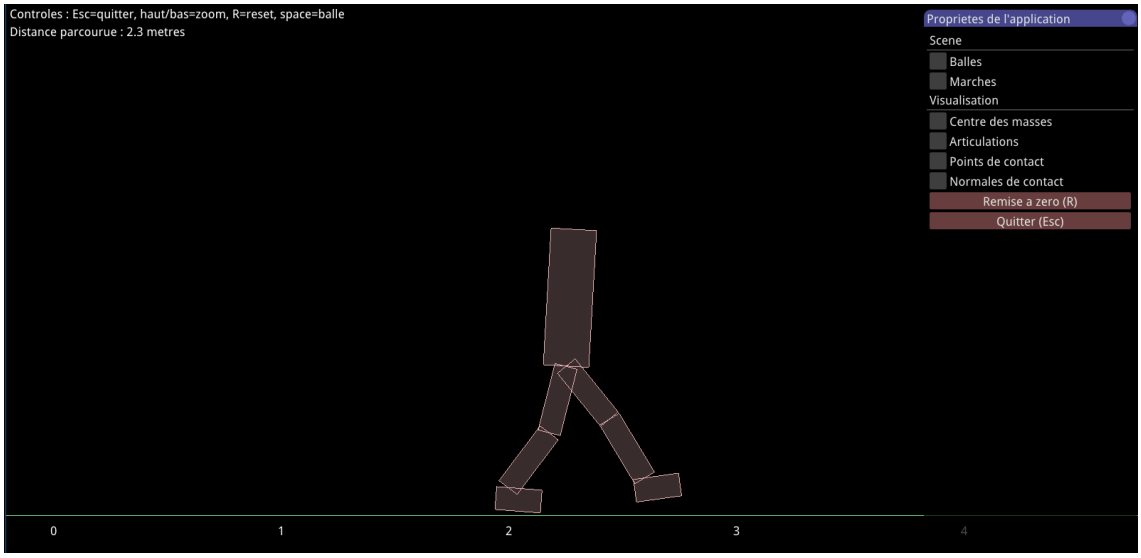
M2 ID3D – Animation, corps articulés et moteurs physiques

TP contrôleur de mouvement

Objectif du TP : Contrôler la marche d'un bipède en 2D



Description courte : Ce TP est réalisé individuellement. Il repose sur les principes introduits en cours (animation cinématique, principe physique et contrôleur). Vous avez à prendre en main plusieurs éléments du contrôleur et à développer un nouvel élément. Votre TP sera évalué sur la base du code fourni, d'un rapport à rédiger et d'une présentation orale ou vidéo de quelques minutes.

Préparation : Téléchargez l'archive correspondant au TP contrôleur sur le site de l'UE. Cette archive contient :

- un répertoire **`Box2D`** contenant les librairies, les fichiers sources, des exemples, un manuel et les projets CodeBlocks permettant de compiler les librairies du moteur physique Box2D
- un répertoire **`src`** contenant le squelette des classes à compléter
- un projet CodeBlocks (Windows et Linux) pour compiler et exécuter le programme

Il est fortement conseillé de travailler sous Windows car tout est déjà précompilé. Si vous souhaitez travailler sur votre machine sous Linux, vous devrez installer CodeBlocks et les librairies GLFW et GLEW vous-même (commande : `apt install codeblocks libglfw3-dev libglew-dev`) et compiler les libraires Box2D et IMGUI grâce aux projets CodeBlocks fournis.

En compilant et exécutant le projet TP_CONTROLEUR fourni, vous verrez le bipède se tenir droit.

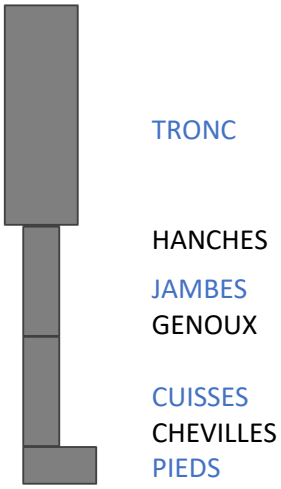
Sur la droite de l'application se trouve un menu (que vous pouvez étendre dans le fichier **`Framework/Main.cpp fonction sInterface`**, les valeurs par défaut des options étant définies dans **`Framework/Application.h structure Settings`**). Dans ce menu, vous pouvez déjà activer/désactiver :

- le lancement de balles sur le bipède (pour tester la robustesse du contrôleur)
- la visualisation du centre de masse
- la visualisation d'autres objets (articulations, points et normales de contact)
- l'apparition de marches au sol (pour tester la robustesse du contrôleur)

Vous pouvez également pauser la simulation, la réinitialiser ou quitter le programme.

Le bipède : Le bipède est composé de sept corps rigides (2 pieds, 2 jambes, 2 cuisses et 1 tronc) connectés par des articulations à un degré de liberté de rotation chacun (chevilles, genoux et hanches). Le monde physique est en 2D, les axes de rotation sont donc orthogonaux à cet espace 2D.

- Le tronc est une boite de 20 cm de largeur et 60 cm de hauteur
- Les jambes et les cuisses sont des boites de 10 cm de largeur et 30 cm de hauteur
- Les pieds sont des boites de 20 cm de largeur et 10 cm de hauteur
- Les hanches relient les jambes au tronc
- Les genoux relient les cuisses aux jambes
- Les chevilles relient les pieds aux cuisses
- Les hanches et chevilles peuvent s'orienter dans l'intervalle $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ et les genoux dans l'intervalle $\left[0, \frac{\pi}{2}\right]$ (ils ne peuvent pas se plier vers l'avant)



Partie « prise en main »

1. Suivi des poses clés

Dans un premier temps, notre but est de permettre au bipède de suivre une pose cible. Pour l'instant la pose cible est une pose où toutes les articulations sont à zéro dans le repère global (représente la pose fixe droite, comme dans la figure ci-dessus). Pour maintenir cette pose, le contrôleur génère les moments articulaires nécessaires pour suivre la pose cible dans le temps. Le suivi est assuré par l'utilisation de régulateurs PD pour chaque articulation.

Etudier la fonction **`Biped::KeyPoseTracking`** et la classe **`PDController`** qui permet d'ajouter dans **`m_motorTarget`** les moments nécessaires au suivi de la pose cible courante. Noter que le constructeur **`Biped::Biped`** construit le tableau **`m_PDControllers`** et donne des valeurs (initiales) aux gains.

Vous pouvez vous assurer que le contrôleur corrige correctement les erreurs dans la pose du bipède dès qu'une perturbation est présente en activant les balles. Vous pouvez également voir ce qu'il se passe si vous modifiez les valeurs des gains des régulateurs.

2. Machine à états finis de poses clés

Vous avez dû remarquer que c'est à la fin du constructeur de **`Biped`**, que la machine à états finis de poses clés utilisée est définie. Deux machines à états finis de poses clés sont présentes dans le code. La première, **`FSM_Stand`**, contient la pose droite (c'était donc en fait une machine à états finis avec un seul état).

Afin de guider le bipède dans un mouvement de marche, une autre machine à états finis de poses clés a été créée, **`FSM_Walk`**. Activer cette machine à états finis en commentant la création de la machine précédente et dé-commentant celle-ci. Au lieu de donner une série de poses cibles à suivre à chaque pas de temps (type mocap) quelques poses clés (6 ici) et une interpolation entre ces poses clés sont utilisées (voir Figure 1).

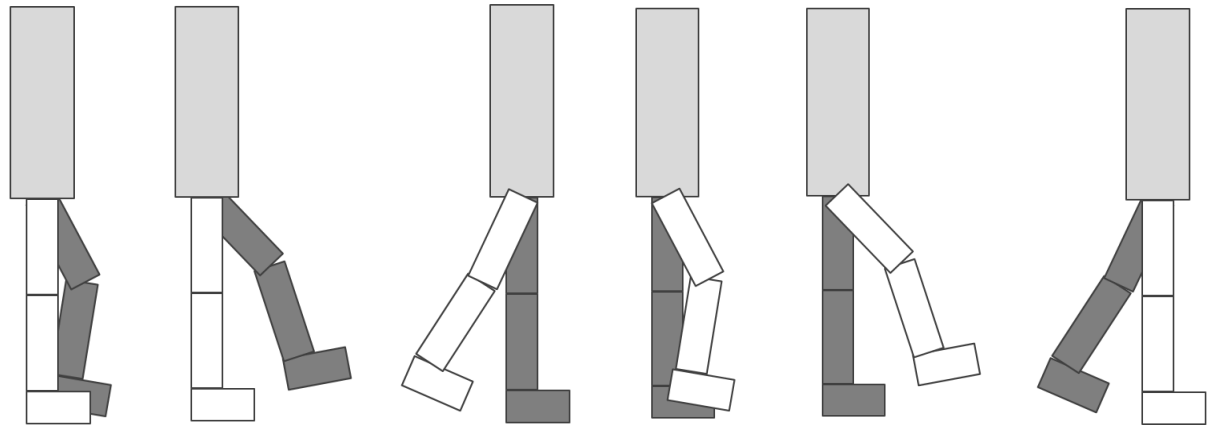


Fig. 1 : Cycle de poses clés symétriques pour la marche normale d'un bipède. Gris = jambe gauche, blanc = jambe droite

Un état de la machine à états finis a été créé par pose et les transitions entre les états sont uniquement basées sur la durée attendue de l'état. La marche étant cyclique, l'état qui suit l'état 5 est l'état 0.

L'angle spécifié dans la pose peut être soit global (orientation d'un corps rigide dans le monde) soit local (angle entre deux corps rigides). Cette information peut être utilisée pour contraindre le contrôleur à suivre une orientation locale (par rapport au corps rigide père) ou globale (par rapport au monde). Par exemple, dans certains états le contrôleur peut essayer de garder le tronc à la verticale dans le monde ou les pieds bien à plat sur le sol, alors qu'il doit essayer de plier le genou correctement (quel que soit l'angle de la hanche).

Etudier les classes **`FSM`**, **`FSM_Stand`** et **`FSM_Walk`** afin de comprendre comment créer et utiliser une machine à états finis de poses clés.

Vous pouvez tester la robustesse du contrôleur en activant les balles et/ou les marches. Vous pouvez également voir ce qu'il se passe quand vous changez des valeurs d'angles ou des durées de transition dans la machine à états.

3. Optimisation du contrôleur

Nous avons déjà pu remarquer que des valeurs de paramètres du contrôleur ont été fixées dans le code, comme par exemple les angles des poses clés, les durées de transition des états, ou les gains des régulateurs PD. Trouver à la main ces valeurs peut s'avérer fastidieux et difficile lorsque le nombre de paramètres est grand. Une solution consiste alors à calculer automatiquement ces valeurs par optimisation. L'idée de base est d'essayer un très grand nombre de combinaisons de valeurs et de ne garder que la meilleure combinaison. Nous devons pour cela définir ce que meilleur signifie pour le mouvement.

Pour activer l'optimisation, vous devez passer le champ **`optimization`** à **`true`** dans la structure **`Settings`** de **`Application.h`**. Si besoin vous pouvez également changer la durée d'optimisation (par défaut 5 secondes) dans le champ **`optiDuration`**.

Dans la structure **`OptimizationData`** de **`Main.cpp`**, le champ **`nbParam`** donne le nombre de paramètres optimisés. Au besoin, une valeur initiale plus grande à **`bestCost`** peut être donnée.

Les meilleures valeurs de l'optimisation sont sauvegardées, à chaque itération, dans un fichier nommé **`"optimization.txt"`**. Pour affecter les valeurs lues depuis le fichier dans les paramètres ou les valeurs à écrire dans le fichier, nous utilisons les deux procédures suivantes : **`Application::setOptimizationData`** et **`Application::getOptimizationData`**. Modifiez **`nbParam`** et les fonctions de lecture/écriture afin d'optimiser les paramètres de votre choix.

Nous avons deux endroits où définir comment un mouvement est défini comme « meilleur » qu'un autre, c'est-à-dire où calculer le coût associé à une simulation :

- Dans la fonction **`getCurrentCost`** de **`Application.h`**, nous pouvons récupérer une évaluation faite au cours de la simulation où le coût peut être incrémenté à chaque pas de temps. Deux exemples sont implémentés : **`sumTorque`** (la somme des carrés des moments) et **`sumAngleVelocity`** (la somme des valeurs absolues des vitesses angulaires). L'idée de ces deux exemples est d'estimer l'énergie dépensée par le bipède afin de la minimiser (principe du moindre effort). Par défaut uniquement les vitesses angulaires sont minimisées.
- Dans la fonction **`main`** de **`Main.cpp`**, nous pouvons calculer un coût global sur la simulation. Par exemple dans le code fourni, nous souhaitons que le bipède aille le plus loin possible (en utilisant sa position à la fin de la simulation **`app->BipedPosition`** et plus précisément on demande au bipède d'avoir parcouru 10 mètres après les 5 secondes de simulation).

Les deux termes de la fonction de coût sont pondérés puis sommés, donnant le coût total pour une simulation. Ce coût est comparé au meilleur coût trouvé jusqu'à présent. Toujours dans la boucle principale du **`main`**, vous trouverez la stratégie d'optimisation choisie. L'idée est de toujours partir de la meilleure configuration, puis de produire aléatoirement une configuration légèrement différente et de la tester. Si cette nouvelle configuration est meilleure que la précédente, alors elle devient la meilleure. Finalement, quoi qu'il arrive, on prend la meilleure configuration, on en produit une nouvelle et on répète le processus. Une stratégie assez simple a été implémentée où la valeur légèrement différente p' de chaque paramètre p vaut : $p' = p \pm random(0, \left|e^{-i*0.001}\right| \times p \times s)$ où i est le nombre d'itérations effectuées et s est un coefficient dans $[0,1]$, ici 0.25, décrivant la proportion de

modification autorisée. Plus on est avancé dans l'optimisation, moins on s'autorise à modifier la meilleure configuration, nous assurant ainsi une certaine convergence vers une solution localement optimale.

A la fin de l'optimisation, vous pouvez ouvrir le fichier **`"optimization.txt"`** et copier les valeurs optimales dans votre code. Le fichier commence par le nombre d'itérations effectuées puis le meilleur coût obtenu et est suivi des valeurs optimales correspondantes (dans l'ordre où elles sont sauvées dans le tableau **`data`** de la procédure **`Application::getOptimizationData`**). Vous pouvez remettre le champ **`optimization`** à **`false`** quand vous souhaitez repasser dans le mode de simulation normal.

Partie « développement »

Dans cette partie, vous allez améliorer ou étendre le contrôleur. Vous devez choisir vous-même le (ou les) développement à implémenter. Pour vous donner de l'inspiration, vous trouverez ci-dessous une liste de développements valides. Le nombre d'étoiles représente la difficulté attendue de l'implémentation complète du développement. Bien évidemment un développement complexe sera plus récompensé qu'un développement facile.

- Implémenter une fonction d'interpolation des poses clés donnant des trajectoires cibles plus continues dans le temps (ex. interpolation cubique, courbe d'Hermite ou de Catmul-Rom) ★
- Ajouter une machine à états pour un autre mouvement (ex. course, saut, nage, combat, monter ou descendre des marches) ★
- Avoir une machine à états dont certaines transitions sont basées sur des évènements (ex. des changements d'appui) ★
- Inclure des termes plus complexes dans la fonction de coût (ex. cyclicité, symétrie, stabilité) ★
- La gestion des limites articulaires permettant au bipède de ne pas s'en approcher trop près. Par exemple en ajoutant des régulateurs proportionnels (i.e. lorsque l'angle courant est proche de la limite, un moment articulaire est ajouté pour s'en éloigner). ★
- Implémenter ou intégrer une stratégie d'optimisation plus performante (convergence plus rapide vers un optimal) comme la Covariance Matrix Adaptation (CMA) ★★
- Implémenter un contrôle interactif et temps-réel de la vitesse de déplacement du bipède. Vous pourrez par exemple appliquer une ou des forces externes, modifier les poses clés, ajouter des moments articulaires ou modifier la durée des états en fonction de la différence entre la vitesse courante et une vitesse désirée. ★★
- Le contrôle par forces virtuelles ou par modèles musculaires ★★
- Conception d'un contrôleur interdisant l'ajout d'énergie externe (somme des moments articulaires qui doit être nulle) ★★
- Des interactions avec des corps déformables ou des fluides (ex. des vêtements, une cape, marcher dans l'eau ou dans la boue) ★★
- La connexion avec un système d'animation cinématique (ex. graphe de mouvement) ★★
- Le calcul des moments articulaires par générateur de patterns cycliques, par réseau de neurones, par apprentissage par renforcement, ou par contrôle optimal ★★★
- Ou autres, soyez créatif (l'originalité est un critère important) mais réaliste, ça doit être faisable en quelques heures

Un développement simpliste tel que l'ajout de bras et tête ou l'application de forces externes pour faire sauter le bipède ne sont pas considérées comme valides. Vous pouvez demander l'accord de votre encadrant de TP si vous doutez de la validité de votre développement.

Pour vous aider dans votre choix et son implémentation, vous pouvez relire les transparents du cours, chercher sur Internet ou lire des articles de recherche en commençant par exemple par « *Interactive Character Animation Using Simulated Physics : A State-of-the-Art Review. Geijtenbeek and Pronost, Computer Graphics Forum, 2012*».

Rendu du TP : Vous devez soumettre une archive contenant le code (exécutable, sources, librairies, projet CodeBlocks et tous les fichiers additionnels nécessaires à l'exécution de votre application), une vidéo commentée de quelques minutes démontrant et expliquant le développement réalisé, et un rapport (Word ou pdf). L'archive sera à déposer sur TOMUSS avant la date limite de dépôt. Pour des raisons de limite de taille de fichier de dépôt, un lien vers la vidéo peut être donné dans le rapport au lieu de mettre la vidéo dans l'archive.

Le rapport (entre 3 et 6 pages, hors annexes et figures) décrit le(s) développement(s) que vous avez choisi et son implémentation. Veuillez à indiquer votre nom, prénom et numéro d'étudiant dans ce document. Ne donner pas de code dans ce document, par contre les algorithmes en pseudocode seront tolérés s'ils aident à la compréhension. L'organisation et le contenu du document va dépendre du choix du développement. Il vous faudra donner des éléments de contexte, expliquer les aspects les plus théoriques, les liens potentiels avec le cours magistral, les références à des articles de recherche si votre développement s'en inspire. Il faudra également que le rapport explique ce qui a été fait, l'interface utilisateur finale, vos choix de conception (ex. classes) et d'implémentation (ex. structures de données ou algorithmes utilisés), les résultats obtenus et les améliorations possibles. Vous pouvez illustrer vos propos avec des images tirées de l'application, des tables, des figures, des graphiques etc.

Les critères d'évaluation sont les suivants :

- Originalité et difficulté
- Qualité de la conception et de la réalisation
- Robustesse, interactivité et stabilité du contrôleur
- Qualité des résultats
- Clarté et organisation de la vidéo ou présentation
- Qualité de la documentation et du rapport
- Capacité à prendre du recul et à décrire les aspects théoriques de la réalisation