

TP Animation de personnage (Character Animation) en C/C++ (M2)

TP 1ere partie : affichage

Affichage récursif (comme en M1n juste pour comprendre)

Ecrivez une fonction récursive qui affiche le squelette (une sphère pour chaque articulation et un cylindre pour chaque membre) d'une créature animée par des données issues de capture de mouvement. Le fichier de l'animation est stocké au format BVH. Votre fonction doit parcourir l'arbre décrivant le squelette récursivement et utilise les fonctions `drawSphere` et `drawCylinder` pour l'affichage.

```
void CharAnimViewer::bvhDrawGL(const BVH& , int frameNumber);  
void CharAnimViewer::bvhDrawGLRec(const BVHJoint& , int frameNumber,  
Transform& T); // la fonction récursive sur le squelette
```

Remarque : on ne s'occupe pas du temps pour l'instant mais uniquement du numéro de la posture.
Remarque 2 : vous pouvez trouver des BVH dans le répertoire data du code de départ. Notamment le fichier `robot.bvh` pour debuguer.

Affichage itératif avec une classe Skeleton

Vous allez créer un module `Skeleton.h/.cpp` (ce code n'est qu'indicatif, vous êtes libre de vos structures de données). Cette classe va stocker un tableau de toutes les articulations (`SkeletonJoint`) du squelette et pour chaque articulation stocke l'identifiant de l'articulation parent et la matrice de passage de l'articulation vers le monde.

Le fichier est déjà présent dans le code départ avec des *TODO à compléter* :

```
class Skeleton {  
public:  
    struct SkeletonJoint  
    {  
        int m_parentId; Le numéro du père dans le tableau de CAJoint de CASkeleton  
        Transform m_l2w; // La matrice du repère de l'articulation vers le monde  
    };  
    Skeleton() {}  
    //! Créer un squelette ayant la même structure que définit dans le BVH c'est à dire  
    //! créer le tableau de SkeletonJoint à la bonne taille, avec les parentId initialisé pour chaque case  
    void init(const BVH& bvh);  
    //! Renvoie la position de l'articulation i en multipliant le m_l2w par le Point(0,0,0)  
    Vector getJointPosition(int i) const;  
    //! Renvoie l'identifiant de l'articulation père de l'articulation numéro i  
    int getParentId(const int i) const;  
  
    //! Renvoie le nombre d'articulation  
    int numberOfJoint() const;  
    //! Positionne ce squelette dans la position n du BVH.  
    //! Assez proche de la fonction récursive (question 1), mais range la matrice (Transform)
```

```
    //! dans la case du tableau. Pour obtenir la matrice allant de l'articulation
    local vers le monde,
    //! il faut multiplier la matrice allant de l'articulation vers son père à
    la matrice du père allant de
    //! l'articulation du père vers le monde.
    void setPose(const BVH& bvh, int frameNumber);
    //! Positionne ce squelette entre la position frameNbSrc du BVH Src et la
    position frameNbDst du bvh Dst
    //void setPoseInterpolation(const BVH& bvhSrc, int frameNbSrc, const BVH&
    bvhDst, int frameNbDst, float t);
    //! Positionne ce squelette entre la position frameNbSrc du BVH Src et la
    position frameNbDst du bvh Dst
    //! idem à setPoseInterpolation mais interpole avec des quaternions sur
    chaque articulations
    //void setPoseInterpolationQ(const BVH& bvhSrc, int frameNbSrc, const BVH&
    bvhDst, int frameNbDst, float t);

    //! Calcule la distance entre deux poses
    //! precondition: les deux squelettes doivent avoir le
    //! même nombre d'articulations (même structure d'arbre)
    //! ==> Sera utile lors de la construction du graphe d'animation
    // friend float distance(const CASkeleton& a, const CASkeleton& b);
```

protected:

```
    //! L'ensemble des articulations.
    //! Remarque : la notion de hiérarchie (arbre) n'est plus nécessaire ici,
    //! pour tracer les os on utilise l'information "parentID" de la class
    CAJoint
    std::vector<SkeletonJoint> m_joint;
};
```

Dans le Viewer vous devez écrire une fonction qui fait l'affichage

```
void CharAnimViewer::skeletonDraw(const Skeleton& ske);
```

Remarque : on sépare bien l'affichage de la gestion du squelette pour pouvoir réutiliser le code Skeleton avec une autre librairie d'affichage.

Transition avec interpolation sur les angles

Ecrivez une fonction qui réalise la transition entre deux postures de deux animations, elle est en commentaire plus haut dans la classe Skeleton.

```
void Skeleton::setPoseInterpolation( ...
```

Entre la fin de l'animation 1 (ayant par exemple 100 postures) et le début de l'animation 2, on appellera cette fonction

```
void Skeleton::setPoseInterpolation(bvh1, 99, bvh2, 0, interpolationValue);
```

en faisant varier *interpolationValue* entre 0 et 1. Par exemple, une valeur de 0.3 va indiquer une interpolation de 70% de la posture source et de 30% de la posture destination. Pour cela vous pouvez interpoler directement les valeurs d'angles données dans chaque Channel. Faites un copier/coller de la fonction setPose et interpoler chaque valeur d'angle.

Que remarque-t-on ? Sur l'image les squelettes sont des poses de l'animation danse.bvh. A gauche la frame 133, à droite la frame 133+40 et au milieu l'interpolation entre les deux. En bas l'interpolation des angles (avec le problème) et en haut l'interpolation avec les quaternions.



Transition avec interpolation basée sur les quaternions (en option) ou plutôt revenez y plus tard

L'interpolation des angles conduit pour certaines configurations à des valeurs erronées : $\text{interpolation}(-20^\circ, 10^\circ) = -5^\circ$ alors que $\text{interpolation}(340^\circ, 10^\circ) = 175^\circ$ pourtant il s'agit des mêmes angles modulo 360° . Sur la sphère en 3D dimension les cas identiques sont plus compliqués à résoudre qu'un simple 'if'. Pour résoudre ce problème de l'interpolation sphérique des rotations de matrice, vous devez utiliser les quaternions, regardez le module Quaternion.h.cpp (très facile à utiliser). [A lire sur les quaternions](#). Ecrivez la fonction suivante qui est en commentaire dans Skeleton plus haut. Cette fonction fait l'interpolation à l'aide de 2 quaternions (slerp) :

```
void Skeleton::setPoseInterpolation_Q(const BVH& bvhSRC, int frameNumberSRC,
const BVH& bvhDST, int frameNumberDST, const float interpolationValue);
```

Vous devez écrire d'abord une classe TransformQ qui comportera un quaternion pour la rotation et un vecteur pour la translation. Cette classe se comportera comme une matrice 4×4 (Class Transform de gkit2light). Le fichier est déjà présent dans le code départ avec des zones TODO à compléter :

```
class TransformQ
{
    TransformQ();
    TransformQ( const Quaternion& q, const Vector& t);
    friend TransformQ operator*(const TransformQ& a, const TransformQ& b);
    friend TransformQ slerp(const TransformQ& a, const TransformQ& b, float
t);
    friend Vector operator*(const TransformQ& a, const Vector& b);
    friend Point operator*(const TransformQ& a, const Point& b);
    ...
protected:
    Quaternion q;
    Vector t;
};
```

TP 2e partie : Contrôleur d'animation

Déplacer une sphère au clavier

Ecrivez une class CharacterController qui à partir des touches claviers contrôlera le déplacement d'un personnage. Dans un 1er temps faites juste déplacer une boule : accélérer, freiner, tourner à droite, tourner à gauche, sauter. Ce contrôleur comportera une position et une vitesse. La vitesse sera modifiée par les flèches (ou un pad) et la position sera mise à jour dans la fonction update du Viewer en utilisant le paramètre elapsedTime à boucle d'affichage. Une classe de Controller peut ressembler à ceci.

```
class CharacterController
{
public:
    CharacterController()

    void update(const float dt);

    void turn(const Transform& transf_v);
    void turnXZ(const float& rot_angle_v);
    void accelerate(const float& speed_inc);
    void setVelocityMax(const float vmax);

    void setPosition(const Vector& p);
    void setVelocityNorm(const float v);

    const Vector& position() const;
    const Vector& direction() const;
    float velocity() const;

protected:
    Transform m_ch2w;    // matrice du character vers le monde
                        // le personnage se déplace vers X
                        // il tourne autour de Y
                        // Z est sa direction droite

    float m_v;           // le vecteur vitesse est m_v * m_ch2w *
Vector(1,0,0)           // ne peut pas accélérer plus que m_vMax

    float m_vMax;

};
```

Déplacer un personnage au clavier

Dans un 2e temps, votre contrôleur comportera également une série d'animation bvh : par exemple marcher, courir, sauter. En fonction de l'action que veut faire le joueur appuyant sur des touches vous changerez d'animation. Ce changement se fera dans un 1er temps brutalement. Dans un 2e temps, vous utiliserez la fonction de transition d'animation écrite précédemment. Ne vous occupez pas non plus des pieds qui glissent sur le sol. Un meilleur contrôle se fera avec le graphe d'animation dans la 3e partie.

TP 3e partie : graphe d'animation

- [Motion Graph de l'article original](#);
- Des BVH avec squelette compatible pour le graphe sont donnés dans le git, répertoire Second_Life.

Nous avons remarqué dans la partie 1 que la transition d'animation ne fonctionne bien que lorsque les deux poses du squelette sont assez proches (il faut bien sûr également que les deux squelettes aient la même topologie). L'idée d'un graphe d'animation est de construire un graphe où chaque noeud correspond à une pose d'une animation et où chaque arrête définit qu'une transition est possible entre les deux poses.

Comparaison de deux poses d'animation

Pour construire un graphe d'animation à partir d'une ou plusieurs animations, on doit être capable de comparer deux poses d'animation. Une distance de 0 indique que les deux poses sont identiques. Une distance grande indique que les 2 poses sont très différentes. A partir de la classe Skeleton, écrivez la fonction de calcul de distance entre deux poses de squelette. Cette fonction est déjà présente dans la classe Skeleton plus haut mais en commentaire. Cette fonction calcule itérativement sur toutes les articulations la somme des distances euclidiennes entre chaque articulation de deux squelettes ayant la même topologie mais dans des poses différentes.

```
friend float Skeleton::Distance(const Skeleton& a, const Skeleton& b);
```

Remarque : il est important de ne pas tenir compte de la translation et de la rotation de l'articulation racine. Une même pose a deux endroits du monde doit donner une distance de 0. Dans un 1er temps, votre personnage aura son noeud root centré en (0,0,0), puis dans la dernière partie de cette question, vous traiterez le centre de gravité.

Construction du graphe

Ecrivez un module MotionGraph qui contiendra un ensemble de BVH et le graphe d'animation définissant des transitions dans cet ensemble d'animation.

- Un noeud du graphe = (Identifiant d'une animation + un numéro de pose);
- un arc du graphe entre deux poses indique la transition possible entre ces deux poses. Deux poses sont compatibles à la transition quand la distance entre les deux squelettes est inférieure à un certain seuil fixé empiriquement.

Vous pouvez créer un module CACore/CAMotionGraph.h/.cpp

```
class MotionGraph
{
    ...
protected:
    //! L'ensemble des BVH du graphe d'animation
    std::vector<BVH> m_BVH;

    //! Un noeud du graphe d'animation est repéré par un entier = un identifiant
    typedef int GrapheNodeID;

    //! Une animation BVH est repérée par un identifiant = un entier
    typedef int BVH_ID;

    //! Un noeud du graphe contient l'identifiant de l'animation, le numéro
    //! de la frame et les identifiants des noeuds successeurs
    //! Remarque : du code plus "joli" aurait créé une classe CAGrapheNode
    struct GrapheNode
    {
        BVH_ID id_bvh;
        int frame;
        std::vector<GrapheNodeID> ids_next;    //! Liste des nœuds successeurs
    };

    //! Tous les noeuds du graphe d'animation
    std::vector<GrapheNode> m_GrapheNode;
};
```

Navigation dans le graphe

Une fois ce graphe construit, on peut définir différente manière de naviguer dedans :

1. Un parcours aléatoire dans le graphe (juste pour vérifier que le graphe est ok);
2. L'utilisateur donne des directions au clavier \Rightarrow le parcours dans le graphe est conditionné par ces contraintes.

Gestion correcte du centre de gravité

Pour chaque arc du graphe, vous devez stocker la transformation (soit une matrice 4×4 , soit un quaternion et une translation) du noeud root (souvent le centre de gravité) entre la pose i et la pose $i+1$. Cette transformation sera appliqué au noeud root de votre personnage quand il empruntera l'arc.

TP ancienne version : Cinématique inverse

La problématique de la cinématique inverse est de retrouver la position d'un squelette (donc les angles des articulations) en donnant pour un ou plusieurs os des positions cibles. Par exemple, on donne une position cible pour la main droite d'un squelette humanoïde, la cinématique inverse donne les différents mouvements (variations d'angles) de tout le corps pour atteindre ce but.

De nombreux algorithmes existent. Dans ce TP, vous pouvez bouger une cible dans le code donné. Vous coderez l'approche [FABRIK](#) (voir cours). C'est une approche à la fois simple à coder et qui donne de bons résultats avec peu d'itération.

Dans un premier temps, vous partirez d'un squelette sans jonction (une chaîne d'articulation en ligne façon serpent). La racine du squelette est fixe et l'extrémité du squelette devant attendre la cible. Par la suite, vous pouvez prévoir des squelettes avec jonction. On peut imaginer produire une animation de marche en donnant la courbe des pieds et du bassin, les autres articulations étant déduite avec l'algorithme de cinématique inverse.

TP ancienne version : couplage MoCap et animation physique

L'idée est de jouer une animation de personnage issue de mocap puis de basculer en animation physique. Par exemple, un personnage marche puis trébuche sur un obstacle et tombe. La partie marche est de la mocap, la chute de la physique. Quand on passe à la physique, chaque membre du corps devient alors un solide rigide (rigid body) dont le mouvement est dirigé par un calcul physique (gravité, collision, liaison avec le père). Pour les calculs physiques, nous allons utiliser la librairie [BulletPhysics](#).

- [La doc de BulletPhysics](#)

Lancez votre application et appuyer sur 'P' pour passer en mode 'physics' ('enter' pour activer le mode animation du viewer). Pour l'instant, vous avez un bras et un avant bras tombant sur le sol. Regardez le code `src/CACore/CARAgdoll.h` qui créer les 2 solides du bras.

A vous de modéliser en physique le squelette issue de la mocap et de donner aux solides physiques l'impulsion initiale correspondant au mouvement de mocap.

TP ancien : édition multi-résolution d'animation

L'idée est d'éditer à plusieurs résolutions une animation (comme un égalizer de son) : amplifier les grandes amplitudes pour exagérer un mouvement sans toucher aux petites amplitudes pour ne pas ajouter des tremblements (ou l'inverse). Tout est très bien expliqué dans l'article suivant section 2, en particulier 2.1 :

- L'article au format PDF : [Motion Signal Processing](#)
 1. Pour chaque rotation de chaque articulation, construire la représentation multi-résolution. Une animation d'une articulation se reconstruit en partant de la valeur moyenne G_n + la somme de toutes les variations aux différentes résolutions.
 2. Demander à l'utilisateur les n ($n=8$ pour une animation à 256 valeurs) coefficients, multiplier chaque variation par ce coefficient puis reconstruire le signal.

Divers articles

- [Web qui répertorie des articles et idées autour de "Game et Animation"](#),
- [Divers articles que vous pouvez lire](#)