

Session 11 : Moteurs Physiques & Middleware

💡 Introduction

Félicitations ! Vous avez écrit votre propre moteur physique basé sur Euler puis Verlet. Vous comprenez maintenant la complexité cachée derrière de simples balles qui rebondissent : intégration, détection (Broad/Narrow phase) et résolution.

Dans l'industrie (Jeux Vidéo, Cinéma, Simulation), nous utilisons rarement des moteurs faits maison (« Homebrew ») pour la physique complexe. Nous utilisons des **Middlewares**.

Cette session explore les géants du marché, leurs spécificités, et comment choisir le bon outil pour Raylib ou Three.js.

1. Concepts avancés des moteurs commerciaux

Avant de comparer les moteurs, il faut comprendre les problèmes qu'ils résolvent et que notre petit moteur Verlet ne gère pas encore.

Rigid Body vs Soft Body

- **Rigid Body (Corps Rigide)** : L'objet ne se déforme jamais. La distance entre deux atomes de l'objet reste constante. C'est 95% de la physique dans les jeux (caisses, personnages, voitures).
- **Soft Body (Corps Mou)** : L'objet se déforme (tissus, gelée, pneus). C'est ce que nous avons simulé avec les ressorts et Verlet ! C'est beaucoup plus coûteux en calcul (CPU).

⚠ Le problème du Tunneling (CCD)

Dans notre moteur, si une balle va très vite, elle peut traverser un mur entre deux frames car P_{old} est d'un côté et P_{new} de l'autre.

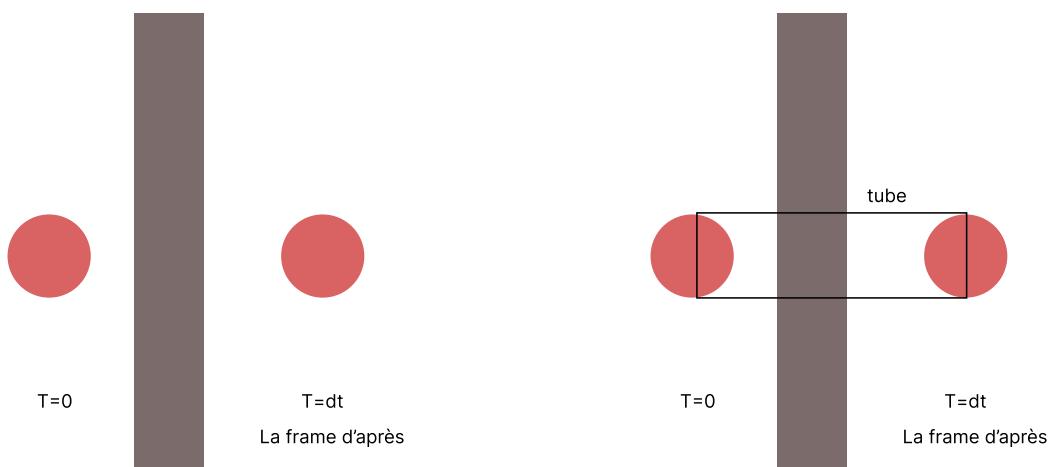


Fig. 1. – CCD (Continuous Collision Detection) pour éviter le tunneling

Les moteurs commerciaux utilisent le **CCD (Continuous Collision Detection)**. Au lieu de regarder des positions discrètes, ils extrudent le volume de l'objet le long de son vecteur vitesse pour vérifier s'il a « balayé » un obstacle.

Sleeping (L'endormissement)

Pour optimiser, un moteur physique détecte quand un objet arrête de bouger (vitesse < seuil pendant x frames). Il le met en état de **Sleep**. L'objet est retiré des calculs d'intégration jusqu'à ce qu'une force extérieure le réveille.

Sans cela, un empilement de 1000 caisses consommerait 100% du CPU même immobile.

2. Les Géants du C++ (Industrie AAA)

Ces moteurs sont les standards de l'industrie pour les jeux PC et Consoles.

Moteur	Spécificités
PhysX (NVIDIA)	<ul style="list-style-type: none"> Le standard actuel (utilisé par Unity et Unreal Engine jusqu'à récemment). Très rapide, très stable. Spécificité : Peut utiliser la carte graphique (GPU) pour les particules et fluides complexes.
Havok	<ul style="list-style-type: none"> Le vétéran (Half-Life 2, Zelda Breath of the Wild). Incroyablement robuste pour les Character Controllers (mouvements de personnages). Très cher (licence propriétaire historique), mais performant.
Jolt Physics	<ul style="list-style-type: none"> La nouvelle star (Open Source). Utilisé dans Horizon Forbidden West. Spécificité : Conçu pour le Multithreading massif. Il écrase les autres sur les processeurs modernes à plusieurs coeurs.

3. Le Roi de la 2D : Box2D

Box2D

Si vous avez joué à **Angry Birds**, **Limbo** ou **Shovel Knight**, vous avez utilisé **Box2D**

Créé par Erin Catto (Blizzard), c'est un moteur « Impulse Based » (comme notre résolution de collision, mais plus mathématique avec des matrices de masse).

C'est la référence absolue pour la 2D. Il est déterministe, stable et gère parfaitement les « stacks » (piles d'objets).

4. Physique pour le Web (Three.js / Javascript)

Puisque nous travaillons parfois avec le Web, le choix du moteur est critique car le JavaScript est moins performant que le C++.

Moteur	Techno	Verdict
Cannon.js	Pur JS	<ul style="list-style-type: none"> Facile à utiliser avec Three.js. Lent si > 100 objets. Abandonné, mais fork « Cannon-ES » actif.
Ammo.js	WASM (C++)	<ul style="list-style-type: none"> C'est le moteur « Bullet » compilé en WebAssembly. Très complet mais lourd à charger. API complexe et documentation difficile.
Rapier	Rust -> WASM	<ul style="list-style-type: none"> Le choix moderne. Écrit en Rust, compilé pour le web. Extrêmement rapide et stable. API moderne et bien documentée.

💡 Pourquoi Rapier ?

Si vous devez choisir un moteur aujourd’hui pour un projet Web/Three.js, choisissez Rapier. Sa structure de données (SIMD) permet de gérer des milliers d’objets sans ralentir le navigateur, là où Cannon.js s’effondrerait.

5. Déterminisme vs « Gameplay »

💡 Le Déterminisme

Une simulation est dite déterministe si : « **Pour un état initial donné et un delta time fixe, le résultat est strictement identique bit-à-bit sur n’importe quel ordinateur.** »

C'est vital pour :

1. Le **Multijoueur** (Pour que tous les joueurs voient la même caisse tomber au même endroit).
2. Les **Replays** (Pour rejouer une partie enregistrée).

Les flottants (float) ne sont pas gérés pareil par tous les processeurs (Intel vs AMD vs ARM). Les moteurs comme **Box2D** ou des versions « Fixed Point » s’efforcent d’être déterministes.

Conclusion : Faut-il faire son propre moteur ?

- **OUI** pour apprendre (ce cours).
- **OUI** pour des cas très simples (Pong, particules visuelles sans gameplay).
- **OUI** pour des cas très spécifiques (Simulation de cordes pure, simulation de sable).

- **NON** pour un jeu commercial complexe. La gestion des collisions 3D (Mesh vs Mesh), l'optimisation spatiale (Octrees/BVH) et la stabilité numérique sont des années de travail que PhysX ou Rapier vous offrent gratuitement.

6. Anatomie d'un Moteur : Architecture ECS

💡 Pourquoi cette complexité ?

Dans notre moteur Verlet, nous avions un `vector<Ball>`. C'est simple, mais ça passe mal à l'échelle (cache CPU, mémoire).

Les moteurs modernes (Rapier, Jolt, Unity ECS) séparent strictement les **Données de la Logique**.

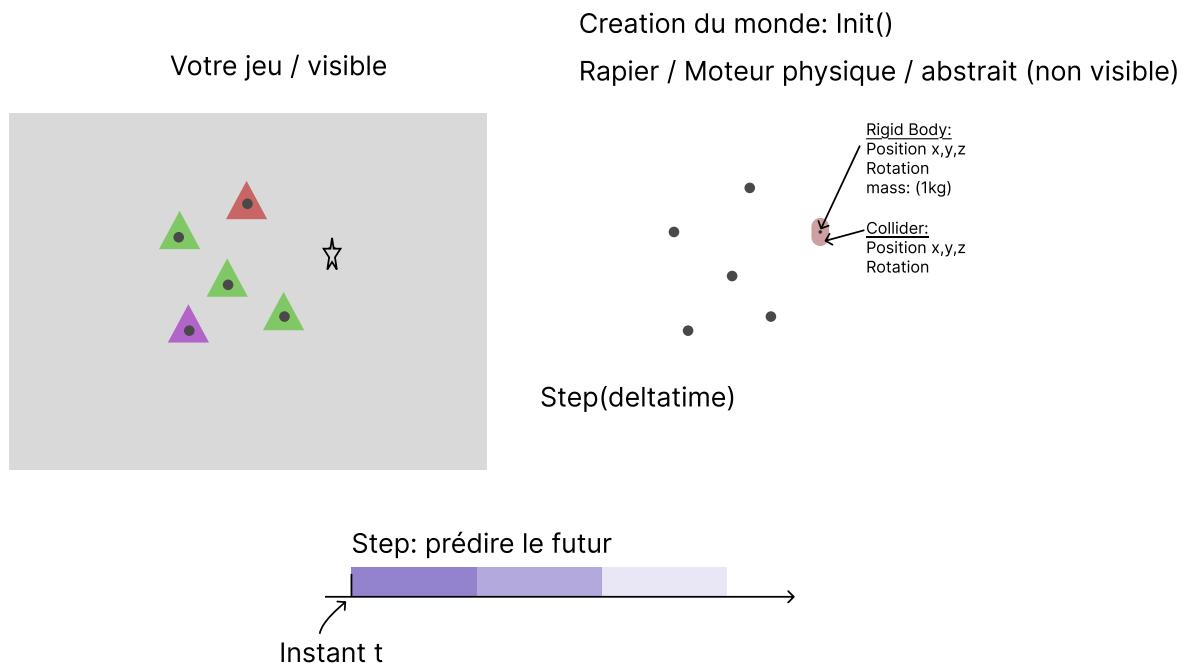


Fig. 2. – Séparation des données et de la logique, un seul objectif: prédire le futur

Pour être efficace, ils utilisent une architecture basée sur les **Entités, Composants et Systèmes** (ECS). En effet, ces architectures **permettent d'optimiser et paralléliser** les tâches répétitives simples sur de nombreux objets (comme le calcul de la physique).

A. Les Structures de Données (The Sets)

Au lieu d'avoir un objet `Ball` qui contient tout (Position, Forme, Masse), Rapier découpe les données dans des conteneurs spécialisés appelés Sets. On n'accède pas aux objets par pointeur, mais par Handle (un ID unique).

- RigidBody Set : Stocke la position (x, y, z), la vitesse et les forces. C'est « l'âme » physique de l'objet.
- Collider Set : Stocke la forme géométrique (Cube, Sphère) et les matériaux (Friction, Restitution). C'est le « corps » tangible.
- Joint Set : Stocke les contraintes (Ressorts, Pivots) qui relient deux RigidBodies.

Generational Arena

Ces « Sets » ne sont pas de simples tableaux. Ce sont des **Generational Arenas**. Si vous supprimez l'objet ID 42 et en créez un nouveau qui prend la place 42, le moteur ajoute une « génération » (ex: `42_v2`). Cela empêche les bugs où une balle continuerait de suivre un objet détruit.

B. Le Pipeline Physique

La fonction `step()` n'est pas magique. Elle orchestre plusieurs étapes critiques séquentiellement :

1. **Gravity & Forces** : Application de la gravité globale et des forces utilisateur.
2. **Broad Phase** : Trouve grossièrement qui **pourrait** toucher qui (AABB).
3. **Narrow Phase** : Calcule précisément les points de contact.
4. **Island Manager (Optimization)** : Le moteur regroupe les objets qui se touchent en « îles ».
 - Si tous les objets d'une île ont une vitesse quasi-nulle, l'île entière est mise en Sleep (Sommeil).
 - Le moteur cesse de les calculer jusqu'à ce qu'une force extérieure les réveille.
 - **C'est le secret pour afficher 10 000 caisses à 60 FPS.**
5. **Solver** : Résout les contraintes (empêcher la pénétration, gérer les joints).
6. **Integration** : Met à jour les positions finales (Euler: $P = P + V \cdot dt$).

C. Les Pipelines Spécialisés

Tous les calculs ne servent pas à faire bouger des objets. Parfois, on veut juste poser une question au monde.

Query Pipeline (Interrogation)

C'est le module utilisé pour le Gameplay (comme notre tir de catapulte). Il utilise les structures accélérées (BVH - Bounding Volume Hierarchy) pour répondre instantanément à :

- Raycasting : « Qu'est-ce qui est sous ma souris ? »
- Shapescasting : « Si j'avance ce personnage ici, va-t-il cogner un mur ? »
- Point Projection : « Quel est l'objet le plus proche de cette bombe ? »

D. Événements et Hooks

Parfois, la physique ne suffit pas, il faut du code de jeu.

- Physics Hooks : Permet de filtrer les collisions **avant** qu'elles ne soient résolues (ex: « Les gentils traversent les gentils, mais cognent les méchants »).
- Event Handler : Notifie le jeu **après** une collision (ex: « Si la balle touche le mur à haute vitesse -> Jouer un son "Boom" »).