Table des matières

[1) Introduction 2](#_Toc473033666)

[a) Mandats de recherche initial 2](#_Toc473033667)

[b) Mandats de recherche effectués 2](#_Toc473033668)

[2) Lexique 2](#_Toc473033669)

[3) Commentaire technique global 2](#_Toc473033670)

[a) Outils 2](#_Toc473033671)

[i. Visualisation des contraintes 2](#_Toc473033672)

[ii. Modification des limites 3](#_Toc473033673)

[iii. Évaluation de l’algorithme 4](#_Toc473033674)

[b) Problématique rencontrées et réglées 5](#_Toc473033675)

[i. ROM selon les angles d’Euler 5](#_Toc473033676)

[ii. DOF using Quaternion (pitch/yaw) and vectors (roll) 6](#_Toc473033677)

[c) Problématique rencontré et à résoudre 7](#_Toc473033678)

[i. Calibration 7](#_Toc473033679)

[ii. DOF using Quaternion only (yaw,pitch,roll) 7](#_Toc473033680)

[iii. Décomposition indépendante sur 3 axes 7](#_Toc473033681)

[4) Revue de code 8](#_Toc473033682)

[5) Liste des impacts dans le code 9](#_Toc473033683)

# Introduction

## Mandats de recherche initial

Heddoko a été approché par le CDRIN pour bénéficier d’une expertise en capture du mouvement, effectuer une revue de code, et aider la compagnie à calibrer leur solution logicielle.

## Mandats de recherche effectués

La revue de code a été effectuée au début du projet afin de mieux prendre en main la base de code existante. Les modifications au code ont été principalement apportées sur les classes BodySegment.cs, et ont été poussées sur le serveur, sur une branche spécifique : CDRIN.

Par la suite, il a été compliqué de concrétiser un premier axe de travail. Le CDRIN a donc proposé de commencer avec un mandat que nous avons cru moins complexe : *range of motion (ROM)*, c’est-à-dire de limiter les rotations transposées au squelette selon un modèle biomécanique. Le ROM peut être vu comme une étape de la calibration où l’acteur va réaliser une série de mouvements pour déterminer ses limites articulaires.

# Lexique

|  |  |
| --- | --- |
| Rotations : | File:Yaw Axis Corrected.svg  Représentation Yaw, Pitch, Roll |
| Quaternion : | https://upload.wikimedia.org/wikipedia/commons/thumb/a/a1/Eulerangles.svg/688px-Eulerangles.svg.pngUn quaternion est un objet mathématique utilisant les nombres complexes et réels. Géométriquement parlant, ils peuvent être pensés comme un axe de rotation, ainsi qu’un angle. Les 4 composantes du quaternion sont alors    avec :   * l’angle de rotation, et * l’axe de rotation. |
| C# : | Langage de programmation orienté objet utilisé dans Unity. |
| Fonction : | Une fonction est une série d’actions programmables pour réaliser une action définie. Une fonction requiert souvent des données pour réaliser sa tâche, ce sont les paramètres, et retourne souvent un résultat. |
| Surcharge de fonction : | La surcharge de fonction permet de donner le même nom à une fonction qu’une autre avec des paramètres différents. |
| Paramètres optionnels : | Les paramètres optionnels permettent d’appeler une fonction avec certains paramètres définis par défaut. |

# Commentaire technique global

## Outils

L’ensemble du développement a été réalisé sous Unity 3D 5.3.4f1 en c#.

### Visualisation des contraintes

Des outils de débogage et de visualisation ont été développés afin de faciliter la recherche et la compréhension géométrique des algorithmes développés (c.f. figures 1 à 4 ci-dessous). On peut notamment remarquer sur ces figures :

* deux arcs de cônes bleus qui représentent la limite Pitch intersectée avec la limite Yaw,
* deux cercles bleus qui représentent la limite Pitch,
* deux arcs de disques verts qui représentent l’intersection de la limite Yaw avec Pitch,
* Une sphère grise qui représente la sphère unitaire,
* Des flèches :
  + Turquoise (orientation dans les limites) ou noire (orientation hors limites) le long de l’objet qui représente l’orientation l’objet,
  + Rouge qui représente la projection de l’objet sur un des deux cônes Pitch (limites supérieure ou inférieure),
  + Un système d’axe orienté (vert, bleu, rouge) le long de la projection pour visualiser le Roll de l’objet;
* Un cercle représentant l’intersection de la projection de l’objet sur le cône Pitch correspondant.
  + turquoise si la projection pitch est sur le cône supérieur et bonne,
  + rouge si la projection pitch est sur le cône supérieur et mauvaise (supérieure),
  + verte si la projection pitch est sur le cône inférieur et bonne,
  + noire si la projection pitch est sur le cône inférieur et mauvaise (inférieure),
* Deux segments noirs décomposant l’orientation de l’objet (flèche turquoise ou noire)

|  |  |
| --- | --- |
| Figure 2 projection supérieure Pitch max | Figure 3 projection OK Pitch max |
|  |  |
| Figure 4 projection OK pitch min | Figure 5 projection inférieure Pitch min |

### Modification des limites

Pour faciliter les itérations de modifications, certains paramètres du ROM sont exposés dans l’éditeur Unity. Principalement, il s’agit de pouvoir indiquer les limites à ne pas dépasser sur les 3 axes de chaque articulation.

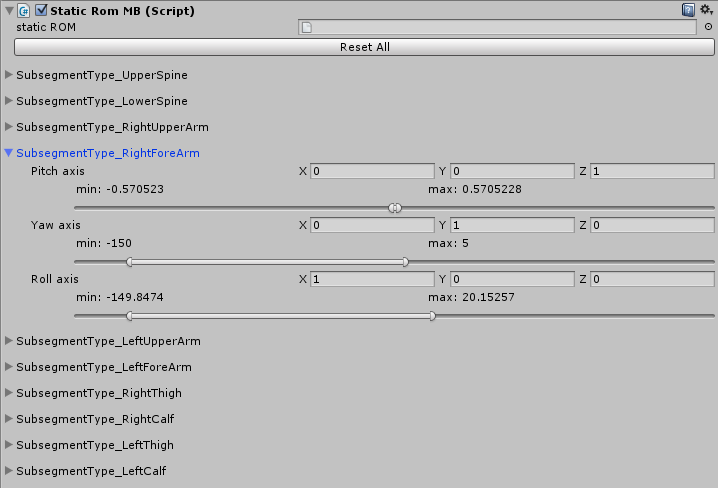


Figure Exposition des limites d'axes par articulation

Il est également possible dans Unity de définir des *labels* qui seront sélectionnable dans l’éditeur pour faciliter la modification des limites articulaires des membres. Il est à souligner que sans les labels, la sélection d’un membre particulier dans la vue 3D n’est pas possible.

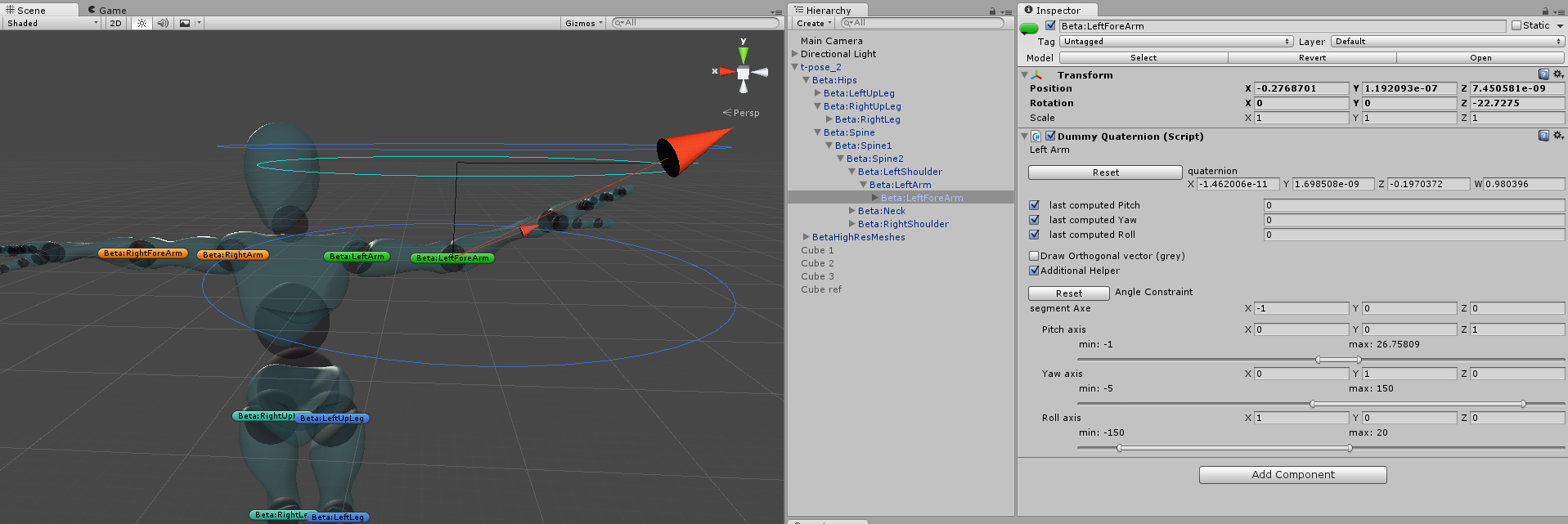


Figure labels dans la vue 3D de unity, une couleur pour chaque membre

### Évaluation de l’algorithme

Afin de bien évaluer les algorithmes mis en place, le code source a été légèrement modifié ou *hacké* temporairement (afin de modifier le comportement à l’extérieur du code original sans le modifier). Par exemple, la camera est modifiée quand on active le ROM afin de montrer deux avatars, l’un étant soumis au ROM, l’autre non. De plus, un système d’axe est affiché sur chaque articulation pour mieux visualiser les rotations.

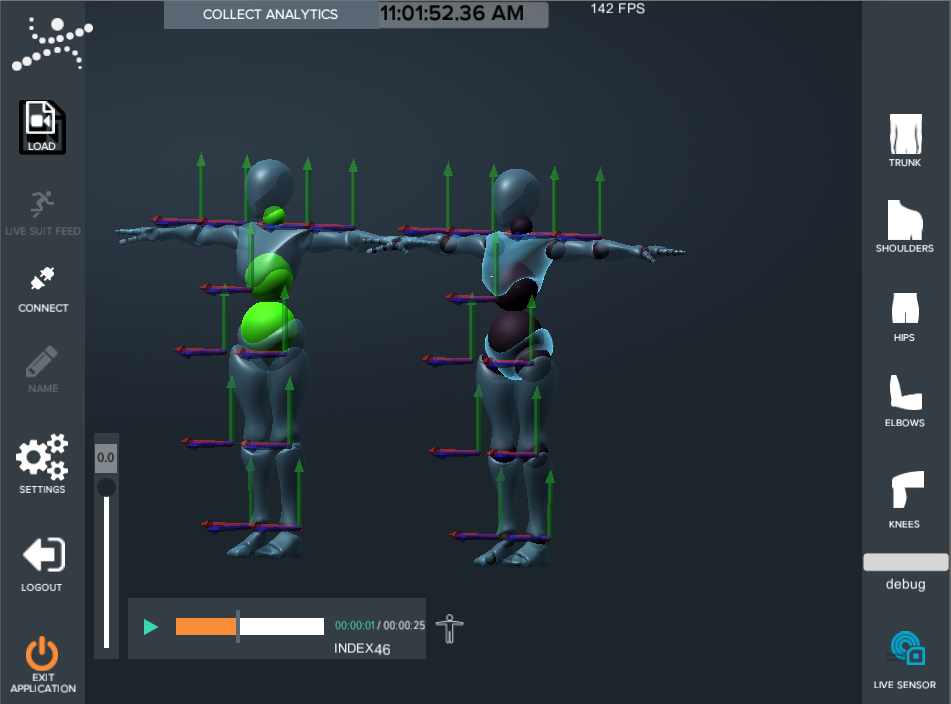


Figure modification de la caméra pour visualiser un avatar soumis au ROM

## Problématique rencontrées et réglées

### ROM selon les angles d’Euler

La solution la plus simple pour mettre en place un ROM est de concevoir une limite par axe, et donc de limiter chaque axe individuellement. Les angles d’Euler permettent d’orienter un solide dans l’espace en décomposant la rotation finale en trois rotations subséquentes.

Unity utilise les quaternions pour représenter les rotations. Il est possible de convertir les angles d’Euler en quaternion et vice versa, avec certaines contraintes. En effet, l’avantage d’utiliser les quaternions est qu’ils ne possèdent pas le phénomène de *gimbal lock* (blocage du cardan). Il survient quand deux des trois axes d’une rotation dans l'espace sont colinéaires. Cette contrainte peut être évitée en inversant des axes de 180°.

La première implémentation du ROM utilise les angles d’Euler. Il est effectivement très simple de convertir dans Unity un quaternion, de calculer la différence et de geler une rotation dès qu’elle dépasse la limite permise. Le problème survient sur les coudes et genoux, lorsque la flexion de l’articulation franchit 90°. Dans les figures Figure 9 et Figure 10, on remarque qu’une rotation d’environ 89° puis 91° sur l’axe X est transformée pour éviter l’apparition du phénomène, et la perte d’un degré de liberté. Ceci complique alors le calcul des limites d’angles puisqu’il faut prendre certaines fois en compte les multiples de 180 sur les autres axes pour calculer l’angle théorique d’un axe.

|  |  |
| --- | --- |
| Figure 9 Rotation avant un gimbal lock (89° sur l'axe X) | Figure 10 Rotation après gimbal lock évité (91° sur l'axe X) |

Pour cette raison, il a été décidé d’utiliser une autre méthode.

### DOF using Quaternion (pitch/yaw) and vectors (roll)

C’est la solution la plus prometteuse élaborée pendant le projet, et aussi est la dernière, mais il aura manqué de temps pour une intégration finale et fonctionnelle.

Il s’agit d’utiliser la représentation vectorielle d’une rotation sur la sphère unitaire. Le quaternion représentant un espace à 4 dimensions, il peut être restreint à un espace à 3+1 dimensions. En effectuant une rotation d’un vecteur directeur par le quaternion, on perd la rotation Roll. Cette rotation peut être retrouvée en comparant les deux rotations : celle initiale, et celle résultante.

En réduisant l’espace de contrainte, il devient plus facile de valider les limites. Ainsi, on remarque qu’en appliquant une rotation Yaw (axe Y dans Unity) à un objet, l’extrémité du vecteur directeur décrit constamment un cercle, quelle que soit la rotation Pitch appliquée sur l’objet. On peut donc représenter les limites Yaw (Figure 11) et Pitch (Figure 12) indépendamment par deux demi-disques, et l’intersection de ces deux limites par des arcs de disques et des arcs de cônes respectivement (Figure 13).

|  |  |  |
| --- | --- | --- |
| Figure 11 représentation limites Yaw | Figure 12 représentation limites Pitch | Figure 13 intersection Yaw Pitch |

Pour savoir si le Pitch est dans les limites définies, l’algorithme calcule la projection du vecteur directeur sur les cônes de limites inférieures et supérieures. On peut ensuite facilement savoir si l’orientation de l’objet est dans la limite en comparant les rayons des disques de bases sur les cônes minimum et maximum.

Pour savoir si le Yaw est dans les limites définies, le calcul est plus simple : il suffit de comparer le signe des produits scalaires entre les normales des plans minimum et maximum et le vecteur directeur de l’objet. Si le signe des deux projections est différent, l’objet est situé entre les deux limites.

## Problématique rencontrée et à résoudre

### Calibration

La problématique de la calibration des capteurs inertiels n’a pas été abordée, et reste donc à résoudre.

### DOF using Quaternion only (yaw,pitch,roll)

L’utilisation des quaternions seulement pour résoudre le problème ne s’est pas avéré être une bonne solution. Les quaternions représentent une rotation 3D dans un espace à 4 dimensions.

Il est possible d’interpoler deux quaternions sur l’hypersphère, avec la formule suivante :

où est l’angle entre les deux quaternions.

Le problème ici est de trouver le quaternionqui représente la projection du quaternion étudié sur les grands-arcs définis par nos contraintes, c’est-à-dire, par exemple, trouver entre et qui minimise la distance avec(notre quaternion de base), où les indices représentent respectivement les rotations extrêmes : (en extrapolant pour les combinaisons possibles).

Les rotations étant indépendantes les unes des autres, il faut alors tester indépendamment les rotations en fixant les deux autres axes, ce qui donne alors 4 comparaisons possibles par axe, soit 43 = 64 projections.

Un algorithme en pseudo code pour chaque projection pourrait ressembler à

**Quaternion** Q\_min, Q\_max, Q\_current;

**Quaternion** slerp; // resultat de la projection avec un pas de 0.1

**float** prev = FLOAT.MAX;

float t;

**for**(t = **0.0f**; t< **1.0f**; t += **0.1f**)

{

slerp = **Quaternion**.Slerp(Q\_min, Q\_max, t);

**float** angle = **Mathf**.Acos(Quaternion.Dot(s, Q\_current));

**if**(prev < angle)

prev = angle;

**else**

**break**;

}

### Décomposition indépendante sur 3 axes

Plutôt que de travailler avec les angles d’Euler, il a été pensé une solution pour projeter le quaternion sur trois axes de manière indépendante. La solution utilise des vecteurs pour comparer l’angle généré pour chaque vecteur d’un repère orthonormé avec l’origine.

public void ExtractAngles(Quaternion a\_QuatLocal, ref Vector3 a\_Angles)

{

**Vector3** localForward = a\_QuatLocal \* **Vector3**.forward;

**Vector3** localRight = a\_QuatLocal \* **Vector3**.right;

**Vector3** localUp = a\_QuatLocal \* **Vector3**.up;

**Vector3** upProj = **Vector3**.ProjectOnPlane(localUp, **Vector3**.right);// pitch

**Vector3** forwardProj = **Vector3**.ProjectOnPlane(localForward, **Vector3**.up);//yaw

**Vector3** rightProj = **Vector3**.ProjectOnPlane(localRight, **Vector3**.forward);//roll

upProj.Normalize();

forwardProj.Normalize();

rightProj.Normalize();

a\_Angles.x = **Vector3**.Angle(**Vector3**.up, upProj);

a\_Angles.y = **Vector3**.Angle(**Vector3**.forward, forwardProj);

a\_Angles.z = **Vector3**.Angle(**Vector3**.right, rightProj);

}

Cette solution permet de bien décomposer une rotation, et de restreindre les mouvements sur chaque axe. La complexité vient de la reconstruction du quaternion à partir de ces angles : toutes les solutions trouvées n’étaient que partielles et perdaient un degré de liberté. En effet, on obtient un angle par axe, mais ceux-ci diffèrent des angles d’Euler traditionnels.

# Revue de code

Pour commencer, un nettoyage des fichiers mis en communs sur le serveur git a été effectué (commits f450054, a7786c5, 3aef934, 760107a, 8a1b15d, e235ae1, e944fe6)

La plus grosse modification a eu lieu dans la classe *BodySegment.cs*. En effet, cette classe était implémentée sur 3 fichiers différents selon des macros définies dans Unity. La plupart du code était copié entre les fichiers, et il est difficile de corriger un potentiel bug dans trois fichiers similaires, mais pas identiques. Pour ne pas détruire le travail précédent, la classe a été définie dans un nouveau fichier BodySegment\_merge.cs, entouré par la macro CDRIN\_BODY\_SEGMENT. Les autres macros sont alors toujours accessibles, et il suffit de désactiver la définition de cette dernière pour retrouver le code original. Tout le développement réalisé à propos du ROM par le CDRIN est utilisé depuis la classe définie dans ce fichier. Il est ainsi peu imbriqué dans le reste de l’application.

Dans le tableau suivant vous trouverez certains des refactors importants effectués :

|  |  |  |
| --- | --- | --- |
| Classe | Nature | Raison (# commit) |
| TorsoAnalysis | Refactor | aider la relecture du code |
| RecordingProgressSubControl | Add function | Update model while dragging progress bar (#815a945) |
| RecordingPlayerView | Improve stability | Improve reload feature (#bbac0f3) |
| Add reference | ROM visualization (#0123693) |
| Flags | Add class | flags updated in a struct for easy save/load of all flags (#ba83d2a) |

# Liste des impacts dans le code

La revue de code a principalement consisté à supprimer du code inutile et rassembler certaines portions de code pour faciliter la relecture (ex : BodySegment.cs)

Comme vu précédemment, la classe BodySegment est la plus impactée par le refactoring. Malheureusement, un problème d’axe qui venait finalement des capteurs ou de l’enregistrement des données a nécessité plusieurs semaines de travail non productives. En effet, les systèmes d’axes différaient sur certains fichiers, et nous avons principalement remis en question le code plutôt que les données.

Certains prototypes de fonctions ont également été modifiés. Une attention particulière a été apportée tout au long des développements pour que ceux-ci s’intègrent dans le code existant avec un minimum de modification. Ainsi, les prototypes de fonctions modifiées le sont avec des paramètres optionnels, ou en surchargeant les fonctions.