

TP Kinect v2 sous Unity 3d

JIN 2017-2018

Frédéric Davesne

Objectifs:

- Savoir récupérer les données de la Kinect sous Unity 3d et la piloter
 - à partir d'une librairie C# sous Unity3d interfaçant le SDK Kinect Microsoft;
 - comme client VRPN (Virtual Reality Peripheral Network) de l'application FFAST¹.
- Qualifier les données récupérées.
- Animer un ou des objets graphiques Unity à partir des données de la Kinect
 - à partir des positions des différents éléments du squelette (vecteurs 3D) ;
 - à partir des rotations d'un élément du squelette par rapport à son "père" dans le graphe de scène (quaternions -> vecteurs 4D) .
- Coder sous Unity les techniques d'I3D suivantes indépendamment du périphérique de RV:
 - technique de sélection/manipulation Main Virtuelle Simple (MVS);
 - technique de sélection RayCasting ;
 - technique de navigation Direction de la main ;
- Utiliser un point d'entrée matériel unique ayant comme sortie un vecteur3d qui peut être injecté à chacune des trois techniques d'I3D.

1. Introduction (voir le fichier Documentation.pdf)

1.1. Essai de la Kinect en dehors de Unity

a. Lancer l'application FFAST (Flexible Action and Articulated Skeleton Toolkit) avec *Display->Background Pixels: RGB*, puis *Connect*.

b. Placez-vous devant la Kinect. Lorsque votre corps (ou une partie) est reconnu, celui-ci est tracké et un squelette correspondant à différentes articulations du corps apparaît en surimpression de l'image RGB/DEPTH.

c. Cliquer sur *Server* et remarquer l'appariement d'un 'squelette' avec l'image RGB/DEPTH lorsque *Tracker0* est associé à un identifiant.

1.2. Production d'un événement à partir d'un geste capté par FFAST puis utilisé par Unity
FFAST permet de générer des événements simples (clavier/souris) à partir de la détection de positions/vitesses/mouvements du squelette lorsque la personne est trackée par la kinect.

a. Emuler la touche 'a' du clavier

- Aller dans *Gestures -> New Gesture -> Add -> Mode Trigger Once -> Add position constraint -> right hand -> To the right of -> Torso -> 30 cm at least* et associer ce geste avec l'appuie de la touche 'a'

- Sauver le mouvement et *Start Emulator*

- Ouvrir une console de commande et essayer lorsque cette console est active ...

¹ Voir <http://projects.ict.usc.edu/mxr/faast/>

- Faire la même chose avec *Gestures* -> *New Gesture* -> *Add* -> *Mode Loop repeatedly* -> *Add Velocity Constraint* -> *right hand* -> *to the right* -> *at least 10 cm/sec*

b. Créer un projet Unity *Essai Kinect*

- Créer un répertoire *Script/Kinect/faast* dans lequel vous allez créer un fichier C# *MoveFromKeyboard.cs*.

c. Créer un *GameObject* Cube

d. Dans le script *MoveFromKeyboard.cs*, récupérer la touche clavier 'a' et déplacer le cube de 0.1 m sur l'axe X.

e. Constatez que le cube se translate lorsque vous effectuez le mouvement adéquat.

Voir Scène FAAST.unity pour la correction

D'après vous, quelles sont les limitations de cette approche?

Manque de précision!!!

2. Utilisation d'un wrapper Kinect v2 pour Unity.

a. Décompresser le package *Premier Essai Kinect v2.unitypackage*

b. Regarder la scène *Première Scène* et cliquer sur *KinectController*. Celui-ci fait l'interface entre les *GameObjects* de la scène courante et les données provenant de la Kinect, car il contient le script *KinectManager.cs* associés à l'interface Kinect. Le paramètre *DisplayUserMap* du script *KinectManager.cs* permet en particulier de visualiser les mouvements de la personnes.

c. Créer un *EmptyObject* *body* puis créer un *GameObject* *main_droite* de type Sphère de rayon 0.1 mètre, dépendant hiérarchiquement de *body*.

d. Associer le déplacement de la sphère de la scène avec le déplacement de la main droite. Pour cela, utiliser le fichier exemple *GetJointPositionDemo.cs* de *KinectScripts/Samples/* pour créer votre script *bouge_main_droite.cs*.

Nous créons un *Empty Object Tracker* dont l'objectif sera de tracker en position la main droite grâce au script *GetJointPosition.cs* inspiré du script *GetJointPositionDemo.cs* qui a comme paramètre d'entrée l'articulation à tracker (ici *Hand Right*) et comme sortie le *Vector3 jointPosition*. Le script *bouge_main_droite.cs*, appliqué au *GameObject main_droite*, utilise la variable *jointPosition* de *Tracker* pour mettre à jour la position de *main_droite*.

e. Créer une classe C# *TraceData.cs* qui trace les données en position de la sphère en fonction du temps et qui les place dans un fichier texte *Trace.txt*. Chaque ligne sera de la forme 'temps' 'x' 'y' 'z'. Tracez la position de votre main droite et regarder le fichier créé. Que constatez-vous?

TraceData.cs est placé sous l'*Empty Object Tracker*. Il récupère des données de la Kinect à chaque frame graphique, y compris lorsque la main droite n'est pas trackée ou qu'un nouveau frame de la Kinect n'est pas encore parvenu !!!

f. Grâce à cette trace de données, essayez de calculer la fréquence des données ainsi qu'une précision approximative de celles-ci. Pour cela, vous pouvez par exemple utiliser Excel sur la première colonne du fichier *Trace.txt*.

On constate en particulier que des lignes identiques peuvent être créées et que le frame rate est très supérieur au 30 Hz donné par le constructeur. Cela est dû au fait qu'il faut attendre qu'un nouveau frame de la Kinect soit reçu. Cette information est disponible en utilisant la fonction `manager.GetBodyFrameTimestamp()`. En faisant attention cette fois-ci à n'écrire dans le fichier *Trace.txt* uniquement lorsqu'un nouveau frame de la Kinect est disponible, nous obtenons une moyenne de l'écart temporel entre deux frames est de 0.0369 secondes soit une fréquence de 27.1 Hz. À comparer au 30 Hz donné par le fabricant. L'écart-type de l'écart temporel entre deux frames est de 0.006 secondes. Les données révèlent qu'un frame sur cinq de la Kinect capté par Unity s'obtient avec un délai compris entre 0.045 s et 0.05 s, ce qui explique un frame rate moyen assez différent de 30 Hz.

g. Modifier *TraceData.cs* de manière à ce que la trace ne s'exécute que si une personne est trackée par la Kinect. Pour cela, modifier le script *GetJointPosition.cs* de manière à tracer les données uniquement si la main droite est trackée. Regarder pour cela le script *KinectManager.cs* afin de trouver la bonne fonction.

Nous utilisons pour cela la fonction `manager.IsJointTracked(userId, (int)joint)`.

h. Créer une nouvelle scène *Squelette animé* reprenant *KinectController* et incluant dans l'EmptyObject *body* tous les GameObjects de type Sphère et de rayon 0.1 mètre associés à toutes les parties du squelette identifiées par la Kinect. Utiliser les points d'entrée de la classe *CubeManController*.

Voir la scène nommée *Deuxième scène Squelette Entier*. Remarquer que tous les *GameObjects* associés à une articulation du corps humain reconnue par la Kinect sont donnés en paramètre de la classe *CubeManController* située sur le *GameObject body*.

3. Utilisation du package `KinectV2withMsSDK.unitypackage`

a. Décompresser le package `KinectV2withMsSDK.unitypackage` et charger la démo *KinectAvatarsDemo1*.

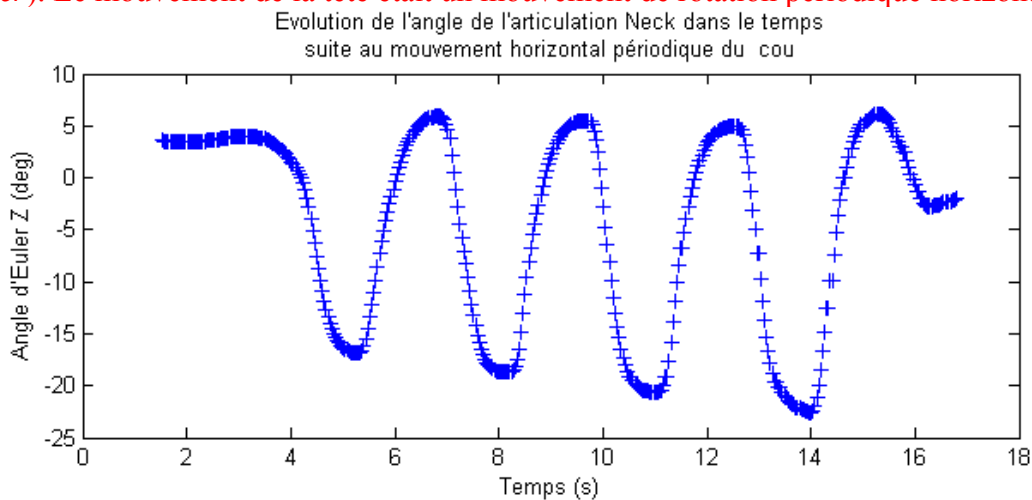
b. Regarder la hiérarchie des GameObjects *Cubeman*, *U_CharacterFront* et *U_CharacterBack* ainsi que les classes associées à ces objets. Comment les différentes parties du squelette peuvent-elles se déplacer entre-elles? Observer en particulier la hiérarchie des différents GameObjects ainsi que les scripts *CubemanController.cs* et *AvatarController.cs* qui permettent de les animer.

La classe *CubeManController* permet de récupérer la position et l'orientation de chacune des articulations sans dépendance hiérarchique des GameObjects associés (voir *Deuxième scène Squelette Entier*). La classe *AvatarController* permet de plaquer un avatar sur un squelette organisé hiérarchiquement du Pelvis jusqu'aux articulations extrêmes (tête, pieds, mains). Dans le premier cas, il est possible d'affecter la position d'une articulation indépendamment des autres alors que dans le deuxième cas, on ne va considérer que la modification de l'angle de l'articulation fille et de l'articulation mère.

c. Ecrire un script *TraceDataRot.cs* qui trace les données de rotation du GameObject *joint_ShoulderRT* hérité de *U_CharacterFront* en vous inspirant du *TraceData.cs* du 2. Faites de même avec les données de rotation du GameObject *joint_Neck*. Qu'en déduisez-vous de la possibilité d'implémenter la technique d'I3D *Direction du regard* avec une Kinect v2?

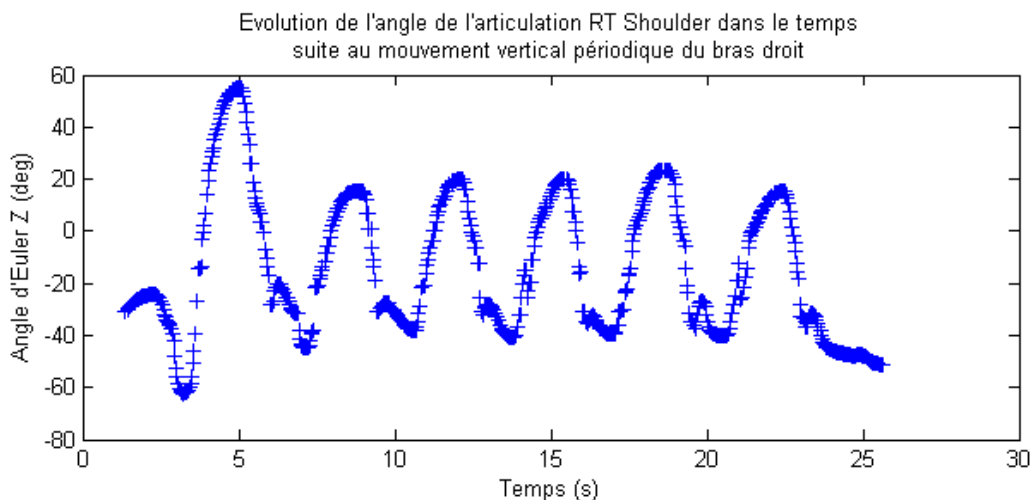
Nous nous inspirons du script *TraceData.cs* pour écrire *TraceDataRot.cs*. Nous traçons ici le *Vector3* *jointOrientation* obtenu en sortie du script *GetJointOrientation.cs*, qui représente les 3 angles d'Euler associés à la partie du squelette *joint_ShoulderRT*, hérité de *U_CharacterFront*.

Ci-dessous le graphique obtenu à partir du fichier *TraceRot.txt* en étudiant l'évolution des axes de rotation du cou (*TraceDataRot.cs* et *GetJointOrientation* placés sur l'empty object *Tracker*). Le mouvement de la tête était un mouvement de rotation périodique horizontal.



Les variations étant conséquentes, on peut espérer obtenir un résultat en utilisant la technique d'I3D *Direction du Regard*, qui a en entrée la direction de la tête de la personne.

Ci-dessous le graphique obtenu à partir du fichier *TraceRot.txt* en étudiant l'évolution des axes de rotation de l'épaule droite (*TraceDataRot.cs* placé sur le GameObject *joint_ShoulderRT*). Le mouvement du bras tendu était un mouvement de périodique, identique à un "battement d'aile".



Les données sont clairement utilisables!

Voir Troisième scène.

d. Ecrire un script *avance.cs* attaché au GameObject *U* permettant de faire avancer *U_CharacterFront* dans la scène.

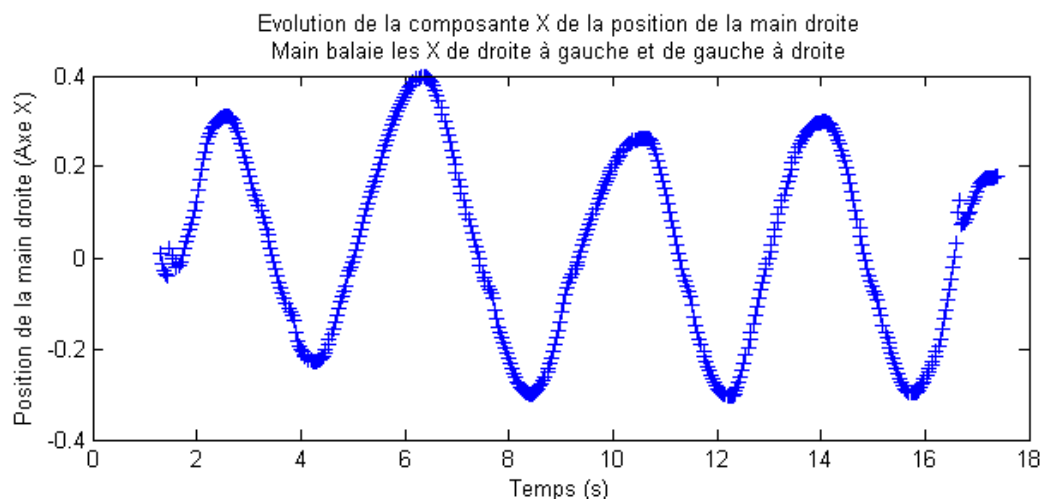
La position absolue du GameObject *U_CharacterFront* varie au cours du temps suivant un déplacement (limité) de la personne devant la Kinect. Pour naviguer dans l'environnement virtuel, on peut utiliser, par exemple, les angles d'Euler associés à *joint_ShoulderRT* et *joint_ShoulderLT* dont on a vu précédemment qu'elles varient "proprement" pour translater *U* donc tout le squelette.

L'idée est de dire que si on lève le bras droit suffisamment haut, le squelette se déplace vers l'avant (ici composante en z) et si on lève le bras gauche suffisamment haut, le squelette se déplace vers l'arrière. Voir *Troisième scène avance*.

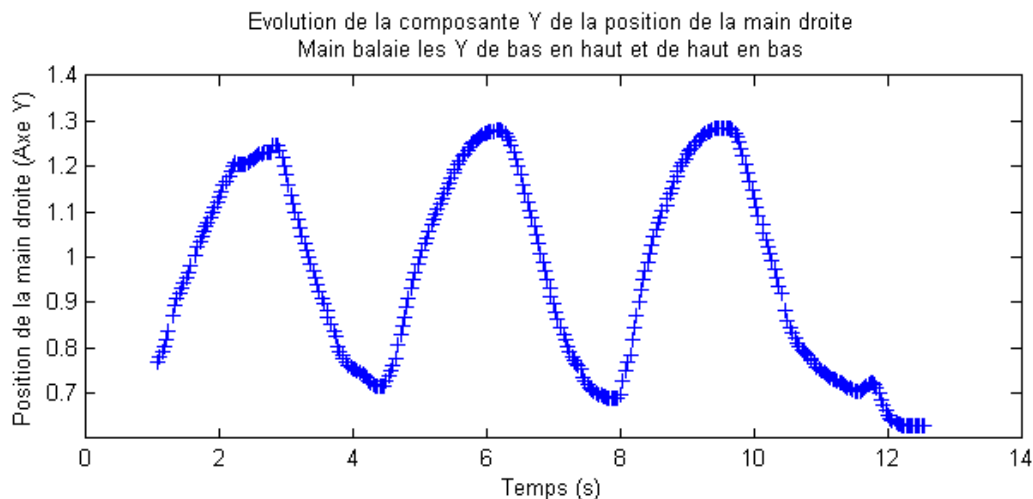
4. Implémentation de la technique de sélection/manipulation Main Virtuelle Simple (MVS).

a. Ecrire un script *mvs.cs* ayant en entrée un *Vector3* issu de la position de la main droite captée par la Kinect, une matrice de passage Réel/Virtuel et en sortie un *Vector3* représentant la position de l'avatar de la main droite *main_droite* dans le monde virtuelle.

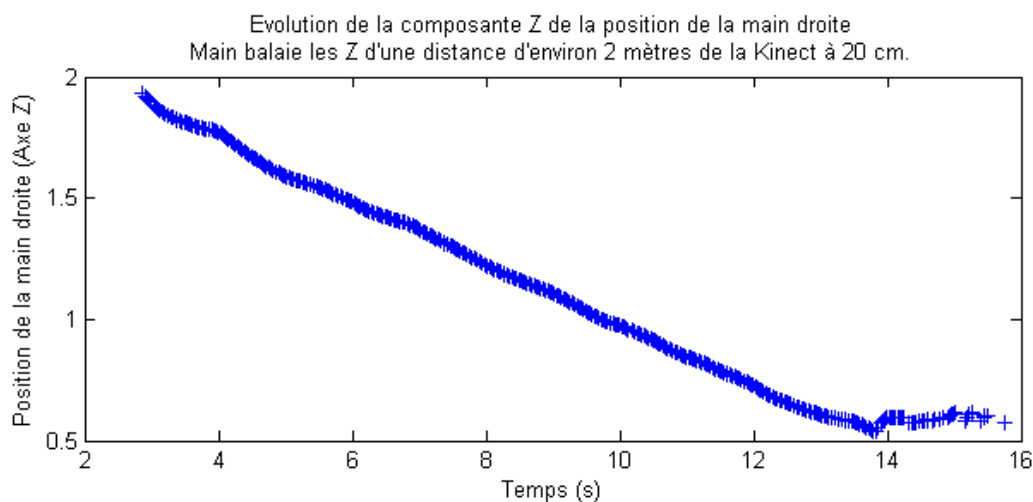
Il nous faut dissocier la position du GameObject qui va représenter la main virtuelle dans l'environnement virtuel (que nous avons nommé *Right Hand Rv* dans le code Unity, voir *Hierarchy*) des données brutes produites par la Kinect. En effet, avant d'utiliser la Kinect, il faut avoir une idée précise de la variations des valeurs suivant les 3 axes liés au repère "Kinect". Voici des graphes obtenus à partir du script *TraceData.cs* (voir 2.) placé sur le GameObject *Tracker*, à partir duquel on récupère les données de la Kinect.



Le premier graphique est obtenu à partir de mouvements de la main droite allant de gauche à droite puis de droite à gauche de la Kinect (axe des X pour la Kinect).



Le second graphique est obtenu à partir de mouvement de la main droite allant de haut en bas puis de bas en haut (Axe des Y de la Kinect).



Enfin, le troisième graphique est obtenu à partir d'un mouvement de la main droite allant d'une distance d'environ 2 mètres de la Kinect à une distance d'environ 20 cm de la Kinect. On se rend compte ici que si on est trop proche de la Kinect, les données sont saturées (voir à partir du temps $t > 14$ secondes). Cela est dû au fait que dans le KinectController, le KinectManager possède un paramètre Min User Distance fixé à 50 cm. Essayez de diminuer cette valeur et observez les données obtenues.

On va utiliser cela pour construire une matrice *Real_to_RV* compatible avec ces observations, qui permettra d'accéder facilement à l'espace du monde virtuel à partir de l'avatar de la main droite de la personne. Ces données correspondent à votre installation (la Kinect for Windows n'a pas les mêmes paramètres optiques que la Kinect pour Xbox, par exemple). C'est donc à vous de bien déterminer les paramètres de cette matrice ainsi que les valeurs adéquates de vecteur d'Offset (translation).

b. Créer un cube dans l'environnement virtuel, que vous nommerez *cible*.

c. Créer un script *EstSelectionnableMVS.cs* ayant comme paramètre d'entrée le `GameObject` *main_droite* et traduisant le fait que *cible* est sélectionnable par *main_droite*. Pour cela, deux possibilités:

- La distance entre *main_droite* et *cible* est inférieure à un réel positif donné
- *main_droite* et *cible* s'intersectent. Pour cela, ajouter la propriété `RigidBody` à *main_droite* et *cible* en otant pour le moment la gravité à chacun des deux `GameObjects`. Cocher *Is Trigger* dans le *Box Collider* associé à *cible*. L'événement d'entrée en collision se gère grâce à la fonction `void OnTriggerEnter (Collider other)`.

Voir le code de EstSelectionnableMVS.cs . Pour obtenir une détection de collision plus précise, on a fixé à Continuous le paramètre Collision Detection des deux GameObjects.

d. Dans le cas où *cible* est sélectionnable, changer sa couleur dans le même script *EstSelectionnableMVS.cs* .

e. Gérer le retour de *cible* à *non sélectionnable* grâce à la fonction `void OnTriggerExit (Collider other)`.

f. Trouver un moyen (i.e. un *Contrôle d'Application*) pour que l'objet *cible* passe de l'état *sélectionnable* à l'état *sélectionné*. Lorsque l'objet *cible* est à l'état *sélectionné*, il devient graphiquement (hiérarchie) fils de *main_droite*. En particulier, la Kinect v2 permet de savoir si une main est "fermée" ou "ouverte". Voir comment grâce à la démo *KinectInteractionDemo1 ...*

*On crée un script *bouge_bras.cs* placé sur un *empty game object* *Contrôle d'Application* qui va retourner le booléen *B1* à vrai ssi la main gauche est en position haute. L'objet *cible* passera de l'état *sélectionnable* à l'état *sélectionné* si *B1* vaut vrai. Dans ce cas, l'objet *cible* devient fils de l'objet *Right Hand Rv*. Lorsque *B1* repassera à faux (bras gauche baissé), l'objet *cible* n'aura plus de père dans la hiérarchie des objets. (voir Quatrième scène).*

5. Implémentation de la technique de sélection Ray Casting .

a. Utiliser la hiérarchie utilisée dans le 4. pour implémenter la technique du RayCasting .Pour cela, écrire un script *raycasting.cs* ayant comme entrée deux angles issus de l'orientation du bras droit, une matrice de passage Réel/Virtuel et en sortie un `Vector3` représentant l'orientation de l'avatar de la main droite dans le monde virtuel.

*On utilise ici une boîte nommée Ray, fille de Virtual Body, de longueur 10 et de coté 0.2, associée à un RigidBody. Les angles d'Euler associés à ce "rayon" sont ceux de *shoulder_right*.*

b. Adapter le cheminement du 4. pour achever l'implémentation de la technique du RayCasting.

Voir Cinquième scène et Cinquième scène pos. Cette dernière utilise la direction entre la main droite et l'épaule droite pour donner la direction du rayon.

6. Implémentation de la technique de navigation Direction de la main.

Utiliser le 5. pour écrire un script *directionmain.cs* permettant de naviguer un vitesse constante v dans le monde virtuelle dans la direction de la main droite (ou gauche) si celle-ci est trackée et ne pas naviguer si celle-ci n'est pas trackée.

On utilise Cinquième scène pos pour déterminer la direction de la main droite qui est couplée avec la direction de la caméra virtuelle. On avance à vitesse constante dans cette direction si la main gauche est en "position haute"

Voir *Sixième scène*.

FIN