

Cinématique Inverse : FABRIK



L'objectif de cette partie est de réaliser une cinématique inverse personnalisée (Inverse Kinematic IK) d'un personnage sous Unity. L'algorithme de cinématique inverse que nous allons coder se base sur FABRIK.

Il existe déjà un algorithme de cinématique inverse dans Unity. Il ne fonctionne qu'avec un squelette de type humanoïde. Dès que votre personnage est différent, il faut réécrire un algorithme d'IK. De plus, un des objectifs de ce TP serait de comparer leur algorithme avec le vôtre. Il existe des implémentations de FABRIK dans le store, jouez le jeu de recoder le vôtre. FABRIK n'est pas un algorithme compliqué.

Il est envisageable d'aller jusqu'à utiliser notre algorithme d'IK pour faire marcher notre personnage en définissant des trajectoires de pieds, ce qui est une approche complètement différente car procédurale de l'approche classique proposer dans la partie MECANIM de ce TP.

1. La cible

- Créez un objet simple que nous allons piloter au clavier et qui sera la cible d'une extrémité (une main ou un pied). Menu "GameObject" puis "3D Object" puis "Sphere"
- Pour que l'objet fasse quelque chose, il faut écrire du code pour lui dire quoi faire. Dans le panneau *Project*, sélectionnez *Create* en haut (ou cliquez avec le bouton droit de la souris n'importe où dans le panneau) et sélectionnez C# script. Nommez-le *IKTargetMovement*.
- Double-cliquez sur votre script pour l'ouvrir (Visual Studio se lance). Chaque fois que vous créez un nouveau script, Unity ajoute deux méthodes par défaut, *Start()* et *Update()*. 'Start' est appelé une fois, lorsque l'objet entre dans le monde du jeu. La mise à jour 'Update' est appelée à chaque image/rendu.
- Copiez et collez la ligne suivante dans la méthode de mise à jour (*Update*).

```
if (Input.GetKey(KeyCode.LeftArrow))  
    transform.position += Vector3.left;
```

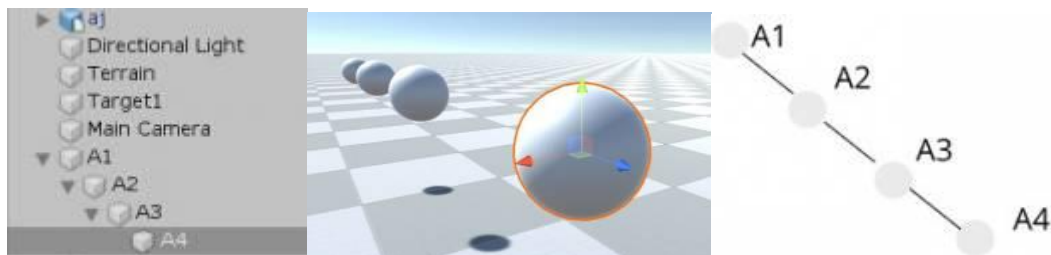
- Sélectionnez la sphère en cliquant dessus. A droite dans le panel "Inspector", Ajoutez un composant "Add component", puis trouvez votre script de mouvement dans la petite barre de recherche (ou section "script") et ajoutez-le à votre capsule.
- Appuyez sur le bouton de lecture en haut de l'éditeur.
- Completez le script avec les 4 (voir 6) directions

2. Un squelette simple (1 unique chaine cinématique)

Un squelette est un arbre hiérarchique. Unity représente le monde sous forme d'un arbre (le graphe de scène).

1. Créez un 1ère sphère de nom A1 qui sera l'“Epaule” du squelette ;
2. Puis créez A2 le Coude, A3 le Poignet et A4 le Doigt.

Votre hiérarchie de scène doit ressembler à ceci.



En sélectionnant la racine de votre objet (A1), créez maintenant un script 'IK.cs' avec le bouton “Add Component”, tapez IK. Comme aucun script de ce nom n'existe pas, Unity vous propose d'en créer un. Vous pouvez partir de ce script à compléter.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IK : MonoBehaviour
{
    // Le transform (noeud) racine de l'arbre,
    // le constructeur créera une sphère sur ce point pour en garder une copie visuelle.
    public GameObject rootNode = null;

    // Un transform (noeud) (probablement une feuille) qui devra arriver sur targetNode
    public Transform srcNode = null;

    // Le transform (noeud) cible pour srcNode
    public Transform targetNode = null;

    // Si vrai, recréer toutes les chaines dans Update
    public bool createChains = true;

    // Toutes les chaines cinématiques
    public List<IKChain> chains = new List<IKChain>();

    // Nombre d'itération de l'algo à chaque appel
    public int nb_ite = 10;

    void Start()
    {
        if (createChains)
        {
            Debug.Log("(Re)Create CHAIN");
            createChains = false;    // la chaîne est créée une seule fois, au début

            // TODO :
            // Création des chaines : une chaîne cinématique est un chemin entre deux nœuds carrefours.
            // Dans la 1ere question, une unique chaine sera suffisante entre srcNode et rootNode.
        }
    }
}
```

```

        // TODO-2 : Pour parcourir tous les transform d'un arbre d'Unity vous pouvez faire une
fonction récursive
        // ou utiliser GetComponentInChildren comme ceci :
        // foreach (Transform tr in gameObject.GetComponentInChildren<Transform>())

        // TODO-2 : Dans le cas où il y a plusieurs chaines, fusionne les IKJoint entre chaque
articulation.
    }
}

void Update()
{
    if (createChains)
        Start();

    if (Input.GetKeyDown(KeyCode.I))
    {
        IKOneStep(true);
    }

    if (Input.GetKeyDown(KeyCode.C))
    {
        Debug.Log("Chains count="+chains.Count);
        foreach (IKChain ch in chains)
            ch.Check();
    }
}

void IKOneStep(bool down)
{
    int j;

    for (j = 0; j < nb_ite; ++j)
    {
        // TODO : IK Backward (remontée), appeler la fonction Backward de IKChain
        // sur toutes les chaines cinématiques.

        // TODO : appliquer les positions des IKJoint aux transform en appelant ToTransform de IKChain

        // IK Forward (descente), appeler la fonction Forward de IKChain
        // sur toutes les chaines cinématiques.

        // TODO : appliquer les positions des IKJoint aux transform en appelant ToTransform de IKChain
    }
}
}

```

La classe IKChain gère une unique chaine cinématique. Les articulations seront stockées dans la liste 'joints'.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IKChain
{
    // Quand la chaine comporte une cible pour la racine.
    // Ce sera le cas que pour la chaine comportant le root de l'arbre.
    private IKJoint rootTarget = null;

    // Quand la chaine à une cible à atteindre,
    // ce ne sera pas forcément le cas pour toutes les chaines.
    private IKJoint endTarget = null;

    // Toutes articulations (IKJoint) triées de la racine vers la feuille. N articulations.
    private List<IKJoint> joints = new List<IKJoint>();
}

```

```
// Contraintes pour chaque articulation : la longueur (à pour
// ajouter des contraintes sur les angles). N-1 contraintes.
private List<float> constraints = new List<float>();

// Un cylindre entre chaque articulation (Joint). N-1 cylindres.
//private List<GameObject> cylinders = new List<GameObject>();

// Créer la chaine d'IK en partant du noeud endNode et en remontant jusqu'au noeud plus haut, ou
// jusqu'à la racine
public IKChain(Transform _rootNode, Transform _endNode, Transform _rootTarget, Transform _endTarget)
{
    Debug.Log("=== IKChain::createChain: ===");
    // TODO : construire la chaine allant de _endNode vers _rootTarget en remontant dans l'arbre (ou
    // l'inverse en descente).
    // Chaque Transform dans Unity a accès à son parent 'tr.parent'
}

public void Merge(IKJoint j)
{
    // TODO-2 : fusionne les noeuds carrefour quand il y a plusieurs chaines cinématiques
    // Dans le cas d'une unique chaine, ne rien faire pour l'instant.
}

public IKJoint First()
{
    return joints[0];
}
public IKJoint Last()
{
    return joints[ joints.Count-1 ];
}

public void Backward()
{
    // TODO : une passe remontée de FABRIK. Placer le noeud N-1 sur la cible,
    // puis on remonte du noeud N-2 au noeud 0 de la liste
    // en résolvant les contrainte avec la fonction Solve de IKJoint.
}

public void Forward()
{
    // TODO : une passe descendante de FABRIK. Placer le noeud 0 sur son origine puis on descend.
    // Codez et déboguez déjà Backward avant d'écrire celle-ci.
}

public void ToTransform()
{
    // TODO : pour tous les noeuds de la liste appliquer la position au transform : voir ToTransform
    // de IKJoint
}

public void Check()
{
    // TODO : des Debug.Log pour afficher le contenu de la chaine (ne sert que pour le debug)
}
}
```

La classe IKJoint stocke un lien vers les Transform de l'arbre de scène Unity, ainsi que la position du noeud durant l'algorithme de FABRIK. Cette position sera appliquée au Transform par la fonction 'ToTransform'.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Assertions;

public class IKJoint
{
    // la position modifiée par l'algo : en fait la somme des positions des sous-branches.
    // _weight comptera le nombre de sous-branches ayant touchées cette articulation.
    private Vector3 _position;

    // un lien vers le Transform de l'arbre d'Unity
    private Transform _transform;

    // un poids qui indique combien de fois ce point a été bougé par l'algo.
    private float _weight = 0.0f;

    public string name
    {
        get
        {
            return transform.name;
        }
    }

    public Vector3 position // la position moyenne
    {
        get
        {
            if (_weight == 0.0f) return _position;
            else return _position / _weight;
        }
    }

    public Vector3 positionTransform
    {
        get
        {
            return _transform.position;
        }
    }

    public Transform transform
    {
        get
        {
            return _transform;
        }
    }

    public Vector3 positionOrigParent
    {
        get
        {
            return _transform.parent.position;
        }
    }

    public IKJoint(Transform t)
    {
        // TODO : initialise _position, _weight
    }

    public void SetPosition(Vector3 p)
    {
        // TODO
    }

    public void AddPosition(Vector3 p)
    {
        // TODO : ajoute une position à 'position' et incrémente '_weight'
    }
}
```

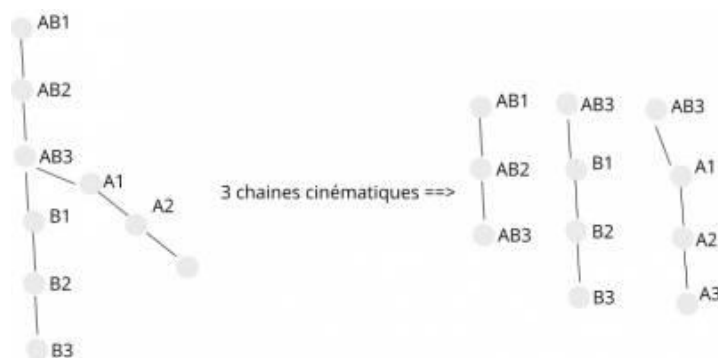
```
}

public void ToTransform()
{
    // TODO : applique la _position moyenne au transform, et remet le poids à 0
}

public void Solve(IKJoint anchor, float l)
{
    // TODO : ajoute une position (avec AddPosition) qui repositionne _position à la distance l
    // en restant sur l'axe entre la position et la position de anchor
}
}
```

3. Un squelette à plusieurs branches (plusieurs chaines cinématiques)

1. Commencez par créer un squelette avec plusieurs chaines cinématique comme indiqué sur la figure ci-dessous;
2. Modifiez le code précédent pour gérer plusieurs branches dans un arbre.



4. Gestion des angles

Pour l'instant nous avons simplement déplacé les positions des articulations dans la fonction 'Solve'. Pour bien faire, il faudrait également gérer l'orientation de l'articulation.

1. Regardez les fonctions [Vector3.RotateToward](#) et [Quaternion.LookRotation](#). Elles vont vous permettre de tourner le repère dans la direction du fils. Ceci est suffisant si au repos l'objet regardait vers le fils.
2. Dans le cas de plusieurs fils, ou dans le cas où il y a une rotation initiale au repos, il faut stocker la rotation initiale et la combiner avec la rotation de la cinématique inverse.

Le moyen de tester et de valider votre code est de charger un modèle avec rigging (un maillage attaché au squelette). Voir pour ceci la partie « MecAnim » de ce TP. Si le skinning (déformation du maillage) reste correcte quand vous bouger le squelette alors votre code est juste.

5. Comparaison avec l'IK d'Unity

Unity implémente déjà un algorithme d'IK sur les personnages humanoïdes. Voir le TP « MecAnim, partie 3 ». Placez deux squelettes humanoïdes l'un à côté de l'autre et comparez les deux approches d'IK.