

Rapport de projet RMI

Joao Fernandes Lopes, Robin Caradec

January 2025

1 Introduction

Le projet consiste en la conception et le développement, sous Unity, d'un véhicule doté d'un bras articulé capable d'effectuer des rotations sur ses différents joints. Ce bras doit être en mesure d'exécuter des rotations en appliquant le Modèle Géométrique Direct (MGD) et de déterminer des angles de rotations nécessaires pour atteindre une cible en appliquant le Modèle Géométrique Inverse (MGI).

De plus, le véhicule doit être en mesure de suivre une trajectoire passant par plusieurs waypoints.

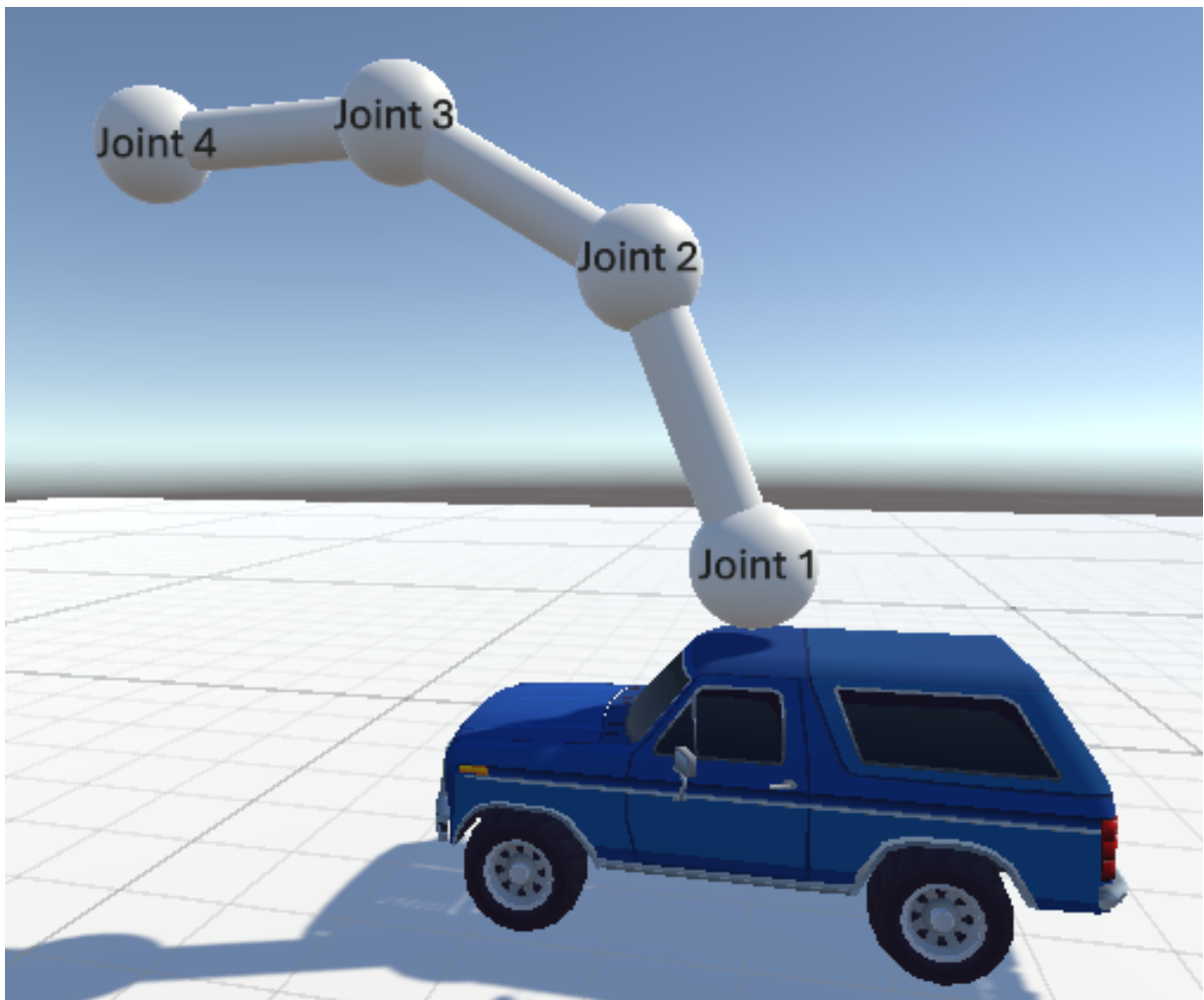


Figure 1: Véhicule équipé d'un bras articulé

2 Guide d'utilisation et contraintes

Pour la mise en place de l'environnement, on crée une plateforme classique en GameObject nommée "Plane", que dans nos tests nous agrandissons à une valeur de 20 pour X, Y et Z. Après avoir placé le terrain, on peut trouver dans "All Prefabs" plusieurs versions de notre "Tronco" qui représente notre voiture et son bras. La version finale à prendre est "Tronco Variant" qui possède un bras droit à la verticale.

Plus tard dans les scripts, il sera probablement nécessaire de soit modifier le nom de la Prefab à "Tronco" ou d'aller dans le script "IndirectRotation4.cs" et de modifier la ligne 147 pour que le GameObject corresponde.

Nous pouvons ensuite faire apparaître un certain nombre de points de contrôle au choix, avec les GameObject "Sphère", et de même pour une target avec le GameObject "Sphère" aussi.

Le nom de la "Sphere" qui représente la target doit être renommé "target". En ce qui concerne les points de contrôle, pour éviter les collisions entre la voiture et les points, nous avons décidé de les mettre légèrement en dessous du terrain et d'une taille réduite. Pour les tests nous avons utilisé une taille de 0.1 pour X, Y et Z ainsi qu'une élévation de -0.05 pour la position en Y, le reste est au choix.

Les scripts et les objets correspondants sont les suivants : RotationMatrixBaseTest.cs avec le joint à la base, RotationMatrix1 avec le joint1, RotationMatrix2 avec le joint2 et IndirectRotation4 avec le dernier joint. Le dernier script "MoveWaypoints" pour la trajectoire associée à l'objet "Tronco" ou "Tronco Variant" si non modifié.

Maintenant, pour tester individuellement le modèle direct, on veut activer les scripts sur chaque joint et avoir le mode "démon" activé sur le script "IndirectRotation4" (activable sur la droite de Unity quand on sélectionne le dernier joint). Une fois fait, on peut faire bouger individuellement chaque joint avec des sliders sur la droite de Unity.

Pour essayer le modèle inverse, il faut absolument avoir désactivé les scripts du modèle direct (sur la base, le joint1 et joint2), et avoir le script de trajectoire activé aussi si on veut tester la trajectoire avec, mais pas obligé.

Pour la trajectoire, veuillez à avoir les points de contrôle ajoutés à la liste présente sur la droite de Unity quand on sélectionne "Tronco" ou "Tronco Variant" si non modifié, avec "target" étant le dernier point de contrôle.

3 Modèle Géométrique Direct

3.1 Implémentation

L'implémentation du modèle géométrique direct commence par la création de classes permettant de manipuler les matrices de rotation (autour des axes X, Y et Z) ainsi que les matrices de translation.

Conformément au cours, ces matrices sont utilisées pour calculer la translation entre chaque articulation du bras et déterminer les angles de rotation à appliquer à chaque joint. Cela permet de définir précisément l'orientation de chaque joint du bras en fonction des paramètres de rotation en entrée.

$$\underbrace{\overbrace{\mathcal{R}(\vec{z}, q_1)}^{^0T_1(q_1)} \overbrace{\mathcal{T}(\vec{x}, L1)}^{^1T_2(q_2)} \overbrace{\mathcal{R}(\vec{z}, q_2)}^{^2T_3(q_3)} \overbrace{\mathcal{T}(\vec{x}, L2)}^{^3T_E} \overbrace{\mathcal{R}(\vec{z}, q_3)}^{^3T_E} \overbrace{\mathcal{T}(\vec{x}, L3)}^{^3T_E}}^{^0T_E(q)}$$

Figure 2: Extrait du MGD vu en cours

Préalablement, on s'assure de calculer les distances qui séparent chaque joint entre eux à partir des positions locales. À partir de ces valeurs, nous sommes en mesure de déterminer les rotations finales aux joints.

```

1  /*
2  *   Déterminer la rotation à appliquer sur un joint
3  *   Input :
4  *       - tetaX : l'angle cible sur X
5  *       - tetaY : l'angle cible sur Y
6  *       - tetaZ : l'angle cible sur Z
7  *       - position : la position du joint précédent
8  *   Output :
9  *       - La rotation finale à appliquer sur le joint
10 /*
11 public Matrix4x4 computeRotationPosition(float tetaX, float tetaY, float tetaZ,
12     Matrix4x4 position)
13 {
14     return position * new RotationMatrixX(tetaX).ToMatrix4x4() * new
15     RotationMatrixY(tetaY).ToMatrix4x4() * new RotationMatrixZ(tetaZ).
16     ToMatrix4x4();
17 }
18
19 /*
20 *   Déterminer la translation à effectuer entre deux joints
21 *   Input :
22 *       - distX : la distance sur l'axe X
23 *       - distY : la distance sur l'axe Y
24 *       - distZ : la distance sur l'axe Z
25 *       - position : la position du joint précédent
26 *   Output :
27 *       - La translation entre les joints
28 */
29 public Matrix4x4 computeTranslation(float distx, float disty, float distz,
30     Matrix4x4 position)
31 {
32     TranslationMatrix trans = new TranslationMatrix(new Vector3(distx, disty,
33     distz));
34     return position * trans.ToMatrix4x4();
35 }

```

3.2 Difficultés rencontrées

Les principales difficultés rencontrées ont été, dans un premier temps, de déterminer à quel moment les translations intervenaient dans le calcul de la rotation finale de chaque joint. Initialement, nous avons développé une version de la rotation qui ne prenait pas en compte ces translations, car les rotations cibles étaient locales à chaque joint. Ce n'est que lorsque nous avons dû faire en sorte que chaque angle soit calculé en fonction des rotations précédentes que nous avons compris l'importance de la succession des étapes Rotation/Translation dans le calcul.

Nous avons également rencontré des difficultés lors de la modélisation du bras au départ. N'ayant que peu d'expérience avec Unity, nous avons d'abord tenté d'ajouter l'élément "joint" de Unity à chaque sphère reliant les cylindres entre eux. Cependant, la solution la plus fonctionnelle a finalement été d'imbriquer chaque segment du bras dans l'arborescence, ce qui a permis une meilleure gestion des positions des joints par rapport à leur joint parent précédent.

4 Modèle Géométrique Inverse

4.1 Implémentation

Notre bras articulé possédant trois degrés de liberté, nous avons deux approches possibles pour le calcul des angles :

- Calculer individuellement les angles locaux alpha et beta à chaque articulation.
- Imposer un angle fixe au troisième joint et ne déterminer que les angles des deux premiers joints.

Dans notre cas, nous avons opté pour la seconde approche, en fixant l'angle du troisième joint à 180° (orienté vers le bas). Cette simplification permet de ramener le problème à deux joints uniquement, ce qui facilite le calcul des angles.

Ensuite, nous avons implémenté le calcul des angles α et β à partir des distances entre les articulations et la cible, en utilisant la géométrie du bras articulé. Les étapes de calcul sont les suivantes :

1. **Alignement du bras vers la cible :** Nous avons d'abord utilisé la fonction `AlignTowardsTarget()` pour orienter le bras vers la cible, en projetant le vecteur de direction sur le plan vertical ZY . Cette fonction effectue une rotation du bras autour de l'axe Z afin de garantir que le bras pointe dans la direction de la cible, tout en évitant un retournement si la cible est derrière le bras.
2. **Calcul de la distance cible et du vecteur de direction :** Une fois le bras orienté vers la cible, nous avons calculé la direction à suivre à partir du premier joint jusqu'à la cible. Cette direction a été projetée dans le plan XZ , en supprimant la composante verticale de la direction. Cette projection permet de travailler dans un plan 2D et d'éviter les effets de la rotation autour de l'axe Y .
3. **Calcul de l'angle θ :** Le premier angle à déterminer est l'angle θ , qui correspond à l'orientation du bras dans le plan ZY . Cet angle est calculé à partir de la direction projetée et est donné par la relation suivante :

$$\theta = \text{atan2}(\text{direction}_z, \text{direction}_y)$$

Cet angle permet d'orienter le bras dans le plan vertical pour se rapprocher de la cible.

4. **Calcul de l'angle α :** L'angle α est déterminé à l'aide de la loi des cosinus dans le triangle formé par les trois joints et la cible. L'angle α permet de déterminer la flexion du bras entre le premier et le deuxième joint pour atteindre la position cible. Le calcul de α se fait par la relation :

$$\cos(\alpha) = \frac{L1^2 + D^2 - L2^2}{2 \cdot L1 \cdot D}$$

où $L1$ est la distance entre le premier et le deuxième joint, $L2$ est la distance entre le deuxième et le troisième joint, et D est la distance entre le premier joint et la cible.

5. **Calcul de l'angle β :** L'angle β représente la flexion du bras entre le deuxième et le troisième joint. Il est également calculé en utilisant la loi des cosinus dans le triangle formé par le premier, le deuxième, et le troisième joint, ainsi que la cible. L'angle β est donné par :

$$\cos(\beta) = \frac{L1^2 + L2^2 - D^2}{2 \cdot L1 \cdot L2}$$

6. **Application des angles aux joints :** Une fois les angles α et β déterminés, nous appliquons les rotations correspondantes aux joints 1 et 2. L'angle θ est utilisé pour orienter le bras dans le plan vertical, tandis que α et β sont utilisés pour ajuster la flexion du bras aux articulations. Si la cible se trouve devant le bras, les angles sont ajustés en conséquence pour garantir un mouvement fluide.

Ce calcul des angles permet de positionner efficacement le bras articulé en fonction de la position de la cible, tout en maintenant une géométrie valide du bras pour éviter des configurations impossibles ou des déformations de l'articulation. L'implémentation a été réalisée dans la fonction `ComputeAngles()`, qui effectue ces calculs à chaque mise à jour de la position de la cible.

4.2 Difficultés rencontrées

L'une des principales difficultés rencontrées a été la détermination de l'offset θ , qu'il était nécessaire d'ajouter à l'angle de rotation du premier joint pour garantir une orientation correcte du bras.

De plus, nous avons intégré une contrainte supplémentaire afin de nous assurer qu'une fois les angles appliqués, le bras ne traverse pas l'intérieur du véhicule. Cette précaution vise à éviter les collisions et à garantir un mouvement réaliste et fonctionnel.

5 Trajectoire

5.1 Implémentation

Pour l'implémentation des trajectoires, il y a eu plusieurs étapes. Dans un premier temps, nous avons d'abord fait une trajectoire simple, avec une ligne droite entre chaque point de contrôle, dans le but de tester si notre modèle géométrique inverse fonctionne correctement. Tout fonctionnait comme prévu, nous avons pu faire quelques réglages sur la trajectoire, notamment régler la distance à laquelle la voiture s'arrête de la target, etc...

Abordons maintenant les trajectoires avec splines naturelles. Nous avons le choix entre deux méthodes vues en cours, une première avec des splines cubiques locales et une deuxième avec des splines cubiques. La première, nous avons besoin de calculer les coefficients localement pour chaque segment de notre trajectoire, mais nous avons alors besoin de résoudre plusieurs systèmes d'équations les uns après les autres. Pour la deuxième méthode globale, on pourrait alors résoudre un système d'équations global pour l'entièreté de notre trajectoire. Nous avons opté pour la première option, avec des splines cubiques locales.

Dans un premier temps, on a notre fonction "EvaluateSpline()", qui va nous permettre d'avoir notre position actuelle sur la trajectoire de la spline. Dans cette fonction, on a une valeur globale "t" qui va être la valeur qui nous permet de connaître notre position globale sur la spline. On a "Clamp" cette valeur entre 0 et 1, car nous avons un problème de loop, la voiture arrivait au bout du circuit et elle revenait au début. Ensuite, on trouve sur quel segment on se trouve, calcule la portion de ce segment sur laquelle on se trouve, et on évalue finalement les coordonnées X et Y de notre voiture.

Une deuxième fonction pour générer les splines cubiques est "CalculateCoefficients()" qui va nous servir à déterminer les coefficients pour chacun des segments de notre trajectoire, et à les utiliser pour le reste de l'exécution. Dans cette fonction, on crée d'abord un tableau pour nos coefficients, qui va contenir n points fois 8, car on prend 4 coefficients pour la coordonnée X et 4 coefficients pour la coordonnée Y. Pour finir, on place à chaque emplacement correspondant dans le tableau, chaque coefficient d'abord pour X puis ceux pour Y.

Commençons maintenant avec la fonction "Start()". Dans cette fonction, on va d'abord faire certaines vérifications d'intégrité, et une fois que tout est bon, transformer nos points de contrôle en liste pour pouvoir les utiliser dans les deux fonctions précédentes. On peut maintenant calculer les coefficients de notre spline et lancer la génération du tracé de notre spline.

Maintenant, dans la fonction "Update()", nous vérifions à nouveau l'intégrité de notre trajectoire, et nous commençons à avancer sur notre spline. La variable "t" va être incrémentée, simplement avec une formule de mesure de distance, avec $\text{Distance} = \text{Vitesse} * \text{Temps}$ et nous ajoutons une division par le nombre de points de contrôle pour maintenir une valeur entre 0 et 1. Une distance importante est la distance entre la voiture et le dernier point de contrôle où nous devons nous arrêter, nous la mettons de côté. Maintenant, si la distance entre la voiture et le dernier point de contrôle est inférieure à X une valeur arbitraire, on s'arrête en mettant la variable "hasStopped" à true. Pour chaque valeur de "t" on utilise ensuite "EvaluateSpline()" pour connaître notre position sur le segment actuel, et sauvegarder cette position. Finalement, on ajuste la direction et l'orientation de la voiture pour un trajet fluide.

5.2 Difficultés rencontrées

Beaucoup de difficultés ont été rencontrées lors de cette partie. La première, après avoir fait la trajectoire simple d'un point à un autre, était de comprendre comment fonctionnaient les splines, beaucoup de recherche et de tentatives ont été faites. On a d'abord commencé par essayer une approche spline cubique locale, car c'était dit comme la forme la plus simple de faire, mais sans succès au départ. Nous avons ensuite essayé les splines cubiques globales qui dans notre contexte pouvaient être simples aussi. Encore une fois sans succès, nous avons commencé à regarder des dérivations, comme par exemple les "Cubic Hermite spline", qui ont commencées à donner quelque chose qui ressemblait à une trajectoire, mais qui ne marchait pas encore correctement. Après plusieurs recherches sur le sujet, nous avons trouvé plusieurs tutoriels, problèmes que les gens ont rencontré, et solutions en utilisant les "splines naturelles", recherches mise dans la section "Références". Un des problèmes spécifiquement auxquels nous avons eu affaire était, par exemple, lors du tracé de notre trajectoire, des traits que la voiture ne suivait pas apparaissaient sans cesse, ce qui faisait des confusions sur la véritable trajectoire de la voiture, ce que nous avons résolu en retouchant la fonction de dessin de trajectoire. Un problème majeur que nous avons eu était aussi le fait que la voiture "loopait" sans arrêt le circuit, une fois arrivée à la fin, elle revenait au départ et repartait. Ce problème était un souci de débordement de notre valeur "t" qui ne pouvait pas arriver à 1. Nous avons alors résolu ce problème avec la fonction "Clamp" et en l'arrêtant juste avant la valeur 1.

References

- [1] https://www.youtube.com/watch?v=7j_BNf9s0jM&t=481s
- [2] <https://www.habrador.com/tutorials/interpolation/1-catmull-rom-splines/>
- [3] <http://www.gamedev.net/reference/articles/article1808.asp>
- [4] [https://en.wikipedia.org/wiki/Spline_\(mathematics\)](https://en.wikipedia.org/wiki/Spline_(mathematics))
- [5] https://en.wikipedia.org/wiki/Cubic_Hermite_spline
- [6] Cours de RMI
- [7] https://fr.wikipedia.org/wiki/Matrice_de_rotation
- [8] <https://www.unity3d-france.com/unity/phpBB3/viewtopic.php?t=13592>
- [9] <https://www.youtube.com/watch?v=MULuu-bkPKs>