Animation de personages 3D : Mokujin

IN55 Synthèse d'images - P2016

Collaborateurs :

JARDINO Thibauld GI04
JAFFALI Hamza GI04
BLEIN Thibaud GI04

Enseignant:

LAURI Fabrice

utbm
université de technologie
Belfort-Montbéliard

OpenGL

Sommaire

Introduction

Dans le cadre de notre spécialité Imagerie, Interactions et Réalité Virtuelle 4e année en école d'ingénieur à l'UTBM, nous avons tout trois choisit d'intégrer le cours d'IN55 : Synthèses d'image. Durant ce cours nous avons appris les bases du rendu d'images et d'animations, et c'est suite à cela que nous avons choisi l'animation d'un personnage comme projet.

Pour notre projet nous voulions pouvoir tout réaliser un maximum par nous-mêmes. Il nous fallait donc afin de pouvoir assurer le projet dans les temps avec les travaux pour les autres cours, un model simple. Ayant été bercé depuis notre jeunesse par les jeux-vidéos, et notamment les jeux de combat, nous avons eu l'idée de réaliser quelques animation sur Mokujin, un personnage emblématique de Tekken réalisé presque uniquement de cylindre, demi-cylindre et sphères. Le principe de ce personnage est de calquer les techniques de ses adversaires et n'a donc qu'une animation de victoire qui lui est propre.

Ce rapport présentera donc cette réalisation, d'abord la modélisation sous Blender, puis le parsing, ensuite le rendu et la texture puis finalement nous aborderons l'animation.

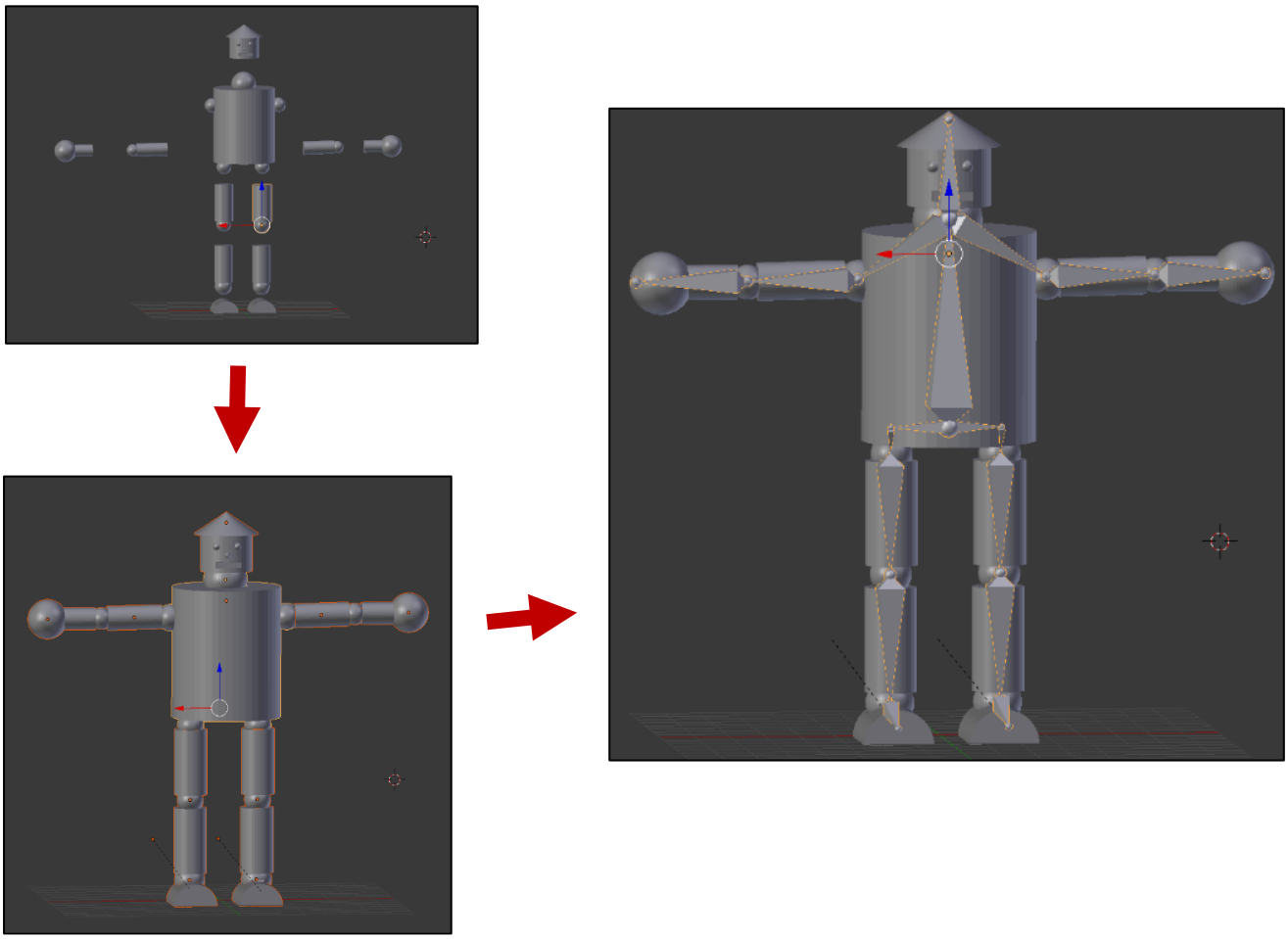
La modélisation sous Blender

Afin de réaliser un corps plus simplement qu'en générant une somme de fonction qui auraient pu générer un corps géométrique nous avons décidé de générer le corps sous Blender.

Ce travail a été découpé en plusieurs parties :

- réaliser une à une les classes d'équivalences cinématique :
  - Tête
  - Bassin
  - Deux parties des bras
  - Deux parties des jambes
  - Pied
- assembler les parties
- réaliser un squelette pour une éventuelle réutilisation pour les animations ou une utilisation postérieur de l'objet
- exporter dans différents formats avec différents paramètres afin de pouvoir trouver le plus adéquat

Ci-dessous la représentation de l'enchainement des étapes.



Le Parsing du fichier généré

Afin de charger le modèle blender de notre Mokujin dans OpenGL, nous avons dû utiliser un parser permettant la récupération des différentes parties de notre personnage.

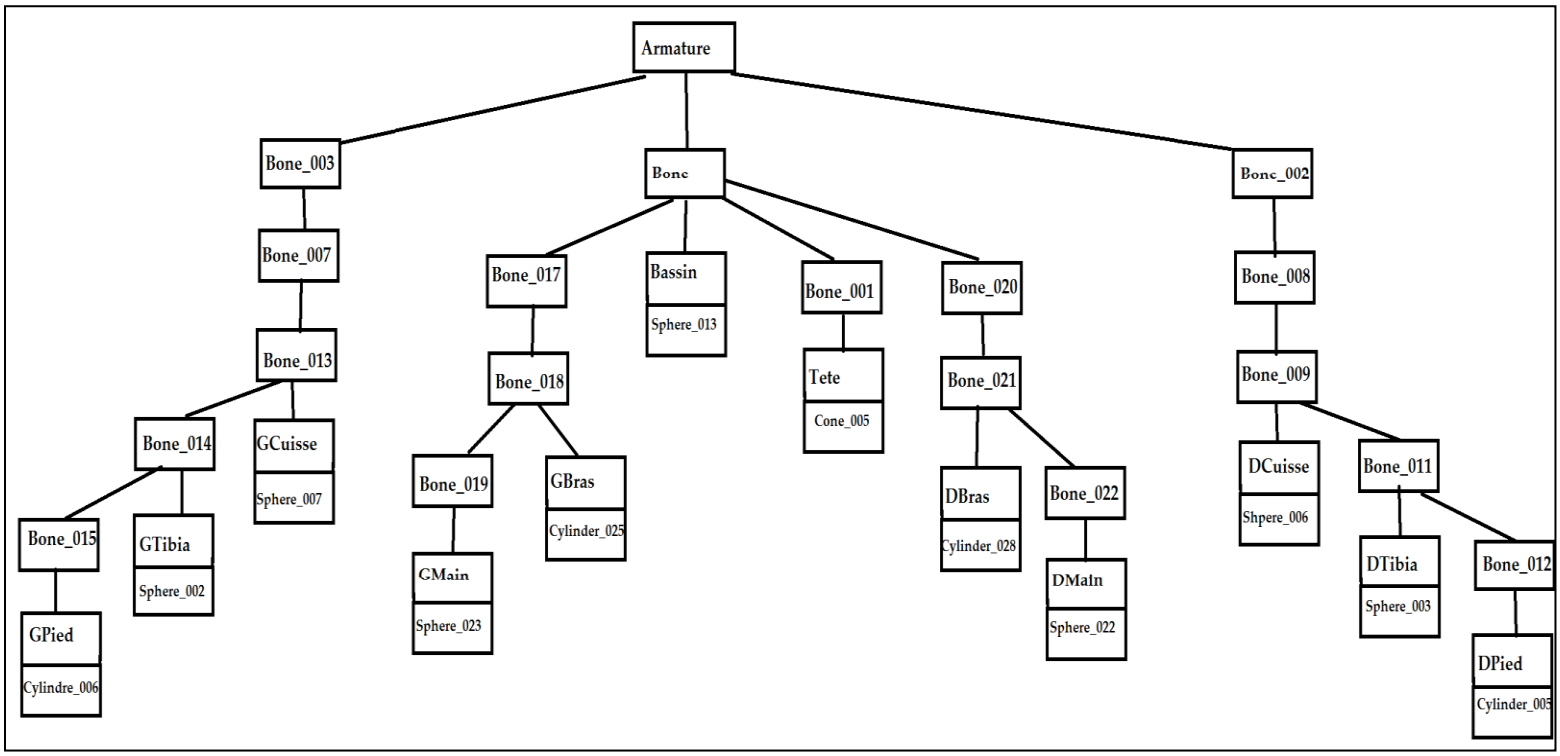
Pour cela, nous avons utilisé Assimp qui est un parser de fichier assez puissant puisqu'il permet de lire un grand nombre de formats différents d'exportation de modèle 3D.

Ici nous avons décidé de parser notre Mokujin au format fbx.
Malheureusement, Assimp a pour but de parser le fichier mais ne propose pas un moyen efficace de le stocker pour faciliter l'animation de notre modèle. En effet, les informations sont catégorisées et mises dans de grandes listes ne permettant que son parcours simple.

Il faut donc aménager des structures pour répondre à cette problématique. Nous avons créé 4 structures :

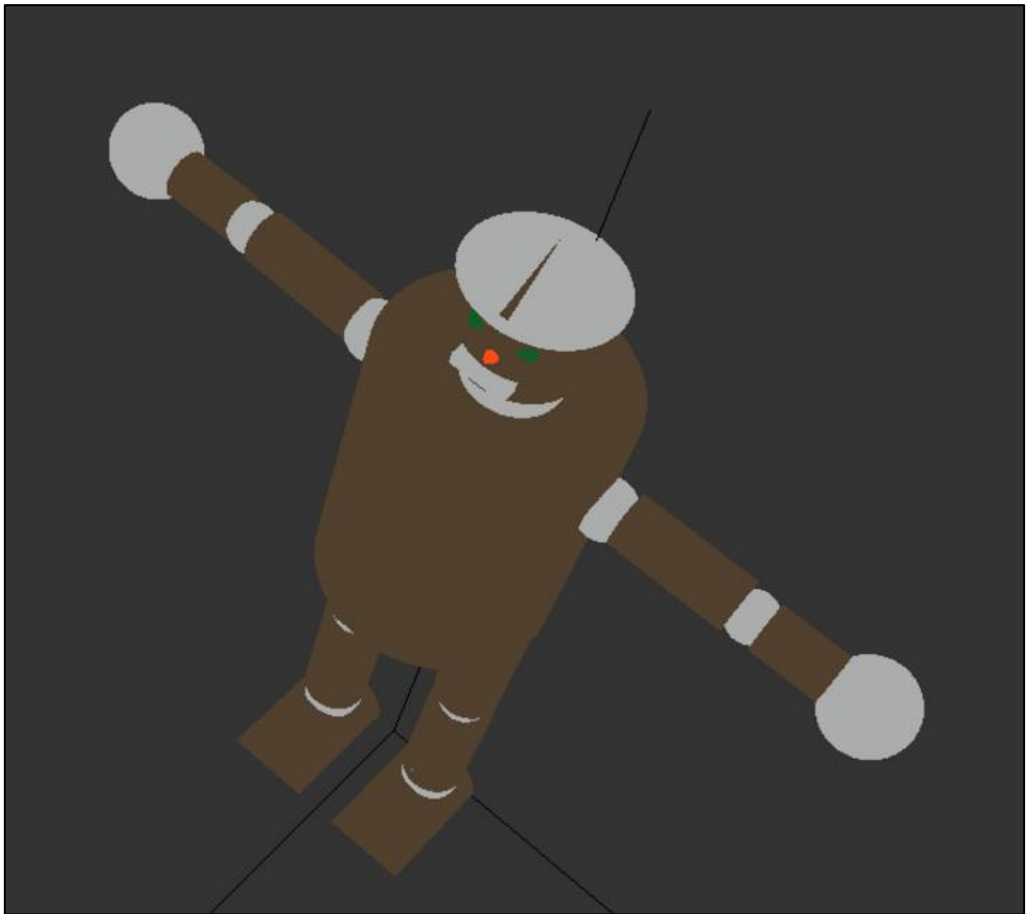
```
struct MaterialInfo
{
    QString Name;
    QVector3D Ambient;
    QVector3D Diffuse;
    QVector3D Specular;
    float Shininess;
};

struct LightInfo
{
    QVector4D Position;
    QVector3D Intensity;
};

struct Mesh
{
    QString name;
    unsigned int indexCount;
    unsigned int indexOffset;
    QVector<float> m_vertices;
    QVector<unsigned int> m_indices;
    QSharedPointer<MaterialInfo> material;
};

struct Node
{
    QString name;
    QMatrix4x4 transformation;
    unsigned int nbMeshes;
    QVector<QSharedPointer<Mesh> > meshes;
    QVector<Node> nodes;
};
```

Les deux utilisées pour l'animation du modèle sont Node et Mesh. Chaque node peut avoir plusieurs Mesh. Comme vous pouvez le voir, le fait que Node contienne QVector<Node> nodes nous permet d'écrire un arbre qui constitue le corps entier de notre Mokujin. Le but étant d'avoir transmission du mouvement entre le bras et l'avant-bras par exemple vu que l'on construit notre corps récursivement.



Application des textures

*Pour la texture, nous avons joué le jeu à 100 % en Opengl. Nous avons codé la classe texture nous permettant de charger et utiliser n'importe quelle image ! Dans un premier temps, nous avons utilisés les mêmes fonctions qui sont proposées dans la fonction createTexture de GlFrameWork avec Qimage. Mais encore, pour aller plus loin, nous avons inclus la librairie SOIL (***Simple OpenGL Image Library***).*



Réalisation de la caméra libre

L'implémentation d'une caméra mobile est une étape importante dans un projet de visualisation de la scène 3D, et permet à l'utilisateur de disposer de fonctionnalités simples comme se déplacer dans la scène, et regarder dans différentes directions. Cette étape a été guidée par le TP3 de IN55, qui nous fournit un Template de base pour les classes Quaternion et Camera.

Implémentation de la classe Quaternion

Le concept mathématique de quaternion est très utilisé dans les métiers de l'image car il permet d'implémenter plus rapidement les rotations dans l'espace, comme le permettent les complexes dans le plan.

Pour implémenter la classe Quaternion et les méthodes correspondantes, nous nous appuyons sur le diagramme UML ci-contre. Les opérateurs de multiplications et d'additions s'implémentent à partir des formules mathématiques connues concernant les quaternions. Nous avons la possibilité de multiplier des quaternions entre eux pour composer des rotations, ou d'appliquer un quaternion à un vecteur de l'espace pour le transformer par une rotation.

Nous avons également implémenté la possibilité de définir un quaternion à partir d'une matrice de rotation, en utilisant les formules données en cours. Une fonctionnalité importante de cette classe est également de pouvoir interpoler deux quaternions, représentant des orientations par exemple, à l'aide de la méthode SLERP. Pour ce faire les données du cours sont également utiles, et c'est une fonctionnalité qui pourra être utilisée pour réaliser un travelling de la caméra d'une scène par exemple, ou permet d'interpoler deux orientations, ce qui peut être utile dans la mise en place d'animations.

	$T > 0$	$T \leq 0$ $r_{00} = \max\{r_{00}, r_{11}, r_{22}\}$	$T \leq 0$ $r_{11} = \max\{r_{00}, r_{11}, r_{22}\}$	$T \leq 0$ $r_{22} = \max\{r_{00}, r_{11}, r_{22}\}$
s	$(\sqrt{T+1})/2$	$(r_{12} - r_{21})/(4x)$	$(r_{20} - r_{02})/(4y)$	$(r_{01} - r_{10})/(4z)$
x	$(r_{21} - r_{12})/(4s)$	$(\sqrt{r_{00} - r_{11} - r_{22} + 1})/2$	$(r_{01} + r_{10})/(4y)$	$(r_{02} + r_{20})/(4z)$
y	$(r_{02} - r_{20})/(4s)$	$(r_{01} + r_{10})/(4x)$	$(\sqrt{r_{11} - r_{00} - r_{22} + 1})/2$	$(r_{12} + r_{21})/(4z)$
z	$(r_{10} - r_{01})/(4s)$	$(r_{02} + r_{20})/(4x)$	$(r_{12} + r_{21})/(4y)$	$(\sqrt{r_{22} - r_{00} - r11 + 1})/2$

Implémentation de la classe Camera

La caméra libre est une composante de base d'une scène 3D, car c'est d'elle que va dépendre la vue finale de l'observateur. Ainsi, avoir la possibilité de changer d'angle de vue dans une scène, et de pouvoir s'y déplacer à sa guise est une fonctionnalité prépondérante, permettant de mettre en valeur le concept même de scène en trois dimensions.

Afin de modéliser une caméra libre dans l'espace, il convient en effet d'assigner une position à cette caméra, ainsi qu'une direction selon laquelle elle devra pointer. La position peut être stockée dans un vecteur à 3 dimensions, et pour plus de commodité et d'efficacité dans les calculs, le vecteur d'orientation est modélisé par un quaternion.

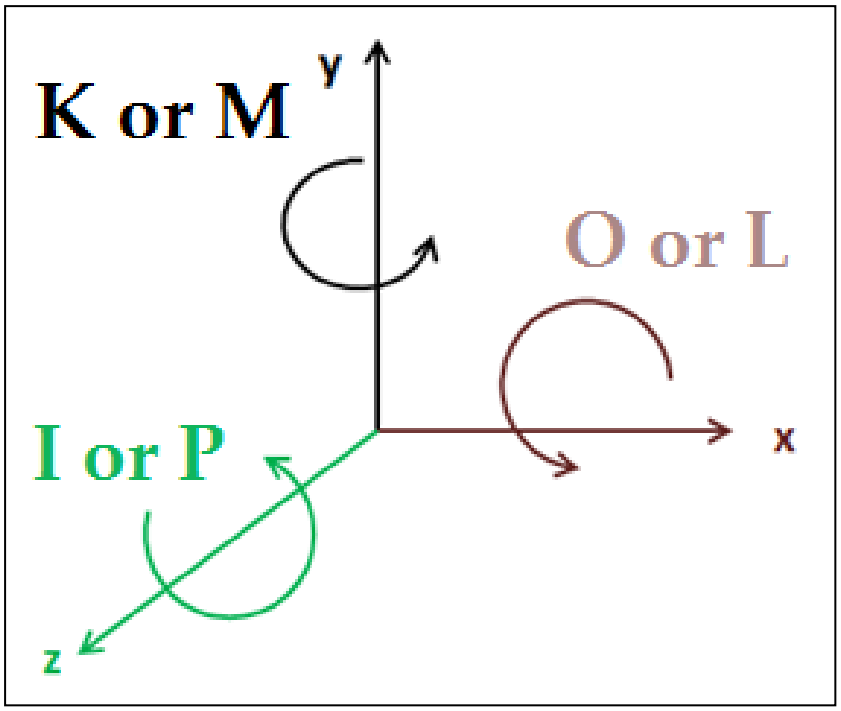
La caméra peut en outre être vue comme un repère, avec comme origine sa position, et une base orthogonale formée par trois vecteurs : le vecteur droite, le vecteur haut, et le vecteur face qui sera ici choisit comme étant le vecteur d'orientation. Le vecteur droite est construit pour être orthogonal au vecteur face, en choisissant astucieusement 2 de ses coordonnées, et en mettant la troisième à 0. Le vecteur haut, enfin, est construit par le produit vectoriel des deux autres vecteurs afin de former une base orthogonale directe de l'espace.

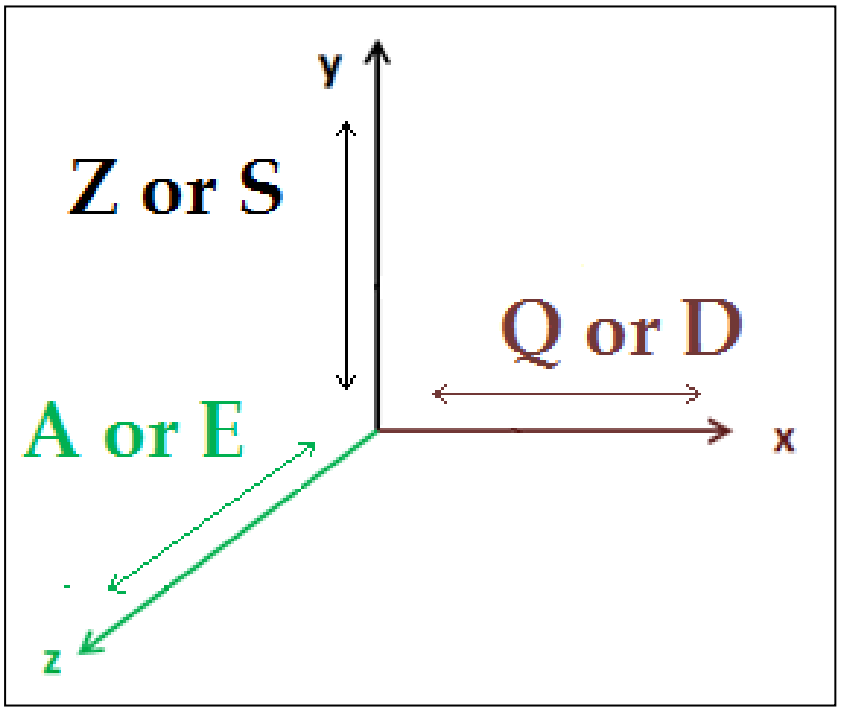
Ainsi, les translations et rotations de la caméra se feront selon ces 3 axes. Les translations assignées au touches du clavier (Z, S, Q, D, A et E) consisteront en un déplacement unitaire prédéfinit selon un des axes de la caméra, et de ce fait ne modifieront que la position dans l'espace de la caméra. La caméra peut également être translaté d'un vecteur quelconque, auquel cas la translation sera décomposée selon les trois axes de la base et les trois translations seront ainsi appelées.

Les rotations assignées au clavier (O, L, K, M, I et P), quant à elles, s'effectuent également selon les trois axes de la caméra, et sont ici gérées par des quaternions. En effet, les rotations autour des axes de la caméra font appel à une fonction générale, permettant la rotation de la caméra autour d'un axe quelconque. Et pour effectuer cette dernière rotation, on instancie un nouveau quaternion et l'on fait appel à la méthode *setFromAxis*, qui permet de rendre ce quaternion en une rotation d'un axe et d'un angle passés en paramètre de cette méthode. Ainsi, en appliquant ce quaternion comme une rotation (multiplication par son inverse à droite, et lui-même à gauche) au quaternion qui modélise l'orientation, nous sommes en mesure d'effectuer une rotation de la vue de la caméra dans l'espace, et dans toutes les directions.

Enfin, pour déterminer la matrice de vue, il suffit de la voir comme une matrice de changement de base du repère classique de la scène au repère de la caméra, pour la construire. La matrice de projection est elle issue directement des démonstrations effectuées en cours. Néanmoins, les fonctions permettant la détermination de ces matrices ne seront pas utilisées dans notre programme, car le Framework fournit dans le Template du

TP1 propose une méthode *lookAt* implémentant déjà cette fonctionnalité, et introduisant moins de temps de calcul que les nôtres.





Animation du Mokujin

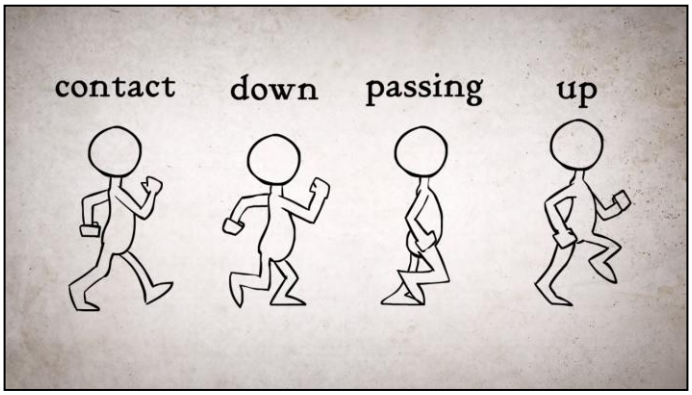
Une fois le modèle 3D chargé, et les textures appliquées, notre personnage 3D est prêt à être animé. Le modèle disposant d'un squelette, l'animation du personnage sera fera en effet par l'application d'isométries affines de l'espace à ces os du squelette. Les transformations responsables de l'animation dépendront aussi du temps, variable d'environnement permettant de faire évoluer les mouvements de Mokujin.

Principe global de l'animation

Pour réaliser l'animation, à chaque rafraichissement de l'affichage, selon une fréquence définie dans le Framework responsable de l'affichage, nous incrémentons une variable de temps d'un pas prédéfinit. En effet, la variable de temps évoluera de 0 à 1 tout au long de l'animation, et sera notamment utilisée comme variable t pour l'interpolation des transformations et positions durant l'animation. L'incrémentation du temps selon un pas définit dépend directement du nombre de frames que l'on veut pour une animation.

Ensuite, cette variable de temps sera introduite comme facteur pour faire évoluer l'angle d'une rotation, ou les composantes d'un vecteur de translation.

Animation 1 : La marche



Dans l'optique d'animer Mokujin avec une marche, nous avons tout d'abord tenté de décomposer les mouvements d'une marche classique. Nous sommes donc arrivés à 2 principales étapes de la marche : le contact d'un pied sur le sol et le passage d'un pied à l'autre.

Les animations consistent alors à bouger les bras opposés aux jambes en avant, à bouger les jambes, les tibias et les pieds de Mokujin, tout ceci se faisant exclusivement par des rotations. D'autre part, une translation selon l'axe y permet de faire avancer le personnage.

Enfin, et pour donner un aspect plus réaliste à la marche, nous avons introduit une translation verticale selon l'axe z, durant la marche, comme cela est représenté sur l'image ci-contre.

Animation 2 : La toupie

Cette animation, pour le moins basique, fut la première mise en place, et consiste en une rotation de tout le corps autour de l'axe z, hormis la tête qui va subir une rotation autour d'un axe lui-même en rotation.

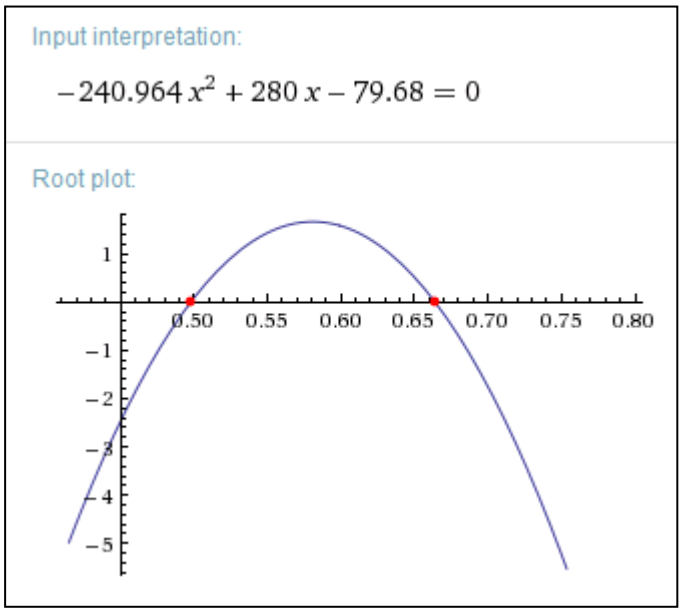
Cela donne un effet de toupie, et la tête, à mi-parcours, se retrouve cachée sous le buste, ce qui ajoute un effet comique à l'animation.

Animation 3 : Flexion – Extension

L'animation de saut vertical peut également être vue comme une animation de flexion – extension, comme on peut en voir en salle de sport. Cette animation a une décomposition en étapes plus complexe que les autres animations, dans la mesure où 6 étapes interviennent dans l'animation : préparation au saut (flexion), phase de poussée des jambes, élévation dans les airs (extension), phase d'atteinte du point culminant, retombée, amortissement de la chute.

C'est ici aussi une succession de rotations qui va animer les membres supérieurs et inférieurs, et une translation qui va permettre d'élever ou rabaisser l'ensemble du buste (et donc du corps).

Pour plus de réalisme, il nous a fallu mettre en place une retombée progressive, notamment en se rapprochant du point culminant. Ceci nous a amené à mettre en place l'équation d'une parabole modélisant la hauteur de Mokujin au cours de cette phase.

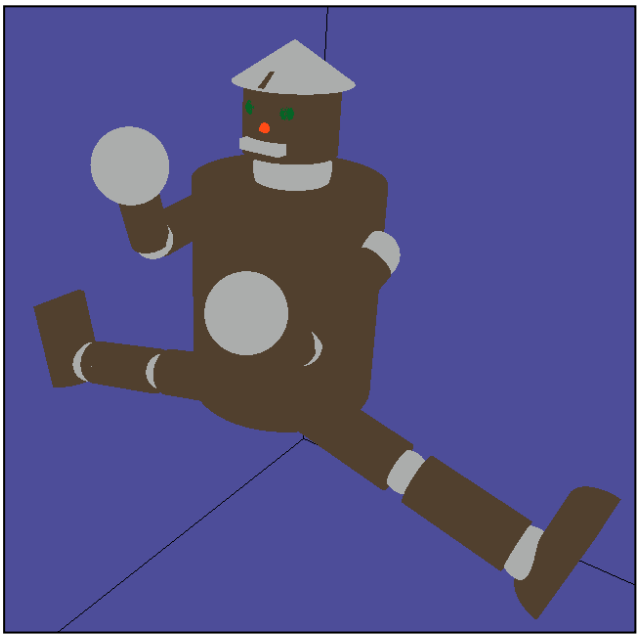


Animation 4 : Grand écart

L'animation du grand fut assez complexe à mettre en place, dans la mesure où les axes des translations et rotations ne sont pas des axes du repère de l'espace.

Ainsi, pour pouvoir réaliser une rotation autour de l'axe que l'on désire, quelque soit l'évolution de l'animation, nous avons choisit de pratiquer un changement de base pour se ramener à l'origine, avant l'application des différentes matrices de rendu et de modèle, pour pouvoir appliquer la rotation, puis se ramener à la base locale de l'objet une fois rendu, et terminer par quelques ajustements avec une translation bien choisie pour maintenir les membres en place autour des articulations.

Néanmoins, il subsiste quelques imperfections au niveau des bras, et l'utilisation ici de key-frames aurait pu aider à déterminer les transformations à effectuer aux bras pour les placer en garde avant classique.



Conclusion

Ce projet nous aura amené à utiliser les connaissances acquises en cours, nous aura amené à les approfondir afin de pouvoir les mettre en avant et les utiliser dans notre métier futur.

Les points les plus problématiques de ce projet auront été, de trouver le format d'export adéquat et réussir à le comprendre et le récupérer correctement afin de pouvoir ensuite y appliquer les transformations.

Afin de rendre ce projet plus vivant, de nombreuses possibilités s'offrent à nous, afficher un font, un sol, ajouter un deuxième personnage et faire des animations plus orientées combats étant donné l'univers dont il est issu.

Ce fut cependant une bonne expérience dans un groupe ou régnais une bonne atmosphère, et chacun à son niveau est fier de son travail accompli.

Bibliographie - Références

- OpenClasrooms.com
- Fr.tuto.com
- Youtube.com
- **Lien Github du projet :** https://github.com/thibauldjardino/in55-animation-mokujin
- http://www.opengl-tutorial.org/fr/beginners-tutorials/tutorial-7-model-loading/
- http://www.assimp.org/index.html
- http://www.wazim.com/Collada_Tutorial_1.htm
- http://ogldev.atspace.co.uk/
- http://www.flipcode.com/documents/charfaq.html#Q2
- http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=23
- http://learnopengl.com/#!Model-Loading/Assimp
- https://sourceforge.net/p/assimp/discussion/817654/thread/f061bdb1/
- https://www.youtube.com/watch?v=2y6aVz0Acx0