Contrôle d’avatar par capteur Kinect

Prenez le contrôle !

BARBESANGE Benjamin & GARÇON Benoît

juin 2015

Remerciements

Nous tenons à remercier Monsieur Emmanuel Mesnard qui nous a permis de réaliser un projet qui nous tenait à cœur mais aussi pour l'aide qu'il nous a fourni tout au long de ce projet.

Nous voudrions aussi remercier notre ami Simon Leschiera pour nous avoir prêter du matériel et nous avoir permis d'avancer plus rapidement dans notre projet.

Table des matières

Présentation du projet

Origine du projet

Dans le cadre de notre projet de première année à l'ISIMA, nous avons choisi de développer un programme de contrôle d'avatar 3D par une Kinect. Ce projet s'inscrit dans la continuité du cours de réalité virtuelle de deuxième année de Prep'ISIMA.

L'objectif sera donc de réussir à animer à l'écran un personnage (humanoïde) grâce aux mouvements de l'utilisateur numérisés par le capteur Kinect. Il s'agira donc dans un premier temps de développer un programme sous Processing capable de mettre en œuvre un tel mécanisme.

Etude préliminaire du projet

Ce projet est beaucoup plus conséquent qu'il ne pouvait le laisser paraître. En effet, il faut tout d'abord s'approprier le matériel : des capteurs propriétaires avec des compatibilités parfois limitées avec certaines technologies. Il convient donc d'étudier toutes les possibilités offertes par le matériel.

Ensuite afin d'avoir une compatibilité reconnue avec les technologies informatiques nous avons décidé d'utiliser la bibliothèque Processing, en effet cette bibliothèque nous est familière et c'est un bon compromis entre haut niveau de programmation et programmation précise. Il existe aussi Processing sous forme d'environnement de développement toutefois pour des raisons de confort de développement il a été décidé d'utiliser Eclipse afin d'avoir accès aux différents outils qu'il propose. Afin d'utiliser Processing correctement dans Eclipse il nous a été nécessaire d'installer le plug-in Proclipsing, cette démarche est expliquée plus en détail en annexe.

Une fois tout ceci en place il nous a fallu ajouter les bibliothèques annexes pour l'utilisation du matériel à savoir **SimpleOpenNi** et **KinectPV2**. Les problèmes ont été nombreux avant de pouvoir compiler un exécutable. Mais après avoir obtenu une version patchée des bibliothèques, la compilation et l'exécution du programme étaient possible.

Alors il nous a fallu rechercher le meilleur moyen de représenter des personnages virtuels et ensuite de les animer, grâce à une animation squelettale. La modélisation des données a été dans ce projet la partie majeure et a demandé beaucoup de recherches.

Enfin, quand notre objectif initial a été atteint, nous nous sommes appliqués à mettre en place des optimisations et améliorations de notre premier algorithme d'animation naïf.

Présentation du programme

Le programme principal contenu dans le fichier **ZZavatar.pde/ZZavatar.java** est un programme mono-utilisateur repérant l'utilisateur à l'aide d'un capteur Kinect (version 1 ou 2 indifféremment). Les mouvements de cet utilisateur sont retranscrits à l'écran au travers d'une modélisation 3D d'un personnage humanoïde (ou non) grâce à notre algorithme de contrôle.

Un contrôle plus classique du programme est possible :

- Les touches directionnelles, '+' et '-' permettent de contrôler une caméra orientée sur le personnage ;
- La touche 's', ou suivant, permet de passer au modèle suivant ;
- La touche 'd' permet d'activer un mode debug ;
- La touche 'f' permet de changer le fond de la scène ;
- La touche 'g' permet d'activer/désactiver le fond.

La gestion des Kinects se fait avec les classes **ZZkinectV1** et **ZZkinectV2** qui sont toutes les deux soumises à l'interface **ZZkinect** ce qui nous a permis d'utiliser indifféremment l'une ou l'autre dans le code principal.

La gestion du modèle 3D est faite par la classe **ZZModel**, celle du squelette par **ZZkeleton** et les textures par **ZZMaterial**. Ces trois classes sont une partie majeure du traitement des données et du projet, tout le projet repose sur cette modélisation.

La classe **ZZbackground** gère quant à elle l'utilisation du fond d'écran.

Les classes **ZZector**, **ZZertex** et **ZZoint** sont des versions plus ou moins spécialisées de la classe PVector de Processing. Chacune de ces classes possèdent des méthodes et attributs qui facilitent une partie du traitement :

- **ZZector** est une classe mère de **ZZoint** et **ZZertex** ;
- **ZZoint** est plutôt utilisé pour la gestion des joints du squelette des différents personnages/utilisateurs ;
- **ZZertex** (venant de vertex) est une spécialisation utile pour la gestion des sommets de maillages (modèles 3D).

Nous avons aussi une classe **ZZfifo** qui est une simple structure de file utile pour la classe **ZZoptimiseur** qui est chargé d'une partie non négligeable de l'amélioration de notre algorithme d'animation : le lissage.

De plus ne programme comporte de nombreux fichiers de données dans le dossier data : les maillages de personnages au format wavefront (.obj), les squelettes au format bvh, les textures (images et .mtl). Nous avons aussi des fichiers .bdd qui sont des listes servant de base de données rudimentaires (liste des modèles ou fond d'écrans à charger).

Nous tenons à signaler qu'il existe deux versions du code source : une première compatible avec les environnements Java classiques et une seconde compatible avec l'environnement Processing. Processing ne permet pas de faire du « pur » Java, c'est pourquoi la version Processing du code est moins conventionnelle.

Etude du matériel

Processing

Processing est un environnement de développement gratuit et libre de droit permettant de créer des applications. Ces applications peuvent utiliser diverses bibliothèques qui sont disponibles pour tous.

Processing est initialement prévu pour pouvoir être utilisé par des non-initiés de la programmation. Sa cible était principalement composée d'artistes du numérique et de jeunes enfants. Toutefois de nombreux développeurs confirmés ont commencé à s'y intéresser et de nouvelles applications lui ont été trouvées.

Grâce à Processing, il est facile d'interagir avec des composants (caméras, Kinect) afin d'effectuer du traitement d'images par exemple, ce qui permet de créer des applications de réalité virtuelle.

Dans le cadre de notre projet, nous utilisons la version 2.2.1 de Processing. De plus, nous avons inclues les bibliothèques suivantes :

- **Core** : bibliothèque indispensable contenant les méthodes de base
- **SimpleOpenNI** : pour le fonctionnement de la Kinect 1
- **KinectPV2** : pour le fonctionnement de la Kinect 2

Capteur Microsoft Kinect 2

Ce capteur nous permettra l'acquisition des mouvements de l'utilisateur. En effet, grâce à ce dernier nous avons accès à pas moins de 25 points représentants des zones clefs du squelette à 60fps dans une résolution 1920x1080.

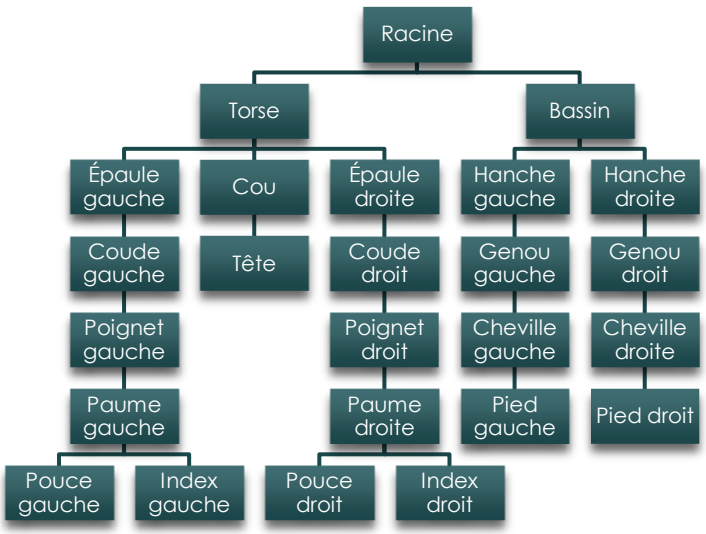

Figure 1 - Joints du squelette Kinect 2

La gestion de la Kinect 2 s'effectue grâce à **ZZKinectV2**.

Nous disposons des informations d'affichage de la Kinect (taille et l'image, image couleur, image en profondeur) ainsi que d'un tableau permettant de stocker les squelettes des utilisateurs.

Capteur Microsoft Kinect 1

Dans le cas de la Kinect 1, nous disposons de moins de points représentant le squelette ; nous en avons 15 :

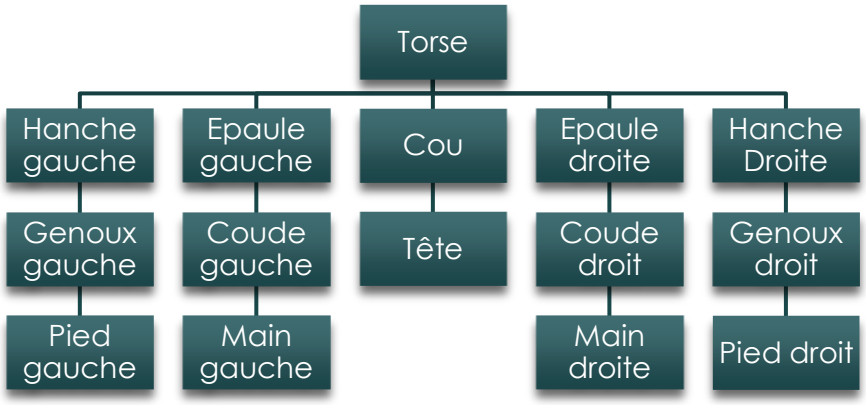

Figure 2 - Joints du squelette Kinect 1

La Kinect 1 permet d'ouvrir un flux vidéo ayant pour résolution 640x480 avec un taux de rafraichissement maximal de 30fps.

Pour pouvoir interagir avec, nous avons besoin de la bibliothèque **SimpleOpenNI**. Il existe d'autres bibliothèques permettant de faire l'interface avec la Kinect, cependant nous avons choisi cette bibliothèque car nous y sommes plus accoutumés.

La gestion de la Kinect 1 s'effectue avec **ZZKinectV1**. Nous avons les mêmes informations d'affichage que pour la Kinect 2 (seules les valeurs des paramètres changent).

Utilisation pratique des Kinects

Afin de simplifier l'utilisation de la Kinect (indépendamment de la version), nous avons décidé de créer une interface **ZZKinect** dont **ZZKinectV1** et **ZZKinectV2** héritent.

Ceci permet donc de déclarer les méthodes pouvant être utilisées sur les Kinects dans la classe mère, pour ensuite définir le code dans les classes filles. Ainsi, il n'y a qu'une seule méthode à utiliser, ce qui permet de simplifier l'utilisation à l'utilisateur.

Les méthodes importantes de la classe mère sont les suivantes :

- **getVersion()** : retourne la version de la Kinect en cours d'utilisation
- **available()** : indique si la Kinect est disponible ou non
- **refresh()** : rafraichit les données utilisées par la Kinect
- **getSkeleton(int numUser)** : recupère le squelette de l'utilisateur spécifié dans un tableau de **ZZoint**
- **isTrackingSkeleton(int skelNum)** : indique si la Kinect capte le squelette de l'utilisateur spécifié
- **getRGBImage()** : récupère l'image couleur de la Kinect pour permettre de l'afficher si besoin

Travail sur les données

Travail préliminaire

Identification des données 3D à traiter

Avant d'animer quelque chose il faut d'abord se demander ce que l'on va animer. En effet, pour animer un personnage il faut avant tout avoir un personnage. Nous nous sommes renseignés sur les différentes possibilités pour représenter un personnage à animer. Il s'est avéré que dans tous les cas deux composantes étaient indispensables : une première représentant le modèle et une deuxième servant à l'animation. Nous avons trouvé dans la littérature beaucoup d'exemples de structures où l'application de modifications à la composante animatrice déformait la composante représentative. C'est ainsi que nous nous sommes orientés vers l'animation squelettale.

Ce modèle d'animation nous a semblé le plus naturel étant donné qu'il est basé sur le principe de la cinétique humaine : tout le corps (dans notre cas le modèle) se déforme en suivant le squelette soumis au contraintes musculaires (dans notre cas les données de mouvements reçu par la Kinect).

Formats et contraintes existantes

Processing est une bibliothèque principalement orientée vers les artistes 2D : nous l'avons appris à notre grand désarroi. Bien que de plus en plus de bibliothèques de réalité virtuelle y voient le jour, l'utilisation de la 3D est très primitive. En effet il n'existe dans la version 2 qu'une seule façon de gérer un objet 3D. Ce système consiste à charger un objet wavefront (obj) dans une PShape : de prime abord cela peut paraitre convenable mais cela ne l'est pas puisque Processing charge les fichiers obj comme un simple maillage de point alors que ce format est beaucoup plus puissant que cela. Normalement un fichier wavefront gère des sous-objets et des sous-groupes. Ce système de chargement ne pouvait donc pas être retenu car il était impossible de discerner quel sommet du maillage correspondait à quelle partie du modèle sans les groupes de vertices.

C'est pourquoi nous nous sommes lancés à la recherche d'un format dédié à l'animation squelettale. Nous avons vu de nombreux formats dont beaucoup de formats qui étaient soit propriétaires, soit binaires ou les deux. Cela faisait d'eux des fichiers difficilement manipulable et trouvable.

Nous avions manipulé durant notre module de réalité virtuelle des fichiers MD2 qui permettaient d'animer des personnages à l'écran et ce format était pris en charge par une bibliothèque Processing. En étudiant ce format nous nous sommes vite rendu compte qu'il ne conviendrait pas. En effet, le seul type d'animation qu'il permettait était l'animation par frame ce qui est incompatible avec notre projet.

Nous avons continué à chercher dans la direction du format MD car il existe aussi deux autres versions plus récentes : le MD3 et le MD5. Le problème avec le MD3 était toujours le même : il ne gère que l'animation par frame. Mais avec le MD5 nous touchions au but : il gère l'animation squelettale, c'est un format de fichier lisible (et compréhensible) par l'homme et il existe de nombreux modèles téléchargeable sur internet. Nous nous sommes donc empressés de rechercher une bibliothèque permettant d'utiliser ces fichiers. Sur Processing nous n'avons rien trouvé, mais sur Java nous avons trouvé tout un framework permettant de charger et manipuler ces modèles : Java Monkey Engine. Le grand problème résidait dans l'incompatibilité avec le système de rendu de Processing.

Nous avons donc abandonné l'idée d'utiliser un outil clé en main pour la gestion des modèles. Toutefois toutes ses recherches n'ont pas été veines puisqu'elles nous ont permis de voir plusieurs façons de gérer les squelettes et les maillages, ce qui nous a inspiré pour créer notre gestion personnelle.

Solution retenue : un format personnel

Nous allons maintenant vous décrire la solution choisie pour la représentation des données. Comme pour la plus part des formats existant nous avons scindé les données en deux parties disjointes : d'un côté le maillage, de l'autre le squelette.

Représentation du modèle : obj et mtl

En ce qui concerne le maillage, nous avons opté pour une représentation classique et très populaire chez les artistes 3D : le format OBJ de Wavefront. De par sa simplicité, sa lisibilité et sa popularité, le format OBJ constitue un format de choix pour ce projet.

Il permet dans un premier temps de déclarer tous les sommets ou vertices du maillage.

Dans un second temps, il décrit l'association des sommets en faces : dans le cadre de notre projet nous limiterons ces faces à 3, nous traiterons donc des triangles. Ces triangles sont triés par groupes eux même triés par objet.

Enfin, le format OBJ permet aussi de gérer les textures grâce au format MTL (Material Template Library) en les appliquant aux triangles concernés.

Dans ces formats on utilisera donc 'v' pour déclarer un vertex, 'vt' pour déclarer un point sur la texture, 'f' pour déclarer une face, o pour déclarer un objet, 'g' pour déclarer un groupe, 'mtllib' pour ouvrir un fichier mtl et 'usemtl' pour utiliser une texture.

Un vertex est un triplé de réels représentant un point dans les trois dimensions de l'espace. Une face est ici un triplé de couple (numéro de vertex ; numéro de vertex texture).

Dans notre projet nous n'utilisons pas les vecteurs normaux, ils n'apparaissent donc pas toutefois la norme Wavefront autorise l'utilisation de ces derniers.

Le contenu général des fichiers OBJ et MTL, adopte le formalisme suivant :

```
# objetType.obj
mtllib  material.mtl
v    0.0    0.0    0.0
v    1.0    0.0    0.0
v    0.0    1.0    0.0
v    0.0    0.0    1.0

vt   0.0    0.5
vt   0.5    0.5
vt   0.0    0.0
vt   0.5    0.0

o    nomObj1
usemtl materio
g    groupe1
f    1/3    2/4    3/1
f    1/2    2/5    4/1
g    groupe2
f    2/3    3/4    4/2
f    3/4    4/2    1/1
```

```
# material.mtl
newmtl materio
    Ns        1
    d         1
    illum     2
    Kd        1.0    1.0    1.0
    Ks        0.0    0.0    0.0
    Ka        0.0    0.0    0.0
    Map_Kd    .\image.png
```

Figure 3 - Exemple de fichier OBJ et MTL

Représentation du squelette : sk et bvh

La représentation du squelette est quelque chose qui nous a tout de suite fait penser à une arborescence. En effet, on peut penser un squelette comme étant une succession d'articulations reliées par des os et convergent vers une racine : la colonne vertébrale.

Ainsi nous avons conçu un nouveau format de fichier différentiable par un suffixe en '.sk' et dont la forme est la suivante :

```
sk skeleton
n 3

j ROOT
v 0.000000 1.260000 0.000000
p null
c WAIST TORSO

j WAIST
v 0.000000 1.000000 0.000000
p ROOT
c null

j TORSO
v 0.000000 2.000000 0.000000
p ROOT
c null
```

```
# utilisation de 'sk' pour déclarer un squelette
# 'n' pour donner le nombre de joints/articulations
du squelette

# 'j' pour déclarer un nouveau joint
# 'v' pour donner sa position (x,y,z)
# 'p' pour donner le nom de son père (parent)
# 'c' pour la liste de ses fils (children)

# null signifie la présence d'une feuille
```

Figure 4 - Exemple de fichier sk

Il nous a ensuite fallu fixer le modèle d'un squelette type afin que l'on puisse faire facilement par la suite le lien entre le joint du squelette d'une Kinect et le joint du squelette de notre modèle. On est ainsi arrivé à une arborescence avec exactement 25 joints prenant

racine sur le joint nommé ROOT (racine du corps au centre de la colonne vertébrale). En annexe se trouve un exemple complet d'un squelette fonctionnel au format sk.

Toutefois notre format sk présente un désavantage majeur : il n'existe aucun moyen de générer totalement ou partiellement de façon automatique un squelette au format sk dans un logiciel type Blender. Pourtant nous générons nos modèles et nos squelettes sur Blender.

Nous avons donc trouvé un format de fichier qui peut être utilisé par Blender : le BVH ou Biovision Hierarchy. C'est un format de fichier d'animation de personnages développé par la société Biovision qui a disparue. Toutefois leur création persiste, elle permet entre autre de représenter les données issues de captures de mouvements. Or une capture de mouvement est une succession de squelettes, nous pouvons donc utiliser ce format.

Le format BVH utilise un système de hiérarchie grâce à des blocs, représentant les joints, imbriqués selon leur degré de relation. Ainsi la ROOT est le bloc principal contenant ses fils TORSO et WAIST contenant eux-mêmes leurs fils et ainsi de suite. Deuxième différence avec notre format : la position des joints est décrite de façon relative à la position de leur parent grâce à un décalage (OFFSET). Voici un exemple de la structure du fichier :

```
HIERARCHY
ROOT TORSO
{
        OFFSET 0.000000 0.000000 1.260000
        CHANNELS 6 Xposition Yposition Zposition Xrotation Zrotation Yrotation
        JOINT NECK
        {
                OFFSET 0.000000 0.000000 0.372500
                CHANNELS 3 Xrotation Zrotation Yrotation
                JOINT HEAD
                {
                        OFFSET 0.000000 -0.032200 0.122000
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                                OFFSET 0.000000 0.154500 0.016200
                        }
                }
        }
}
```

Figure 5 - Exemple de fichier BVH

La racine de cette arborescence est signalée par « ROOT », les nœuds par « JOINT » et les feuilles par « End Site ». Dans notre projet nous utilisons toujours la même hiérarchie que précédemment, le modèle du squelette au format BVH est disponible en annexe.

Les données dans le programme

Nous allons maintenant aborder la structure des données dans le programme lui-même et voir comment elles sont liées les unes aux autres.

Description des classes et attributs

Tout d'abord nous allons voir la classe **ZZkeleton** qui comme son nom l'indique correspond au squelette du modèle. Pour mieux comprendre l'étude de cette classe voici un aperçu du début de celle-ci :

```
class ZZkeleton {
    public final static int WAIST           = 0;
    public final static int ROOT            = 1;
    public final static int NECK            = 2;
    public final static int HEAD            = 3;
    public final static int SHOULDER_LEFT   = 4;
    public final static int ELBOW_LEFT      = 5;
    public final static int WRIST_LEFT      = 6;
    public final static int HAND_LEFT       = 7;
    public final static int SHOULDER_RIGHT  = 8;
    public final static int ELBOW_RIGHT     = 9;
    public final static int WRIST_RIGHT     = 10;
    public final static int HAND_RIGHT      = 11;
    public final static int HIP_LEFT        = 12;
    public final static int KNEE_LEFT       = 13;
    public final static int ANKLE_LEFT      = 14;
    public final static int FOOT_LEFT       = 15;
    public final static int HIP_RIGHT       = 16;
    public final static int KNEE_RIGHT      = 17;
    public final static int ANKLE_RIGHT     = 18;
    public final static int FOOT_RIGHT      = 19;
    public final static int TORSO           = 20;
    public final static int INDEX_LEFT      = 21;
    public final static int THUMB_LEFT      = 22;
    public final static int INDEX_RIGHT     = 23;
    public final static int THUMB_RIGHT     = 24;

    protected float[][] lastRotation;
    private ZZoint[] joints;
    protected int jointsNumber;
    protected String name;
```

Figure 6 - Attributs de la classe ZZkeleton

Comme on peut le voir ci-dessus, nous avons défini des constantes permettant d'accéder rapidement à des joints. Ces constantes ont les mêmes valeurs que celles de la librairie Kinect ce qui nous permet de faire le lien entre les différents squelettes très rapidement, en O(1).

Ensuite on peut voir sur cet extrait l'existence du tableau « joints ». Son type est **ZZoint**, une classe héritée de plusieurs générations de **PVector** et dont la particularité est, entre autre, d'avoir un champ pour l'indice d'un père et les indices d'enfants. On peut ainsi créer une arborescence de joints donc un squelette.

Continuons avec l'étude de la classe **ZZModel** qui va centraliser l'ensemble des informations concernant le modèle du personnage, à savoir : le maillage à base de **PShape**, le squelette de type **ZZkeleton** et les matériaux de type **ZZMaterial**.

Comme on peut le voir dans l'extrait de code suivant, la classe **ZZModel** contient certes une PShape permettant d'afficher le modèle à l'écran, mais surtout les attributs

« groups » et « vertices » qui jouent un rôle majeur dans la modification et donc l'animation du personnage. En effet, « groups » est un tableau à deux dimensions indiquant pour chaque groupe quelles vertices de « vertices » lui sont associées grâce à leurs indices. Ensuite « vertices » pointe pour chaque vertex vers toutes les occurrences dans toutes les faces où ce même vertex est présent, ce qui n'est pas le cas dans PShape. Nous avons ainsi un accès pour modification en O(1) à tous nos groupes représentant toutes les parties du corps de nos modèles.

Il faut aussi noter qu'un indice de groupe correspond exactement à une constante représentant le joint pour la Kinect, donc un groupe correspond à un joint.

```
class ZZModel {
    protected PApplet app;
    protected PShape model;
    protected ZZkeleton skeleton;
    private ZZkeleton basisSkel; // squelette de base du modele
    private PShape basisMod; // modele de base du modele
    private ArrayList<ZZertex> basisVert;
    ArrayList<ZZertex> vertices;
    ArrayList<ZZector> vertiTexture;
    ArrayList<Integer>[] groups;
    ArrayList<ZZMaterial> materiel = null;
    protected int idUser = 0; // determine le numero du joueur
```

Figure 7 - Attributs de la classe ZZModel

Cette classe comporte aussi des attributs permettant d'effectuer une remise à zéro mais nous verrons ça ultérieurement.

Chargement en mémoire

Les fonctions de chargement font le lien entre la représentation des données dans les fichiers et dans le programme.

Le chargement du maillage correspond à une lecture séquentielle et linéaire des lignes du fichier. En voici le principe :

- On crée une nouvelle PShape ;
- Lecture du fichier obj
- Pour chaque ligne faire
    - Si ligne contient "v " alors on ajoute un vertex à vertices ;
    - Si ligne contient "vt " alors on ajoute un vertex-texture ;
    - Si ligne contient "mtllib " alors on charge les textures ;
    - Si ligne contient "f " alors
        - On crée une nouvelle face ;
        - On la texture ;
        - On enregistre dans quels groupes chaque vertex va ;
        - On ajoute la face au groupe courant ;
    - Si ligne contient "g " alors
        - On crée un nouveau groupe courant ;
        - On l'ajoute au model ;
- Fin pour

Concernant les textures et les squelettes on a quelque chose de très similaire : selon le mot clé trouvé dans une ligne on sait qu'on va charger telle ou telle information.

Contrôle et amélioration

Animation de l'avatar

Principe de l'algorithme

Le principe de l'algorithme d'animation est le suivant :

- On dispose de la position de l'utilisateur à un certain moment
- Pour chaque partie du modèle, faire
  - Calculer la différence d'angle entre la position actuelle du membre de l'utilisateur et la position du membre dans le modèle
  - Appliquer la rotation du membre
- Fin pour

L'animation se fait sur plus ou moins de membres en fonction de la Kinect en cours d'utilisation.

De plus, la position du modèle dans la scène est définie par la position du Torse dans la réalité.

Application au modèle

Le modèle est géré avec la classe **ZZModel**. Nous disposons des informations détaillées dans la Figure 7.

La lecture et la modification de ces informations vont permettre l'animation. Nous avons une méthode principale qui va appeler la méthode suivante avec toutes les parties de l'avatar que l'on peut animer (selon la Kinect) :

```
private void movePart(int part, ZZoint [] mouv) {
    // declaration des variables
    PVector v1,v2;
    float f1, f2, f3, f4;

    // calcul des vecteurs
    v1 = mouv[part].copy();
    v1.sub(mouv[skeleton.getJoint(part).getParent()]);
    v2 = this.skeleton.getJoint(part).copy();
v2.sub(this.skeleton.getJoint(skeleton.getJoint(part).getParent()));

    // calcul des angles entre les vecteurs
    f1 = PApplet.atan2(v1.y, v1.x);
    f2 = PApplet.atan2(v2.y, v2.x);
    f3 = f1 - f2; // angle theta

    f1 = PApplet.atan2(v1.z, v1.x);
    f2 = PApplet.atan2(v2.z, v2.x);
    f4 = f1 - f2; // angle phi

    rotatePart(part, f3, f4, 0); // application des rotations
}
```

Figure 8 - Extrait de la méthode d'animation d'une partie du modèle

Cette méthode est sujette à des améliorations que nous allons aborder dans la partie suivante.

Améliorations du mouvement

L'algorithme précédent permet effectivement d'animer notre avatar. Cependant on peut constater que les mouvements ne sont pas forcément fidèles et que certains soucis apparaissent au niveau de l'orientation des membres.

Pour remédier à cela, nous avons mis en places plusieurs techniques qui corrigent ces défauts.

Remise à zéro du modèle

La première amélioration testée afin de rendre plus réalistes les mouvements de l'avatar a été de réinitialiser celui-ci avant chaque modification de l'orientation des membres.

En effet, après de nombreuses rotations des membres, il était possible que certaines parties du corps de l'avatar n'aient pas la bonne orientation (par exemple, le dessus l'avant-bras se retrouvait à l'envers).

Pour pallier à cela, à chaque chargement d'un modèle, nous stockons sa position initiale (squelette et vertices). Ainsi, les mouvements se feront toujours par rapport à la position de base de l'avatar, ce qui permet de limiter les mauvais mouvements.

La réinitialisation du modèle s'effectue avant que les rotations ne se fassent sur les membres (donc au tout début de la fonction de mouvement). L'affichage du modèle se faisant après cette rotation, nous ne voyons donc pas la phase de remise à zéro du modèle.

Dans cette partie, il a fallu veiller à bien faire une recopie des données lors de l'initialisation ou de la remise à zéro. En effet, il faut recopier intégralement le squelette et les vertices du modèle à chaque remise à zéro. C'est pourquoi les performances se trouvent légèrement dégradées. De plus, étant donné que l'on doit stocker un squelette supplémentaire et un tableau de vertices en plus, la structure de l'avatar s'en retrouve donc alourdie.

Lissage sur plusieurs échantillons

Un autre moyen d'améliorer l'animation du modèle est d'effectuer un moyennage sur un certain nombre de mesures. Dans le programme, nous ferons un lissage sur 3 mesures. Ce nombre est modifiable dans le code grâce à la constante **NBCAPT**.

Afin d'effectuer le lissage sur les données recueillies (les positions de l'utilisateur), nous stockons les positions successives de l'utilisateur jusqu'à en avoir le nombre requis, alors nous les moyennons. Ensuite nous utilisons la fonction **lerp** prédéfinie qui effectue une interpolation entre la position actuelle et la nouvelle position moyennée et retourne plusieurs positions lissées du mouvement de l'utilisateur.

```
if (cptEch==nbEch) {     // si on a tous les echantillons
    cptEch = 0;
    ZZoint.div(arrivee, nbEch);
    for (int i = 1; i <= nbEch; i++) {
        positionsOptimisees.put(ZZoint.lerp(depart, arrivee, i/nbEch));
    }
    depart = arrivee;    // verifier la copie
}
```

Figure 9 - Extrait de la méthode de lissage

Ces positions ainsi obtenues sont utilisées pour calculer les angles de rotation de chaque membre de l'avatar.

Ceci a pour effet de réduire le taux de rafraichissement de l'avatar, car il faut attendre d'avoir tous les échantillons pour ensuite faire le moyennage et pouvoir faire bouger l'avatar. On peut cependant remarquer Qu'il y a moins d'erreurs dans les mouvements de l'avatar qui sont plus fidèles à ceux de l'utilisateur.

Delta de modification d'un membre

Toujours dans le but d'améliorer les mouvements de l'avatar, le suivi des angles a été mis en place.

Désormais dans la classe permettant la gestion du squelette, nous disposons d'un tableau stockant les angles de la dernière rotation uniquement, pour chaque partie du squelette. Les 3 angles sont stockés (selon l'axe x, y et z).

Ainsi avant d'appliquer la rotation d'un membre, on vérifie si la différence entre l'angle à appliquer et l'ancien angle est supérieure à un delta (noté p dans la fonction de rotation de la partie). Si tel est le cas, nous appliquons cet angle sur le membre concerné et nous modifions le tableau des rotations.

```
double p = 0.3; // seuil pour effectuer la rotation

if (app.abs(app.abs(skeleton.getLastRotation(part)[0])-app.abs(f3)) > p ||
    app.abs(app.abs(skeleton.getLastRotation(part)[1]) - app.abs(f4)) > p)
{
    rotatePart(part, f3, f4, 0); // application des rotations
}
```

Figure 10 - Extrait d'amélioration du delta de mouvement dans ZZModel

Cette amélioration permet d'éviter les « tremblements » de l'avatar causés par les petits mouvements du corps de l'utilisateur. Les performances d'animation de l'avatar ne sont pas impactée étant donné que les calculs pour déterminer si la rotation est effectuée ou non sont simples.

Conclusion

Bilan du projet

A partir de rien, nous avons donc réussi à réaliser un programme complexe permettant de contrôler un avatar virtuel en 3D de façon impressionnante dans un temps imparti très court.

Nous avons conçu tout un système de gestion des modèles et de leur squelette ce qui nous a permis de mieux appréhender le travail sur des données 3D.

Puis l'animation du modèle, c'est-à-dire la rotation de chaque membre autour des articulations a été l'occasion de mettre en perspective nos cours de mathématiques sur la géométrie des surfaces dans l'espace.

Ensuite l'intégration des données Microsoft Kinect 1 et 2 a été très instructives et nous pensons maintenant avoir une bonne connaissance technique sur ce matériel suite aux nombreux problèmes rencontrés.

Enfin, tout ce temps passé à utiliser les bibliothèques Processing et Processing lui-même nous a été très bénéfique. L'architecture générale et même des domaines très précis nous sont aujourd'hui très familier. Ceci est dû selon nous au fait d'avoir décidé de développer sous un environnement tiers (Eclipse) au lieu de rester sur l'IDE originel où la plupart des mécaniques sont masquées ou simplifiées.

Améliorations possibles

Par manque de temps et aussi parce qu'il est bien difficile de mettre des limites à un projet aussi ouvert dans le nombre de possibilités, nous n'avons pas pu développer certaines choses que nous tenons à présenter ici.

L'animation de l'avatar est assez fidèle à la réalité. Il est cependant possible de gérer le fait que l'utilisateur soit de face ou de dos, ce qui n'est pas implémenté pour le moment. En effet, nous avons voulu faire un programme le plus intuitif possible, or quand l'utilisateur regarde l'écran il s'attend à se voir de face.

Pour améliorer encore plus la stabilité du modèle et ainsi éviter les retournements intempestifs de membres, nous avons trouvé une solution supplémentaire. La Kinect offre la possibilité de connaitre la position mais aussi l'orientation de chaque joint du corps de l'utilisateur. Cette information se trouve sous forme d'une matrice reprenant les informations des quaternions de ces joints. Connaissant l'orientation de chaque joint dans l'espace, il est alors possible d'appliquer un post-traitement au modèle afin de corriger les éventuels membres retournés grâce à une méthode assez identique à celle du mouvement simple.

Concernant le prétraitement des données, il serait assez intéressant de mettre en place un programme permettant de générer de façon plus automatique le modèle utilisable dans le programme. On pourrait tout à fait concevoir un programme annexe qui prendrait en entrée un maillage au format obj (wavefront) et qui appliquerait un algorithme de squelettisation spécial et donnerait en sortie un fichier obj avec les groupes de faces et le fichier bvh associé.

Enfin une dernière modification à laquelle nous avons pensé était de mettre en place une indépendance des mouvements du modèle par rapport au squelette de l'utilisateur. Ainsi un utilisateur, gardant une attitude humaine, pourrait contrôler un T-Rex gardant son attitude de prédateur préhistorique. Or à l'heure actuelle, le T-Rex adopte avec notre programme une attitude humaine.

Table des figures

Bibliographie

- Site web Processing, www.processing.org
- Forum de Processing, forum.processing.org
- Documentation SimpleOpenNi, code.google.com/p/simple-openni
- Documentation Proclipsing, code.google.com/p/proclipsing
- Documentation KinectPV2 : github
- Wikipédia (Définitions, formats)
- Recherche diverses : www.google.com

Annexes

Création d'un avatar

Nous allons à présent vous donner les indications les plus précises possibles pour créer vous-même vos propres avatars à animer :

ETAPE 1

La première étape consiste tout simplement à choisir un modèle ou mesh qui vous plaît et l'importer sous Blender.

ETAPE 2

Ensuite il convient de lui assigner l'armature appropriée avec les bons noms (il est recommandé d'utiliser ./SkeletonTemplate.blend).

ETAPE 3

Veillez à ce que l'armature corresponde bien à l'ossature du modèle, ensuite sélectionnez le mesh puis l'armature.

ETAPE 4

Nous allons maintenant lier le chaque face du maillage à un os du squelette en faisant comme suit : Ctrl+P > Set Parent to > Armature Deform > With Automatic Weights.

ETAPE 5

Une fois ceci fait on peut exporter les données résultantes :

- On exporte le nouveau maillage : File > Export > Wavefront (.obj) avec les options : "Include UVs" "Write Materials" "Triangulate Faces" "Polygroups" "Objects as OBJ Objects".
- Ensuite on exporte le squelette dans le format bvh : File > Export > Motion Capture (.bvh).

ETAPE 6

Dans Misfit Model 3D, chargez le mesh et vérifiez (et corrigez) l'assignement automatique des faces aux groupes. Pour ce faire aller dans le menu "Groups > Edit Groups...". En particulier vérifiez "No group" et d'éventuels "(null)".

ETAPE 7

Exportez le résultat obtenu depuis Misfit (ne pas sauver les normales).

ETAPE 8

Copiez tous les fichiers dans le répertoire data du programme. Attention toutefois à veiller à bien donner le même nom à tous les fichiers.

Utilisation de Processing sous Eclipse

L'environnement de développement Processing étant assez limité en termes de clarté de code et d'indentation, nous avons choisi et développer le projet sous Eclipse grâce à un plug-in.

Le plug-in en question est Proclipsing et permet facilement de créer du code Processing en utilisant les mêmes bibliothèques que sous Processing.

Pour l'installer il faut simplement ajouter le plug-in dans Eclipse : Help > Install new software > Add. Il faut ensuite saisir le nom du plug-in, ici « Proclipsing » et indiquer l'URL d'installation (http://proclipsing.googlecode.com/svn/tags/current_releases/proclipsingSite/). Après confirmation du choix, on choisit tous les composants et le plug-in s'installe.

Une fois le plug-in installé, il ne reste plus qu'à le paramétrer en allant dans les préférences et choisissant Proclipsing. Il faut indiquer le répertoire Processing, le répertoire des librairies et le répertoire des sketchs.

Lorsque ceci est effectué, on peut utiliser Eclipse pour créer du code Processing.

Dans un premier temps, on crée la classe principale du projet qui hérite de **PApplet**. Puis on définit les méthodes setup() et draw(), de base sous Processing. On ajoute également la méthode suivante :

```
public static void main(String _args[]) {
	PApplet.main(new String[]{zzavatar.ZZavatar.class.getName() });
}
```

Cette méthode permet le bon fonctionnement du projet dans Eclipse. A partir de maintenant, il est possible de créer son code comme sous Processing. Il est également possible de créer d'autres classes afin de mieux séparer les données comme c'est le cas dans ce projet.

Squelette fonctionnel au format sk

```
sk skeleton
n 25

j ROOT
v 0.000000 1.260000 0.000000
p null
c WAIST TORSO

j WAIST
v 0.000000 1.000000 0.000000
p ROOT
c HIP_LEFT HIP_RIGHT

j HIP_LEFT
v -0.129800 1.029400 0.014900
p WAIST
c KNEE_LEFT

j KNEE_LEFT
v -0.183600 0.585200 -0.016900
p HIP_LEFT
c ANKLE_LEFT

j ANKLE_LEFT
v -0.232200 0.110000 0.074200
p KNEE_LEFT
c FOOT_LEFT

j FOOT_LEFT
v -0.233600 0.024400 -0.145000
p ANKLE_LEFT
c null

j HIP_RIGHT
v 0.129800 1.029400 0.014900
p WAIST
c KNEE_RIGHT

j KNEE_RIGHT
v 0.183600 0.585200 -0.016900
p HIP_RIGHT
c ANKLE_RIGHT

j ANKLE_RIGHT
v 0.232200 0.110000 0.074200
p KNEE_RIGHT
c FOOT_RIGHT

j FOOT_RIGHT
v 0.233600 0.024400 -0.145000
p ANKLE_RIGHT
c null

j TORSO
v 0.000000 1.632500 0.000000
p ROOT
c NECK SHOULDER_LEFT
SHOULDER_RIGHT

j NECK
v 0.000000 1.754500 0.032200
p TORSO
c HEAD

j HEAD
v 0.000000 1.870700 -0.122300
p NECK
c null

j SHOULDER_LEFT
v -0.228100 1.652200 0.083470
p TORSO
c ELBOW_LEFT

j ELBOW_LEFT
v -0.489000 1.623600 0.071800
p SHOULDER_LEFT
c WRIST_LEFT

j WRIST_LEFT
v -0.771100 1.621100 0.010000
p ELBOW_LEFT
c HAND_LEFT

j HAND_LEFT
v -0.793100 1.621000 -0.024000
p WRIST_LEFT
c INDEX_LEFT THUMB_LEFT

j INDEX_LEFT
v -0.918900 1.598400 -0.080000
p HAND_LEFT
c null

j THUMB_LEFT
v -0.862600 1.578600 -0.080000
p HAND_LEFT
c null

j SHOULDER_RIGHT
v 0.228100 1.652200 0.083470
p TORSO
c ELBOW_RIGHT

j ELBOW_RIGHT
v 0.489000 1.623600 0.071800
p SHOULDER_RIGHT
c WRIST_RIGHT

j WRIST_RIGHT
v 0.771100 1.621100 0.010000
p ELBOW_RIGHT
c HAND_RIGHT

j HAND_RIGHT
v 0.793100 1.621000 -0.024000
p WRIST_RIGHT
c INDEX_RIGHT THUMB_RIGHT

j INDEX_RIGHT
v 0.918900 1.598400 -0.080000
p HAND_RIGHT
c null

j THUMB_RIGHT
v 0.862600 1.578600 -0.080000
p HAND_RIGHT
c null
```

Squelette fonctionnel au format bvh

```
HIERARCHY
ROOT TORSO
{
    OFFSET 0.000000 0.000000 1.260000
    CHANNELS 6 Xposition Yposition Zposition Xrotation Zrotation Yrotation
    JOINT NECK
    {
        OFFSET 0.000000 0.000000 0.372500
        CHANNELS 3 Xrotation Zrotation Yrotation
        JOINT HEAD
        {
            OFFSET 0.000000 -0.032200 0.122000
            CHANNELS 3 Xrotation Zrotation Yrotation
            End Site
            {
                OFFSET 0.000000 0.154500 0.016200
            }
        }
    }
    JOINT SHOULDER_LEFT
    {
        OFFSET 0.000000 0.000000 0.372500
        CHANNELS 3 Xrotation Zrotation Yrotation
        JOINT ELBOW_LEFT
        {
            OFFSET -0.228100 -0.083470 0.019700
            CHANNELS 3 Xrotation Zrotation Yrotation
            JOINT WRIST_LEFT
            {
                OFFSET -0.260900 0.011670 -0.028600
                CHANNELS 3 Xrotation Zrotation Yrotation
                JOINT HAND_LEFT
                {
                    OFFSET -0.282100 0.061800 -0.002500
                    CHANNELS 3 Xrotation Zrotation Yrotation
                    JOINT THUMB_LEFT
                    {
                        OFFSET -0.022000 0.034000 -0.000100
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                            OFFSET -0.069500 0.056000 -0.042400
                        }
                    }
                    JOINT INDEX_LEFT
                    {
                        OFFSET -0.022000 0.034000 -0.000100
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                            OFFSET -0.125800 0.056000 -0.022600
                        }
                    }
                }
            }
        }
    }
    JOINT SHOULDER_RIGHT
```

```
    {
        OFFSET 0.000000 0.000000 0.372500
        CHANNELS 3 Xrotation Zrotation Yrotation
        JOINT ELBOW_RIGHT
        {
            OFFSET 0.228100 -0.083470 0.019700
            CHANNELS 3 Xrotation Zrotation Yrotation
            JOINT WRIST_RIGHT
            {
                OFFSET 0.260900 0.011670 -0.028600
                CHANNELS 3 Xrotation Zrotation Yrotation
                JOINT HAND_RIGHT
                {
                    OFFSET 0.282100 0.061800 -0.002500
                    CHANNELS 3 Xrotation Zrotation Yrotation
                    JOINT THUMB_RIGHT
                    {
                        OFFSET 0.022000 0.034000 -0.000100
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                            OFFSET 0.069500 0.056000 -0.042400
                        }
                    }
                    JOINT INDEX_RIGHT
                    {
                        OFFSET 0.022000 0.034000 -0.000100
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                            OFFSET 0.125800 0.056000 -0.022600
                        }
                    }
                }
            }
        }
    }
    JOINT WAIST
    {
        OFFSET 0.000000 0.000000 0.000000
        CHANNELS 6 Xposition Yposition Zposition Xrotation Zrotation Yrotation
        JOINT HIP_RIGHT
        {
            OFFSET 0.000000 0.000000 -0.260000
            CHANNELS 3 Xrotation Zrotation Yrotation
            JOINT KNEE_RIGHT
            {
                OFFSET 0.129800 -0.014900 0.029400
                CHANNELS 3 Xrotation Zrotation Yrotation
                JOINT ANKLE_RIGHT
                {
                    OFFSET 0.053800 0.031800 -0.444200
                    CHANNELS 3 Xrotation Zrotation Yrotation
                    JOINT FOOT_RIGHT
                    {
                        OFFSET 0.048600 -0.091100 -0.475200
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                            OFFSET 0.001400 0.219200 -0.085600
                        }
```

```
                    }
                }
            }
        }
        JOINT HIP_LEFT
        {
            OFFSET 0.000000 0.000000 -0.260000
            CHANNELS 3 Xrotation Zrotation Yrotation
            JOINT KNEE_LEFT
            {
                OFFSET -0.129800 -0.014900 0.029400
                CHANNELS 3 Xrotation Zrotation Yrotation
                JOINT ANKLE_LEFT
                {
                    OFFSET -0.053800 0.031800 -0.444200
                    CHANNELS 3 Xrotation Zrotation Yrotation
                    JOINT FOOT_LEFT
                    {
                        OFFSET -0.048600 -0.091100 -0.475200
                        CHANNELS 3 Xrotation Zrotation Yrotation
                        End Site
                        {
                            OFFSET -0.001400 0.219200 -0.085600
                        }
                    }
                }
            }
        }
    }
}
```

Glossaire

A

Animation squelettale
Principe d'animation d'un modèle par le moyen d'un squelette 8
Attitude de prédateur préhistorique
Posture inclinée portée sur l'avant, cf Jurassic Park 18
Attitude humaine
Posture droite de bipède évolué 18

D

Debug
Mode dans lequel des informations supplémentaires sont apportées sur l'exécution du programme 4

F

FPS ou "Frames per seconds"
Images par secondes 5, 6
Framework
Ensemble de logiciels ou bibliothèques destinés à une utilisation particulière 9

I

IDE ou Environnement de Développement Intégré
Environnement permettant le développement de programmes avec l'aide de nombreux outils (=éditeur de texte++) 17
Interpolation
Calcul d'une position transitoire entre deux points connus 15

M

Maillage
Modélisation en "fil de fer" du personnage ou bien représentation sous forme de graphe(s) d'un objet 3D 4, 9, 18
Mono-utilisateur
Programme ne pouvant être utilisé que par un seul utilisateur à la fois....4

O

Os ou Bone
Segment reliant deux articulations d'un modèle 10, 20

P

Plug-in
Module permettant l'ajout de fonctionnalités périphériques à un programme sans en affecter le code principal 21
Post-traitement
Traitement effectué avant d'afficher le résultat final 17
Projet
Module permettant d'être évalué sur un sujet génial 3

Q

Quaternion

Structure complexe contenant des nombres fournissant des informations à la suite de plusieurs opérations sur ceux-ci 17

S

Squelettisation

Création d'un squelette à partir d'un maillage par différents procédés (érosion, régression) 18

T

Taux de rafraichissement

Nombre d'images affichées par secondes 6, 16

V

Vertex (-ices)

Sommet(s) ou point(s) dans un maillage 4, 8