Cinématique inverse

R. Kulpa

Le but de ce TP est de vous faire mettre en oeuvre un algorithme simplifié (mais rapide) de cinématique inverse sur une chaine polyarticulée. Allez chercher le fichier *IK_ etudiants.zip* à l'adresse suivante :

http ://m2slab.com/ip/IK_etudiants.zip.

Placez-vous dans votre répertoire de TP et décompressez cette archive. Le répertoire IK, ainsi extrait, contient les sous-répertoires et fichiers suivants :

- main.cc : le programme principal qui déclare une fenêtre GLUT et permet d'afficher le résultat de votre animation.
- Animation.h : une classe vous permettant d'implémenter votre animation.
- SceneGraph : un répertoire contenant un ensemble de classes gérant un graphe de scène. Cela vous facilitera la description de la chaîne articulaire.
- Math : un répertoire contenant des classes de vecteurs, de quaternions et une classe spécifique permettant de gérer des chaînes articulaires et la dérivation de celle-ci par rapport à un angle.

1 Description du package fourni

1.1 Visualisation

Le programme principal se charge de l'affichage de vos objets graphiques. Tout navigation dans la visualisation 3D est volontairement bloquée. Plus tard dans le TP, le bouton gauche de la souris permettra de spécifier la cible désirée pour la cinématique inverse. La caméra est orientée de la manière suivante : elle regarde suivant l'axe Z avec l'axe Y dirigé vers le haut et l'axe des X dirigé vers la gauche.

1.2 Graphe de scène

Les objets du graphe de scène sont décrits dans un namespace nommé SceneGraph et se trouvent dans le répertoire SceneGraph. Voici la hiérarchie des classes du graphe de scène ainsi que les méthodes principales :

- class DrawObject : une classe abstraite dont hérite tous les noeuds du graphe de scène. Elle propose une méthode draw() qui permettra à tous les noeuds du graphe de s'afficher en utilisant les primitives OpenGL.
  - class Group : une classe permettant de gérer des groupes d'objets. Elle dispose d'une méthode addChild(DrawObject*) permettant d'ajouter des fils.
    - class Translation : une classe permettant de gérer des translations. Elle dispose d'une méthode setVector(float x, float y, float z) permettant de changer le vecteur de translation.
    - class Rotation : une classe permettant de gérer des rotations. Elle dispose d'une méthode setAngle(float angle) permettant de changer l'angle de rotation ainsi que d'une méthode setVector(float x, float y, float z) permettant de changer l'axe de rotation.
  - class Sphere : une classe permettant d'afficher une sphère. Son constructeur est le suivant : Sphere(float rayon).
  - class Cylinder : une classe permettant d'afficher un cylindre. Son constructeur est le suivant : Cylinder(float rayonBase, float rayonSommet, float hauteur). Il est à noter que la hauteur est alignée sur l'axe Y lors de la création de la géométrie.
  - class Color : une classe permettant de changer la couleur des objets.

1.3 Les outils mathématiques

Les outils mathématiques sont décrits dans un namespace nommé Math et se trouvent dans le répertoire nommé Math. Voici une description de la hiérarchie des classes fournies :

- class Object : la classe de base des objets mathématiques.
  - class Vector3 : la classe de gestion des vecteurs en 3D. Elle implémente tous les opérateurs (+,-...) et utilise l'opérateur * pour le produit scalaire.
  - class Quaternion : la classe de gestion de quaternions. Elle implémente tous les operateurs et propose une méthode rotate(Quaternion q) permettant de calculer les rotations.
- class Chain : une classe permettant de décrire une chaîne articulaire simple (sans fourche). Cette classe propose une méthode add(Object*) permettant d'ajouter une transformation (Translation, Rotation) à la suite des transformations déjà ajoutées. A partir de la description de l'enchaînement de ces transformations, la méthode Vector3 compute() vous permet de récupérer la position de l'extrémité de la chaine articulaire. Pour vous permettre d'effectuer des calculs de cinématique inverse, elle dispose d'une méthode Vector3 derivate(float epsilon, Quaternion * q) vous permettant de calculer une estimation de la dérivée (à epsilon près) de la position finale de la chaine en fonction de l'angle associé au quaternion q.

1.4 La classe Animation

La classe Animation, contenue dans le fichier Animation.h va vous permettre de décrire la géométrie d'un objet polyarticulé ainsi que de l'animer. Cette classe, dont une instance est créée dans le programme principal (fichier main.cc), comporte deux méthodes importantes :

- void computeAnimation() : cette méthode est appelée avant la phase de rendu pour vous permettre de calculer l'animation associée à la scène. Vous devrez implémenter cette méthode pour gérer vos animations.
- void drawScene() : cette méthode est appelée durant la phase de rendu pour vous permettre d'afficher la scène. Elle est déjà implémentée et affiche le noeud m_root (racine du graphe de scène).

Note : dans le répertoire doc/html, vous trouverez le fichier index.html correspondant à la documentation doxygen des diverses classes sus-citées. N'hésitez pas à la consulter.

2 Questions

Question 1 : Dans la classe animation, créez une méthode permettant d'ajouter une chaine polyarticulée au graphe de scène. La chaine polyarticulée sera représentée par une suite de cylindres de 0.05 de rayon et 1 de hauteur reliés par des articulations symbolisées par des sphères de rayon 0.08. Chaque articulation comportera une rotation suivant l'axe des Z (pour passer en 3D, il suffirait de rajouter des rotations suivant les autres axes). Cette méthode devra tenir compte de la donnée membre *nbSegments* qui définit le nombre de segments constituant la chaîne. Une fois cette méthode implémentée, testez-la en l'appelant dans le constructeur de la classe Animation.

Question 2 : Nous allons maintenant préparer la structure de données nous permettant de gérer l'animation. A cette fin, nous allons modifier la méthode précédente. Dans un premier temps, vous allez compléter, parallèlement à la création du graphe de scène, une instance de la classe Chain (déclarée comme attribut) contenant toutes les transformations associées à la chaine polyarticulée (noeud Translation <=> Vector3, noeud Rotation <=> Quaternion). Dans le même temps, vous allez ajouter deux attributs de type std : :vector<Quaternion*> et std : :vector<Vector3*>. Parallèlement à la construction, vous allez stocker dans ces deux tables des pointeurs sur les instances de Vector3 dans le graphe de scène et le Quaternion correspondant dans la chaine articulaire (classe Chain).

Question 3 : Afin de tester la chaîne cinématique, créez une sphère de rayon 0.12 et placez-là à l'extrémité de la chaîne (méthode compute()).

Question 4 : Dans le cours, retrouvez l'équation qu'il va falloir implémenter pour faire la cinématique inverse par CCD. Cette équation permet de calculer $d\theta_i$ en fonction de dP. Dès lors, que pouvez-vous déjà calculer dans cette équation et qu'est-ce qui sera plus difficile à calculer ?

Question 5 : L'expression de la question précédente vous permet de calculer la variation d'un angle de la chaine articulée en fonction de dP. Pour mettre en place la cinématique inverse par CCD, nous allons utiliser cette expression pour calculer itérativement chacun des angles. Pour ce faire, nous allons utiliser l'algorithme suivant :

1. Calcul de $d\theta_1$ en fonction de dP et $f(\theta_1, ..., \theta_n)$ puis calcul de la nouvelle valeur de θ_1 à partir de $d\theta_1$.
2. Calcul de $d\theta_2$ en fonction de dP et $f(\theta_1, ..., \theta_n)$ puis calcul de la nouvelle valeur de θ_2 à partir de $d\theta_2$.
3. ...

Ecrivez cet algorithme dans une nouvelle méthode computeOneIteration() de la classe Animation. Elle utilisera la méthode *derivate(float epsilon, Quaternion * q)* de l'instance de la classe *Chain* pour calculer les $d\theta_i$. On prendra par exemple une valeur de 0.001 pour *epsilon*. Cette méthode prendra en entrée la cible *target* à atteindre. Cette cible est fournie par la méthode computeAnimation() (qui doit appeler la méthode computeOneIteration()). Ensuite, on utilisera $dP = cible - chain.compute()$ pour le calcul de la cinématique inverse. La méthode devra, pour chaque itération, mettre à jour l'angle associé aux quaternions, ainsi que l'angle associé aux rotations dans le graphe de scène (utilisation des deux tables déclarées dans la question 2). Testez cette méthode en l'appelant directement dans la méthode computeAnimation(). Expliquez pourquoi l'adaptation n'est pas instantanée.

Question 6 : Afin d'atteindre instantanément la cible et ne pas être dépendant du nombre d'itérations nécessaires, utilisez une boucle qui calcule de nouvelles itérations jusqu'à ce que la cible soit atteinte (que la distance entre la position de l'extrémité de la chaîne polyarticulée et la cible soit inférieure à un certain seuil). Testez votre algorithme avec des cibles atteignables puis avec des cibles lointaines. Expliquez pourquoi l'algorithme ne converge pas dans ce dernier cas.

Question 7 : Afin de ne pas continuer à faire le calcul de la cinématique inverse alors qu'aucune nouvelle cible n'est spécifiée, votre méthode computeAnimation ne doit faire le calcul que lorsqu'une nouvelle cible est choisie, c'est-à-dire lorsque le booléen *newTarget* est vrai. Testez votre méthode.

Question 8 : Proposez une solution permettant à votre cinématique inverse de ne pas se bloquer lorsque les cibles sont inatteignables. Implémentez-la.

Question 9 : Proposez une méthode permettant de gérer les butées articulaires et implémentez-la.

Question 10 : Les adaptations obtenues ne sont pas réparties de manière homogène le long de la chaîne cinématique. Proposez une solution pour régler ce problème (notion d'amortissement).

Question 11 : Pour terminer, afin d'obtenir une animation, on va modifier la cible fournie à la cinématique inverse. En effet, à chaque fois que l'on aura une nouvelle cible (paramètre *newTarget* de *computeAnimation()*), on va stocker la position de départ (actuelle) et celle d'arrivée (la cible). Et tant qu'on n'a pas de nouvelle cible, on va avancer linéairement vers cette cible. Implémentez-le et testez-le.

Question 12 : Utiliser maintenant une spline de type Hermine pour contrôler l'extrémité (utiliser les classes Vector4 et Matrix44). Dans un premier temps, on utilisera des dérivées constantes suivant l'axe X. Ensuite, on utilisera la dérivée d'arrivée de l'effecteur sur la dernière cible comme vecteur de départ de la spline. Le vecteur d'arrivée est à votre convenance.