UNIVERSITE PARIS-SUD

Master Informatique 1ere Année

Année 2015-2016

Rapport de TER Humains Virtuels

par

Yang CHEN

Dribble de ballon

Enseignant: *Christian Jacquemin, Université Paris-Sud*

Table des matière

1 Introduction

1.1 – Description du domaine de travail

Réalisation en monôme d'un projet supervisé dans le cadre de l'UE Humains Virtuels du semestre 2 de la première année de Master informatique à l'Université de Paris-Sud.

C'est un projet dans le domaine de Réalité Virtuelle en utilisant les techniques d'infographie 3D.

La réalité virtuelle est une technique qui utilise la simulation informatique pour produire un monde virtuel en trois dimensions.
Il y a deux grandes parties dans ce cours :
- Animation fluide par points-clés
- Animation par squelette

Ce projet est réalisé par l'animation par squelette, l'animation par squelette est couramment utilisé par le jeu vidéo et dans l'industrie du film, elle peut également être appliquée à des objets mécaniques.
L'objectif de ce projet est la réalisation et la conception de dribble de basketball, il permet à un personnage de pouvoir dribbler le ballon de basketball dans un environnement en trois dimensions.
Dans le basket, le dribble est la méthode pour avancer le ballon par soi-même. Il consiste à faire rebondir le ballon sur le sol en continu avec une seule main. Un dribble doit être fait avec le bout des doigts, le poignet devrait pousser le ballon, et l'avant-bras doit se déplacer du haut vers le bas.

1.2 - État de l'art

J'ai examiné de nombreux documents relatifs à mon sujet, dans la littérature scientifique il y en a beaucoup qui proposent de très bonnes solutions.

Une méthode efficace est d'utiliser le moteur physique Open Dynamics Engine (ODE) [1] afin d'avoir des réponses physiques réalistes lors de l'interaction des modèles d'avatar avec l'environnement. Le modèle d'avatar sera représenté dans ODE comme un ensemble de corps rigides, reliées entre eux par un certain nombre d'articulations qui limitent les positions et orientations [3]. Il utilise également un cadre libre et modulaire "Nebula Device" pour construire des visualisations 3D [4], Nebula fournit un système d'animation de personnage et contient un système de détection de collision qui traite de la géométrie statique à l'aide de la bibliothèque "Optimised Collision Detection" (OPCODE) [2].

Le système de détection de collision Nebula utilise le terme "collide shape" pour se référer aux données qui décrivent la forme d'un objet pouvant être impliqué dans des collisions. Le "collide shape" est une forme géométrie qui peut entrer en collision avec différent objets comme le monde réel. Les formes de collision les plus fondamentaux sont des sphères et cubes. La forme de collision n'a pas de position dans le repère du monde, ils sont attachés à des objets de collision ou des corps rigides.
Il fournit une méthode pour améliorer la performance de détection de collision, on utilise une sphère englobant comme un "collide shape", on peut encapsuler cette sphère englobant dans l'objet et la sphère englobant est utilisée pour la détection de collision rugueuse [3].
Le système de détection de collision Nebula associe une sphère englobante avec tous les objets qui peut être impliqué dans une collision à un moment ou à un autre. Il fournit deux méthodes pour vérifier la collision entre une paire d'objets en mouvement, la première méthode est "the quick swept sphere approach" [5], c'est une méthode rapide. Une autre méthode relativement plus lente mais plus précise qui place une limite supérieure sur le nombre maximum de tests d'intersection qui sera fait, il n'effectue que des plusieurs tests sur le déplacement du vecteur si l'objet a dépassé plus de 1/8 de rayon de sa sphère englobant.

1.3 – Lien entre état de l'art et mon sujet

Par l'étude des littératures scientifiques, je me suis inspiré de diverses bonnes idées. Il y a deux phases de détection de collision sur mon sujet. La première phase est la collision entre le ballon et le sol, la détection de collision dans cette phase est simple car le sol est un objet stationnaire. Dans le processus de chute d'un ballon, on détectera la coordonnée y d'un ballon, si la coordonnée y est inférieure ou égale à son rayon, le ballon et le sol se croisent, il faut faire rebondir le ballon en inversant la vitesse et mettant à jour la nouvelle coordonné y avec la coordonnée y précédent.

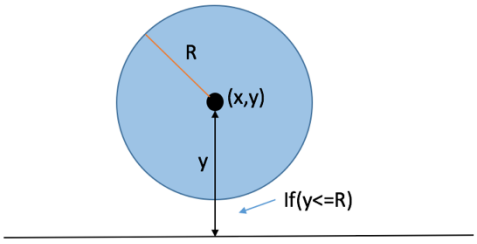


FIGURE 1. PHASE 1 : La collision entre le ballon et le sol

La deuxième phase de détection de collision est la collision entre le ballon et la main, c'est un peu plus compliqué car la main et le ballon sont des objets en mouvement. Je me suis inspiré de la littérature, pour simplifier le problème, je vais encapsuler la main dans une sphère qui l'englobe : ce n'est pas une vraie sphère mais un objet imaginaire, donc la collision se passe entre deux sphères (une des sphères représentant le ballon). Le modèle mathématique de sphère est plus simple que le modèle mathématique de polygone. Dans mon sujet, on considère que le ballon et la main se déplacent dans une même ligne, on va calculer la distance minimale entre les centres des deux ballons, si la distance entre deux ballons est inférieure ou égale à la somme du rayon du ballon et du rayon de la sphère imaginaire de la main, alors les deux sphères se croisent. Pour bouger la main, on calcule l'interpolation avec keyframing, si on détecte la collision, on considère que la main ne change pas de vitesse. Pour le ballon, on va donner la vitesse de la main au ballon dans ce moment, la nouvelle vitesse est aussi dans la même ligne auparavant. Je n'utilise pas le

moteur physique car il n'y a pas beaucoup d'objets dans l'environnement, il n'y en a que deux.

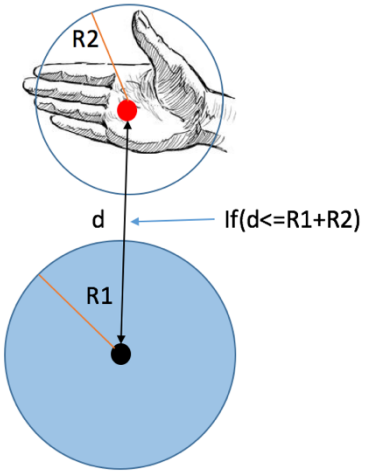


FIGURE 2. PHASE 2 : La collision entre le ballon et la main

2. Architecture

2.1 – Structure de Données

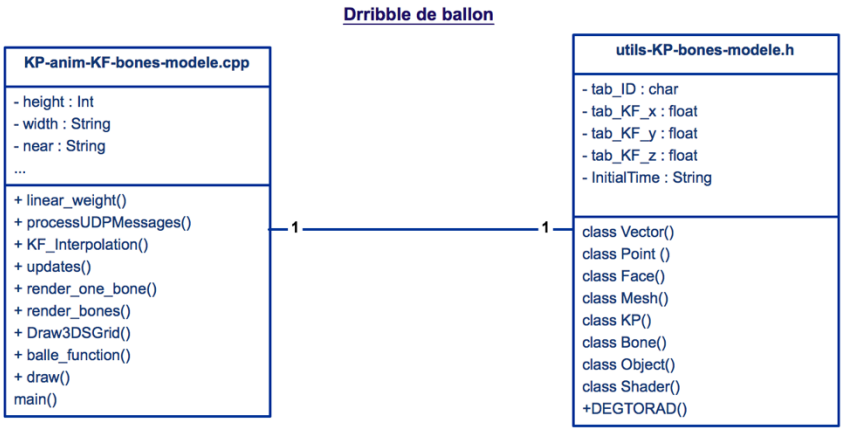


Figure3 : UML 1

KP-anim-KF-bones-modele.cpp

-Draw3DSGrid() : dessiner le sol
-balle-function() : animation de ballon
-KF_Interpolation() : interpolation entre les keyframe

utils-KP-bones-modele.h

Déclaration des fichiers de header et différente classes.
La fonction **animate_one_point**() dans la classe Object est utilisé pour calculer la transformation de chaque vertex.

2.2 – Les bones choisis et leurs positions

J'ai choisi des bones comme ci-dessous, on peut trouver la hiérarchie des bones : les os fils, les os père, la racine et l'effecteur. On peut exporter du squelette à l'aide du script python, il permet de hiérarchiser les os du modèle.

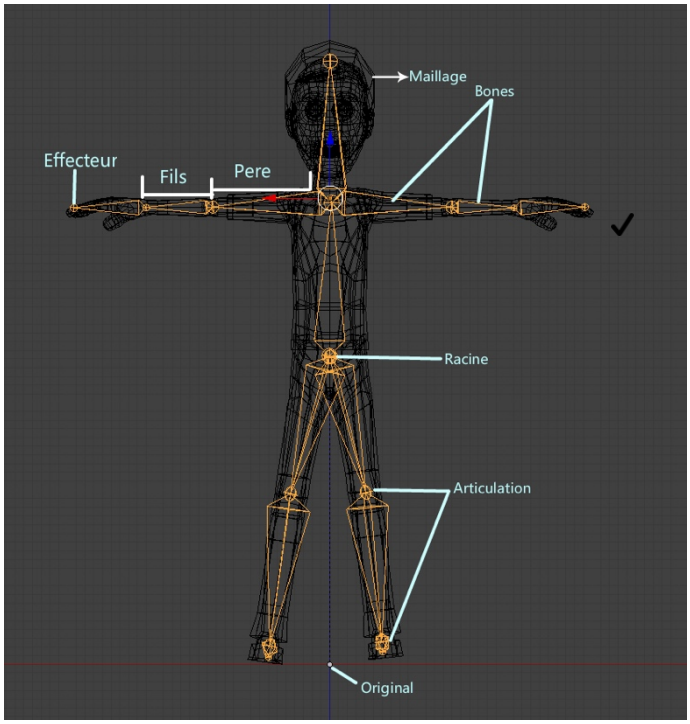
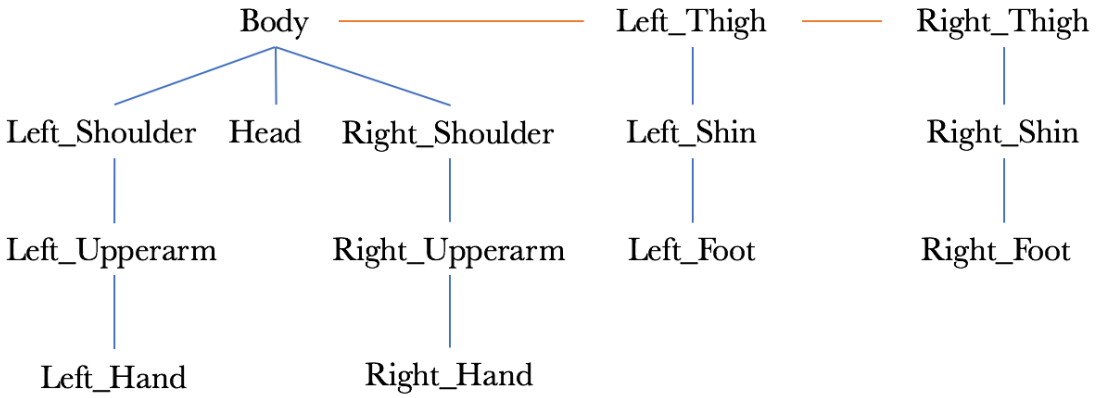


Figure4. Squelette du modèle

3. Mise en oeuvre du projet

3.1 - La hiérarchie de bones

Pour réaliser un avatar réaliste, j'ai mis quatorze os sur mon modèle. Le squelette est composé d'un ensemble d'os que l'on organise hiérarchiquement, de façon à former un arbre.



Chaque os du squelette a deux articulations et chaque articulation est reliée au moins un os. Lorsqu'on va faire une animation, seulement pré-calcule les poses-clés de son squelette avec Blender, puis créer un algorithme qui calcule l'interpolation entre les "keyframes". Enfin, il suffit d'appliquer l'interpolation des vertex basant ses calculs sur les poids de vertex et de la hiérarchie des os.

Lorsqu'on applique un "skin", on doit également définir des poids vertex pour chaque os. Cela se fait dans le Blender, on peut l'exporter le maillage avec les sommets pondérés, les os et les textures.

3.2 - Animation du ballon (rebonds)

Pour réaliser l'animation du ballon, je crée un modèle de ballon avec Blender, j'exporte un fichier Ball.obj. Ensuite j'importe ce fichier dans le programme, je définis la position de ballon pour poser le ballon devant l'avatar. Pour simuler le chute d'une balle, j'introduis les lois de la physique qui permet de simuler le mouvement d'un objet sous l'effet de la gravité, j'ajoute également la force friction comme le monde réel. Il y a beaucoup de paramètre ajoutées : la position actuelle, la position précédente, la vitesse actuelle, la vitesse précédente, l'accélération actuelle, l'accélération précédente et aussi la longueur du pas de temps effectué.

J'utilise l'intégration numérique pour calculer la vitesse et position à l'instant suivant à partir de l'instant courant en effectuant une intégration du temps, par exemple le calcul de coordonnée x :

$$ball_{x_{actuel}} = ball_{x_{précédent}} + \left(ball_{V\_X_{précédent}} * pas\ de\ temps\right)$$

Le calcul de vitesse da l'axe x :

$$ball_{v\_x_{actuel}} = ball_{v\_x_{précédent}} + \left(ball_{accélération\_X_{précédent}} * pas\ de\ temps\right)$$

Pour calculer l'accélération, j'introduis le coefficient de l'accélération normale de la pesanteur terrestre et le coefficient d'amortissement, la formule :

$$ball_{accélération\_x_{actuel}} = 0 - \left(ball_{V\_X_{actuel}} * Coef_{\text{amortissement}}\right)$$

Dans mon programme, l'avatar debout sur la direction de l'axe Z, donc le ballon se déplace sur l'axe Z :

$$ball_{accélération\_z_{actuel}} = -G - \left(ball_{V\_Z_{actuel}} * \ Coef_{\text{amortissement}}\right)$$

Ensuite je mets à jour les variables précédentes par les variables actuelles pour calculer des nouvelles variables. Pour animer le ballon, j'ajoute un tableau BallTranslation[] dans le fichier "utils-KP-bones-modele.h" pour stocker les translations sur l'axe x, y et z, après j'ajoute les coordonnées x, y et z dans ce tableau, avec la fonction translate de la

bibliothèque GLM, je peux animer le ballon. Le ballon est une sphère, on n'a pas besoin de rotation ici.

3.3 - Animation du bras et de la main

Quaternion

D'abord, je définis l'angle et l'axe de rotation des x, y et z dans quatre tableaux, chaque tableau contenant 6 keyframes. J'utilise le quaternion de la bibliothèque GLM pour représenter l'angle et l'axe de rotation des x, y, z d'une posture par un quaternion, un autre quaternion pour représenter la posture suivante. Si on calcule l'angle, il faut convertir les degrés en radians. On parcourt tous les os de l'avatar et charge l'angle et les coordonnées x, y, z de chaque os par une boucle.

Posture

Pour simuler l'animation du dribble. Je définis cinq postures de faire l'animation du bras et la main, je remplis l'angle et l'axe de rotation des x, y et z de chaque os dans les quatre tableaux, chaque posture est un Keyframing, les cinq postures principales comme ci-dessous :

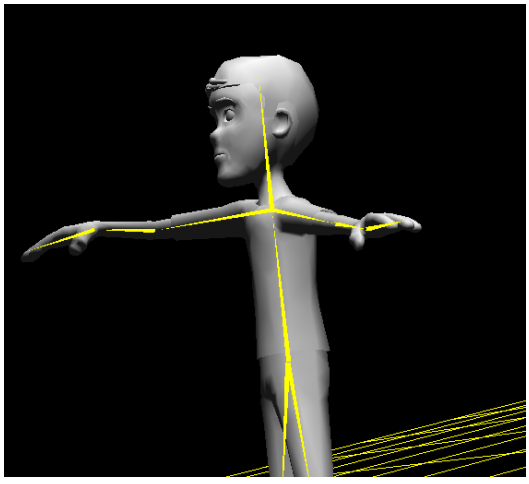


Figure5 : Posture de dribble 1

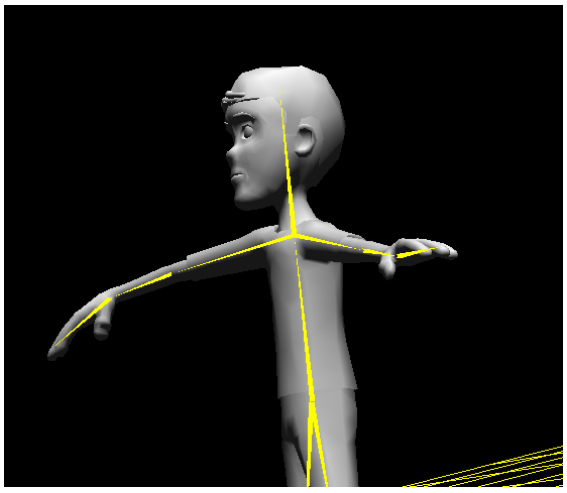


Figure6 : Posture de dribble 2

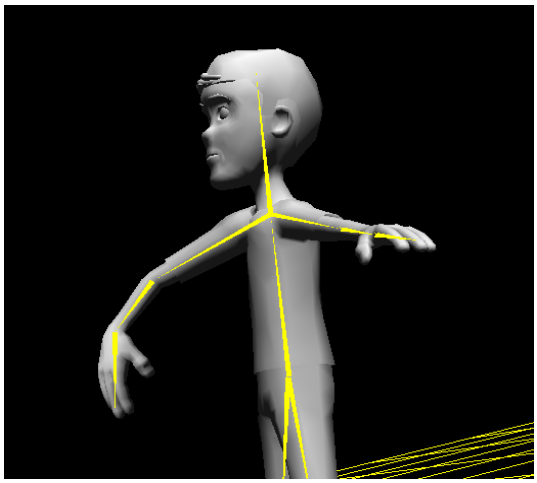


Figure 7 : Posture de dribble 3

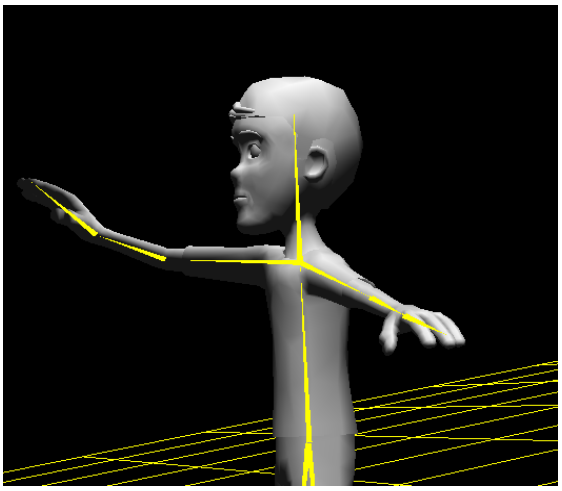


Figure 8 : Posture de dribble 4

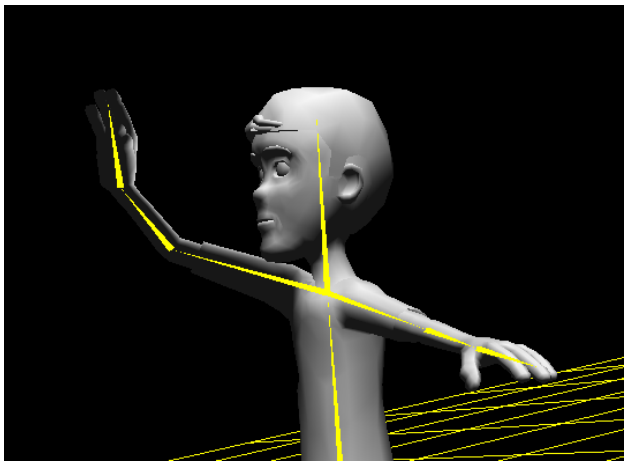


Figure 9 : Posture de dribble 5

Ce sont les postures principales, j'ajoute aussi quelques postures pour simuler comme un homme réel.

Interpolation

Ensuite je vais faire l'interpolation entre les keyframing, il consiste à évaluer une posture intermédiaire cohérente entre deux postures. J'utilise la fonction "slerp" (*Spherical Linear Interpolation*) de la bibliothèque GLM, on définit une variable de temps **alpha**, je prends les

deux quaternions et la variable alpha comme paramètres de slerp, il va interpoler les cinq postures au-dessous. La fonction d'interpolation slerp va produire un nouveau quaternion qui contient l'angle de rotation et l'axe de rotation de ce quaternion. J'utilise la fonction "rotate" de la bibliothèque GLM et ce quaternion pour mettre à jour la matrice de rotation d'os ***boneAnimationRotationMatix.*** À la fin, le bras et la main doivent se déplacer du haut vers le bas après du bas vers le haut sans arrêt.

3.4 – Synchronisation de la main et le ballon

Pour réaliser l'animation de dribble, il faut synchroniser l'animation de la main et du ballon. La synchronisation contient quatre phases, on fait l'interpolation entre les quatre phases, dans la phase1, la main se croise avec le ballon. Dans la phase2, la main se déplace vers du bas, le ballon arrive à la position 2(*Figure 10*). Dans la phase3, le ballon arrive à la position 3, sur le sol, la main arrive à la position 3, c'est la même position de position1. Dans la phase4, le ballon qui rebondit sur le sol, il arrive à position 4, la main se déplace vers le haut, après il va répéter les quatre phases dans une boucle.

Je définis un paramètre de temps, une vitesse initiale du ballon et une position initiale du ballon pour synchroniser les deux. Je mets la fonction d'animation du ballon et la fonction d'interpolation dans une même boucle while, je trouve que la boucle while s'exécute 30 fois pour parcourir tous les keyframes de la main, donc le ballon devrait arriver à la position 1 (*Figure 10*) pour la trentième fois de la boucle. Je récupère les coordonnées et la vitesse de la postion1 de la main comme la position et la vitesse initiale du ballon, après j'utilise la formule de calcule la position du ballon(*Chapitre3.2*) et ces paramètres pour calculer le paramètre de temps (*la longueur du pas de temps effectué*). Quand on détecte la collision entre la main et le ballon, j'utilise la vitesse de la main comme la nouvelle vitesse du ballon. La main et le ballon se croise toujours à la position1, c'est la synchronisation.

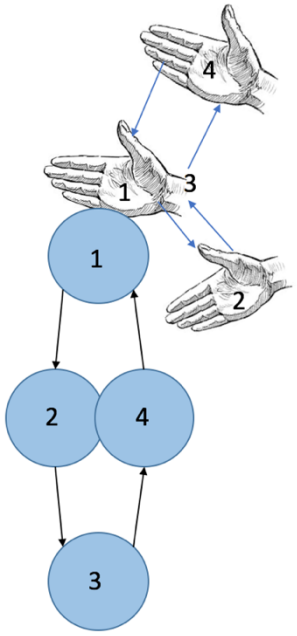


Figure10: Synchronisation

3.5 - Gestion des collisions pendant l'animation et algorithme de combinaison des collisions avec l'animation.

Comme j'ai énoncé la méthode de détection de collision, maintenant je vais détailler le concept.

Dans la phase de détection de collision entre le ballon et le sol, je dessine le sol, le sol est représenté par une grille 50x50, les pieds de l'avatar sont sur le sol, l'axe z dans le repère du modèle est opposé de l'axe y, la coordonnée z de sol est 0, donc si la coordonnée z du centre de ballon est inférieure ou égale au rayon du ballon, je vais inverser la vitesse du ballon de l'axe z et mettre à jour la nouvelle coordonné z avec la coordonnée z précédent pour rebondir le ballon.

Dans la phase de détection de collision entre le ballon et la main, il est important de trouver le centre de la main, qui est difficilement trouvable, car pour trouver le centre du ballon, je peux simplement utiliser la coordonnée (x, y, z) du ballon, mais pour trouver la position de la main, on ne connaît que la coordonnée initiale d'un vertex du centre de la main droite par le logiciel Blender(*Figure11*). Quand on fait l'interpolation de la main, la coordonnée changera tout le temps, donc il faut récupérer la coordonnée de tout le temps.

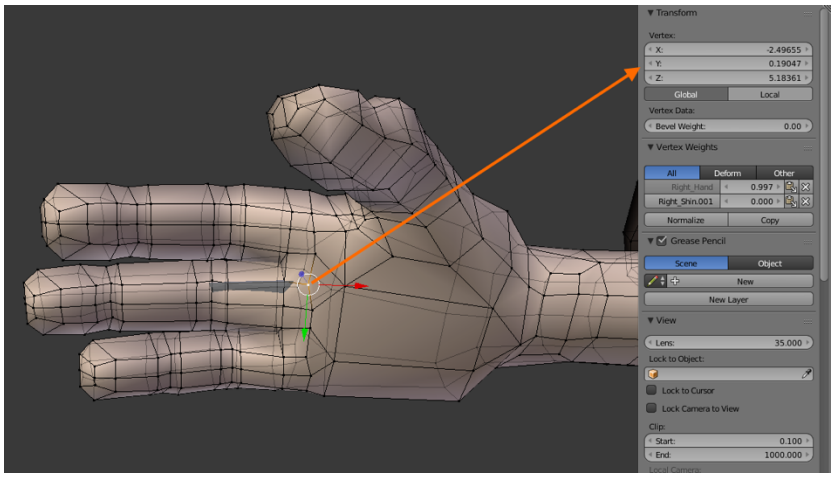


Figure11 : Position du centre de la main droite

Je trouve un moyen pour trouver la coordonnée de chaque moment. Dans le programme, pour chaque maillage, la transformation du maillage est calculée dans le repère du maillage. Donc il faut calculer la matrice relative avec la transformation courant d'articulation et l'inverse de transformation initial local d'articulation :

$$matRelative = initJointTransformation \times currentJointTransformation$$

On met la matrice "matRelative" dans un tableau de mat4 ***pointAnimationMatrix*** pour calculer la transformation des positions de chaque vertex du maillage. Il utilise la somme pondérée de toutes les os du vertex et pointAnimationMatrix pour calculer la nouvelle coordonnée de chaque vertex quand on fait l'interpolation. On connaît la position initiale de la main droite dans le repère du monde, c'est la coordonnée (*-2.496552, 0.190474, 5.183606*), donc dans la fonction de calcule l'animation de chaque point (*dans le fichier utils-KP-bones-modele.h*), je récupère toutes les coordonnées de ce point de chaque moment quand on fait l'interpolation. Dans la fonction de détection de collision, j'utilise ces coordonnées et la position du ballon pour calculer la distance euclidienne entre le centre de la main et le ballon de chaque moment, à la fin je définis la distance minimale pour détecter la collision entre le ballon et la main comme la méthode que j'ai énoncée.

Pour calculer la vitesse de chaque moment de la main, je calcule la distance euclidienne et le temps entre deux coordonnées du point centre de la main, je peux les utiliser pour calculer la vitesse de chaque moment.

4. Synthèse et conclusion

L'objectif de ce projet est de réaliser un humain virtuel qui peut dribbler le ballon comme un humain réel, mais c'est encore plus loin, car le mouvement d'avatar est généré par le programme, si on souhaite réaliser un mouvement comme réel, il faut utiliser la technique de l'animation par capture de mouvements. La capture de mouvement peut réduire le temps de développement de l'animation par squelette et augmenter le niveau de réalisme. À présent, ces moments de travail ont beaucoup modifiés mon aptitude et renforcées notre détermination à trouver des solutions. J'ai dû améliorer notre méthode et apprendre à travailler dans l'urgence, car bien que nous ayons prévu beaucoup de choses avant de commencer, je n'avais alors pas assez d'expérience pour voir concrètement la fin du projet.

Annexe

Commandes pour l'utilisation

Gestion du clavier :
- "+" : déplacer vers le bas
- "-" : déplacer vers le haut
- "<" : dézoomer
- ">" : zoomer
- "v" : déplacer le ballon vers l'axe z positif
- "V" : déplacer le ballon vers l'axe z négatif
- "b" : déplacer le ballon vers l'axe x positif
- "B" : déplacer le ballon vers l'axe x négatif
- "n" : déplacer le ballon vers l'axe y positif
- "N" : déplacer le ballon vers l'axe y négatif

Compilation:
make KP-anim-KF-bones-modele
Lancer l'application PERL :
unset PERL5LIB + ./Bones-control_gtk.prl
Lancer le programme :
./KP-anim-KF-bones-modele

BIBLIOGRAPHIE

[1] "Open Dynamics Engine (ODE) Community Wiki", http://ode-wiki.org/wiki, visité en 04/2016.

[2] Terdima, P., "OPCODE home page "P. http://www.codercorner.com/Opcode.htm, visité en 04/2016.

[3] Macagon, V. and Wünsche, B., "Efficient Collision Detection for Skeletally Animated Models in Interactive Environments" ,(2003).

[4] The Nebula Device Wiki., "home page", http://nebuladevice.sourceforge.net, visité en 04/2016.

[5] Gomez, M., "Simple intersection tests for games", Gamasutra.com, October 18, (1999).