Session 17 : Character Controllers

> **L'objectif du jour : Le Gameplay Feel**
>
> Déplacer un avatar dans un monde physique est un défi paradoxal. Un simple cube physique soumis aux lois de Newton semble "ivre" (glisse, roule, bascule). Pour un jeu de tir (FPS) ou de plateforme, on veut une réponse **immédiate** et **précise**.
>
> **Concepts clés :**
> - **DCC vs KCC** : Choisir entre la physique pure ou le contrôle par code.
> - **Projection de Vecteur** : Les mathématiques pour glisser le long des murs.
> - **État du sol** : Savoir si on peut sauter ou si on tombe.
> - **Game Feel** : Coyote Time et Jump Buffering.

1. Le dilemme : Dynamic vs Kinematic

Le choix du type de corps détermine toute l'architecture du code de mouvement.

1.1 Dynamic Character Controller (DCC)

Approche : On utilise un **RigidBody Dynamic** standard (masse > 0). On le déplace en appliquant des **forces** ou en modifiant sa **vélocité**.

- **Implémentation :** `body.addForce(input * speed)` ou `body.setLinVel(input * speed)`.
- **Problème de la Rotation :** Une capsule physique va rouler par terre. Il faut impérativement **verrouiller la rotation** sur les axes X et Z (`Angular Factor = 0`).
- **Problème de la Friction :** Le personnage glisse comme sur de la glace lors de l'arrêt (inertie). Il faut implémenter une friction manuelle élevée ou un "damping" agressif.
- **Utilisation idéale : Rocket League**, **Gang Beasts**, billes roulantes.

1.2 Kinematic Character Controller (KCC)

Approche : On utilise un **RigidBody Kinematic**. Le moteur physique ne l'affecte pas (pas de gravité, pas de rebond). Le code calcule la position exacte P_{n+1} et demande au moteur de "téléporter" l'objet tout en vérifiant les collisions.

- **Implémentation :** Algorithme "Move and Slide".
- **Avantages :** Contrôle total ("Tight controls"). Arrêt instantané, saut précis, pas d'effets de ressort bizarres contre les murs.
- **Inconvénients :** Il faut recoder la gravité et la résolution des collisions soi-même.
- **Utilisation idéale : Quake**, **Call of Duty**, **Celeste**.

2. Mathématiques : L'Algorithme "Move and Slide"

Si un KCC avance tout droit et rencontre un mur à 45°, il ne doit pas se bloquer. Il doit conserver la partie de son mouvement qui est parallèle au mur.

Projection et Rejet. *Soit un vecteur vitesse $\vec{v}$ et un mur de normale $\vec{n}$. On veut supprimer la partie de $\vec{v}$ qui "rentre" dans le mur.*

1. ***Produit Scalaire :*** *Calculer l'intensité de la vitesse dans la direction du mur.*

$$\text{Intensité} = \vec{v} \cdot \vec{n}$$

   (Si le résultat est < 0, on avance vers le mur).

2. ***Rejet (Slide) :*** *On soustrait cette composante au vecteur original.*

$$\vec{v}_{\text{slide}} = \vec{v} - (\text{Intensité} \times \vec{n})$$

Résultat : *Le personnage glisse le long de la surface sans perdre toute sa vitesse .*

> **Le problème du Coin (Corner Trap)**
>
> Une seule projection ne suffit pas ! Si vous glissez le long d'un mur et que vous frappez un deuxième mur (un coin), vous devez "glisser sur le glissement".
>
> **Solution :** L'algorithme doit être une boucle (récursion limitée). `Pour i de 0 à MaxBounces :`
> 1. Avancer.
> 2. Si collision -> Reculer à l'impact, calculer nouveau vecteur glissé, recommencer.

3. Senseurs : Grounded Check & Gravité

Dans un KCC, la gravité est une variable `verticalVelocity` que l'on gère manuellement.

3.1 La détection du sol (Grounded)

Pour savoir si on peut sauter, on ne peut pas juste vérifier "est-ce que je touche quelque chose", car on peut toucher un mur ou un plafond. Il faut toucher quelque chose **en dessous**.

- **Méthode 1 : Raycast.** Un rayon fin part des pieds vers le bas. **Défaut :** Peut rater un rebord ou passer entre deux planches.
- **Méthode 2 : Shapecast (Recommandée).** On projette la forme de la capsule (ou une sphère) vers le bas. Si elle touche à une distance très faible (ε), on est au sol.

3.2 Le "Juice" : Coyote Time & Buffering

Pour qu'un jeu soit agréable, il faut tricher avec la physique.

- **Coyote Time :** Permet de sauter quelques frames **après** avoir quitté une plateforme (comme Wile E. Coyote).
- **Jump Buffering :** Si le joueur appuie sur saut **juste avant** de toucher le sol, l'action est enregistrée et exécutée à l'atterrissage.

4. Travail Pratique : Implémentation du KCC

Objectif : Coder un contrôleur FPS (First Person Shooter) robuste utilisant `Rapier` (ou équivalent) en mode Kinematic.

Étape A : La Capsule et la Caméra

1. Créer un RigidBody **Kinematic**.
2. Lui attacher un Collider **Capsule** (Hauteur 1.8m, Rayon 0.4m).
3. Placer une caméra en enfant de l'objet (au niveau des yeux).
4. Coder la rotation souris (Yaw pour le corps, Pitch pour la caméra).

Étape B : Vélocité et Inputs

1. Récupérer les touches ZQSD (ou WASD).
2. Construire un vecteur de direction local $(x, 0, z)$.
3. Le transformer en **Direction Monde** (pour qu'avancer se fasse dans la direction du regard).

$$\vec{d}_{\text{monde}} = \text{Rotation}_y \times \vec{d}_{\text{local}}$$

4. Appliquer une vitesse : $\vec{v} = \vec{d}_{\text{monde}} \times \text{Speed}$.

Étape C : Intégration et Gravité

1. Créer une variable persistante `velocityY` (float).
2. À chaque frame, diminuer cette valeur : `velocityY -= 9.81 * dt`.
3. Combiner avec le mouvement horizontal : `finalVelocity = (vx, velocityY, vz)`.
4. Déplacer le corps : `position += finalVelocity * dt`. **(À ce stade, le joueur traverse le sol et tombe à l'infini).**

Étape D : Gestion des Collisions (Le cœur du TP)

Utiliser le `KinematicCharacterController` de Rapier (ou coder le Raycast manuel si moteur maison).

1. Configurer le Shapecast vers le bas avec un offset de peau ("Skin width").
2. Si collision détectée au sol : `isGrounded = true` et `velocityY = 0`.
3. Si collision latérale détectée (mur) : Appliquer la formule du glissement vue en section 2 pour corriger `vx` et `vz`.

Étape E : Le Saut

1. Si `Input.Space` et `isGrounded` :

$$\text{velocityY} = \sqrt{2 \times \text{HauteurSaut} \times \text{Gravité}}$$

(Formule physique pour atteindre une hauteur précise).