrtest" du dossier "Scenes" est la plus simple des scènes à lancer pour le serveur).

L'ordinateur utilisant la VR peut lui lancer le jeu depuis l'éditeur Unity (une fois que le casque VR est branché et prêt à être utilisé). Quelque soit la scène qu'il choisira, il devra avant toute chose cliquer sur le bouton "Find Server" et choisir l'adresse IP correspondant à la machine utilisant FAAST (normalement il ne devrait y avoir qu'une adresse IP qui apparaît). Deux scènes sont jouables pour le joueur :

- La scène “Test” du dossier “Scenes” possédant un “vrai“ miroir pour se voir et un mini jeu dans lequel (en utilisant le pad des manettes) on fait apparaître des objets qui doivent toucher leurs homologues plus grand et tout cela en utilisant son corps pour les pousser.
- La scène “Level01” du dossier “Scenes/Level” ou l’objectif est d’écraser des cafards qui vous tourne autour en les touchant. Il y a cependant un problème lié au positionnement du squelette qui fait que le joueur peut se retrouver à l'intérieur du sol. Une amélioration possible serait de fixer la position du joueur au niveau du sol et de lui donner la possibilité de se déplacer avec les joysticks du contrôleur VIVE.

Problèmes rencontrés

Voici une liste des différents problèmes rencontrés au cours du projet qui ont nécessités soit un changement de cap soit une diminution de la qualité espérée du projet :

- Problème de communication entre machine en utilisant le VRPN. L’utilisation en locale du VRPN fonctionne, cependant dés que le VRPN essaie de communiquer avec une autre machine cela ne marche plus. Au lancement d’Unity (et du package UVRPN) avec une adresse réseau non local, le logiciel FAAST se bloque et n’envoie plus les informations.
- Le miroir présent dans la scène “Test” donne deux visuelles différents pour les deux “yeux” du casque HTC Vive. Nous pensons que la cause est dû au fait que le shader utilisé dépend du point de vue de la caméra, qui est donc différent pour chaque “oeil” du casque HTC Vive. Nous avons essayé de chercher un moyen de faire en sorte que le point de vue pris en compte par le shader soit différent de la caméra du joueur, et que donc le miroir soit “unifié” mais les changements à apporter été trop profond dans le moteur. Nous avions donc décidé de nous concentrer sur d’autres aspects du projet.
- La reconstruction par transformation linéaire semble être à la fois la solution la plus exacte et la plus rapide en terme de calcul. Cependant les résultats obtenus sont erronés. Est-ce par manque de nombre de points (3 étant le minimum nécessaire, est-ce qu’avec un point de plus les résultat serait-il plus correct ?)
- La reconstruction par calcul direct offre une approximation plus que satisfaisante en un temps correct, ici nous ne nous sommes intéressés qu’à la position dans l’espace des points or nous possédons aussi leur orientation. Une amélioration logique serait donc d’utiliser les orientations au lieu des positions, cela garantit un squelette qui ne partirait pas dans tous les sens.