Journal de Bord
Projet de Développement Informatique

Création d’un moteur physique pour les jeux vidéos

Projet réalisé dans le cadre du module 8INF935 - Mathématiques et physique pour le jeu vidéo
Par Hippolyte COSSERAT, Emeline GÉHANNO, Florian HEME et Dimitri JASICA

Table des matières

I - INTRODUCTION :

L'objectif global de ce projet est de créer un moteur physique capable de gérer des objets solides implémentant un système de gestion de collision pour les jeux vidéo de manière itérative en 4 phases distinctes.
Ce journal a pour but de référencer les différentes difficultés, les astuces de programmation utilisées ainsi que les choix effectués lors de ce projet.

II - SPRINT 1 :

A - DIFFICULTÉS :

Problème 1 : Installation de visual studio sur ordinateur personnel :
→ Impossible de lancer le logiciel dû à des soucis de build : résolu à l'aide de Lucas Gonzalez

Problème 2 : Impossibilité d'utiliser une fonction main dans les fichiers de tests unitaires.
→ Elaboration d'un fichier .h pour les tests. Ainsi nous avons pu appeler les fonctions de test dans les fichiers ofApp.

B - ASTUCES DE PROGRAMMATION :

Astuce 1 : Mettre en place une classe *Projectile* qui permet de gérer l'affichage de *Particle*
Astuce 2 : Attribution de fichiers sur lesquels travailler à chaque membre du groupe pour éviter tout conflits (en plus des protections de Git).

C - JUSTIFICATIONS DES CHOIX :

Pour le choix de l'intégrateur, nous avons choisi d'implémenter la méthode d'Euler. L'intégration de Verlet est plus précise, mais seulement si l'on garde en mémoire la position précédente de chaque particule, ce qui multiplie la complexité spatiale. La méthode d'Euler est donc un meilleur compromis entre précision et coût, et les trajectoires obtenues nous semblent finalement très réalistes. Une classe Particle2 qui implémente l'intégrateur de Verlet en échange d'un plus grand coût spatial pourra être créée si nécessaire.

D - CONCLUSION :

Améliorer l'interface utilisateur, notamment mettre en place des boutons pour le choix des projectiles (en ajoutant l'addon *ofxGui* par exemple).

Choisir les bons paramètres pour les différents projectiles afin que ça soit plus réaliste.

E - ANNEXE :

Tableau de bord du sprint 1 :

TABLEAU DE BORD

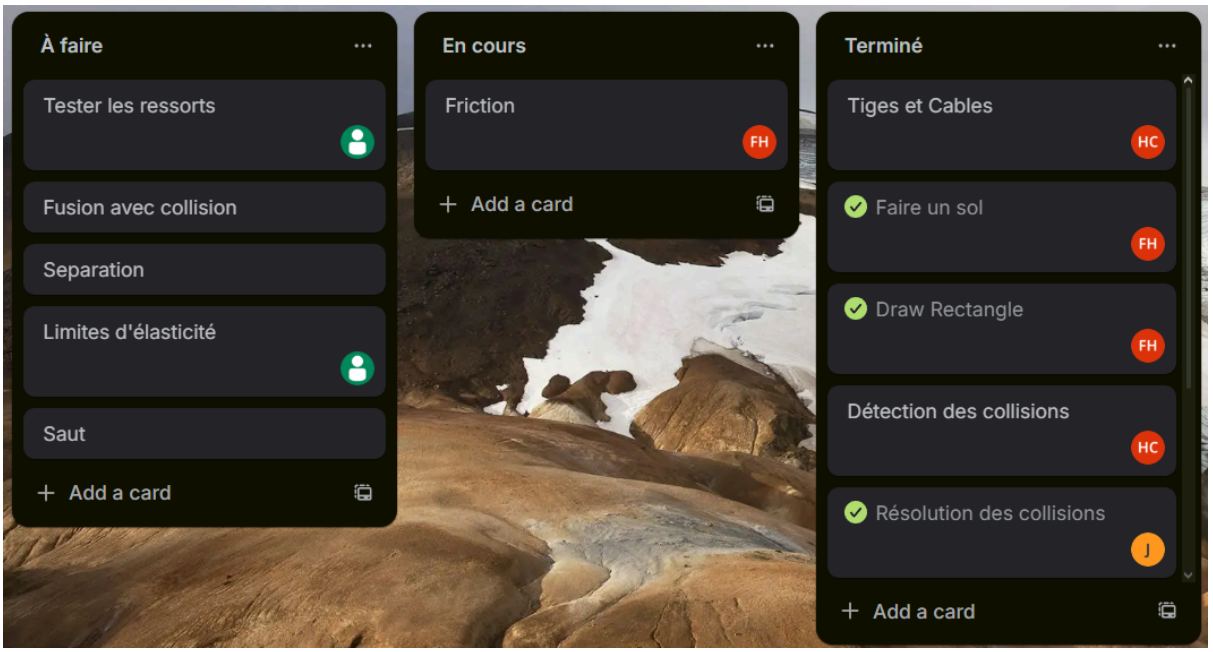
Membre(s) du groupe	Tâche effectuée	Date	Temps de travail (en heure)
Hippolyte	Gestion de la physique du moteur	10/09/2025	1
Hippolyte et Florian	Implémentation des Vecteurs	10/09/2025	1
Dimitri	Création README	10/09/2025	1
Emeline & Florian	Gestion de l'affichage	17/09/2025	4
Florian	Tests unitaires	17/09/2025	2
Dimitri	Création ppt présentation	17/09/2025	1

II - SPRINT 2 :

A - DIFFICULTÉS :

Problème 1 : Organisation :
→ Ce sprint étant plus dense que le premier et avec une semaine de relâche, il fallait bien s'organiser pour pouvoir travailler efficacement et correctement.

SOLUTION : Mise en place d'un rendez-vous hebdomadaire en dehors des cours afin de faire le point et d'avancer sur le projet. Mise en place également d'un trello listant et assignant les tâches à réaliser pour le sprint.



Problème 2 : Codage et fusion des fonctionnalités :
→ Ce sprint s'est avéré plus complexe fonctionnellement que le précédent, nécessitant l'implémentation de nombreuses fonctionnalités, elles-mêmes plus élaborées. Les tâches avaient été réparties entre les membres du groupe, et nous avions constaté que chaque fonctionnalité (gestion des forces, du blob, détection et résolution des collisions, etc.) était opérationnelle individuellement. Cependant, une fois fusionnées, l'ensemble du moteur présentait finalement des bugs difficiles à identifier et à corriger, notamment au niveau de la gestion des collisions.

SOLUTION : Brainstorming, travaux et séances intensives de débogguage en groupe afin de trouver les sources des différents problèmes. Nous avons également bien réorganisé l'architecture de notre projet afin de bien différencier chaques fonctionnalités et de ne pas tout regrouper au même endroit.

Problème 3 : ordre de la boucle de mise à jour
→ L'ordre d'exécution des composants du moteur physique est crucial pour le comportement des objets, notamment dans la gestion des forces, des collisions et du mouvement des particules. Par le passé, l'actualisation des forces, suivie des positions, puis la détection des collisions, s'est avérée inefficace lors de l'intégration d'impulsions dans la résolution des collisions.

SOLUTION : L'ordre optimal d'exécution des composants du moteur physique (application des forces, mise à jour des accélérations et des vitesses, détection et résolution des collisions, puis mise à jour finale des positions) n'a été découvert que très tardivement. Cette difficulté était due à l'utilisation, lors du sprint précédent, d'un intégrateur qui mettait à jour la position et la vitesse simultanément.

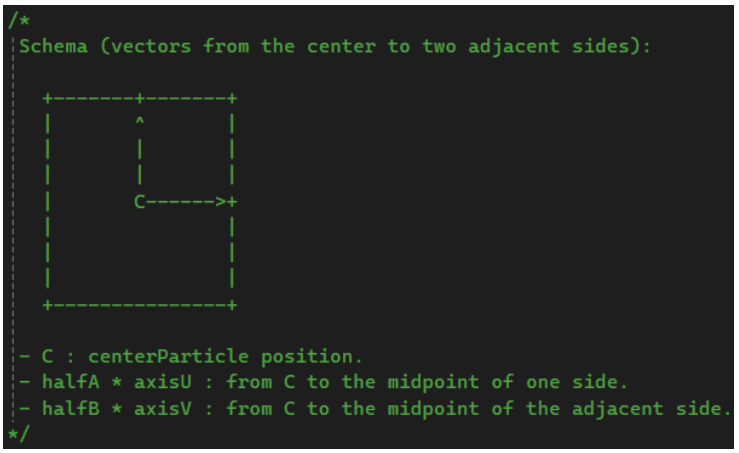
B - ASTUCES DE PROGRAMMATION :

Astuce 1 : Mise en place de la classe abstraite de base *Actor*
Astuce 2 : Création de classes distinctes pour chaque forme géométrique (*Circle*, *Rect*, *Blob*)
Astuce 3 : Mise en place de la classe *World*

C - JUSTIFICATIONS DES CHOIX :

Actor sert de classe de base pour toutes les entités physiques (cercles, rectangles, blob). Cela permet de traiter toutes les entités de manière uniforme simplifiant les boucles de mise à jour et de dessin (polymorphisme). Elle permet également de centraliser les données et méthodes communes à tous les objets physiques : *centerParticle*, *getShape()*, *collidesWith()*.
Chaque classe pour les formes géométriques, qui hérite de *Actor*, est responsable des données liées à sa forme :

- *Circle* : *radius*
- *Rect* : *axisU*, *axisV*, *halfA*, *halfB*



- *Blob* : *circles* (liste des particules non centrales)

Et également les logiques liées à sa forme, notamment pour les collisions.
World est le conteneur principal et le moteur de la simulation physique. Elle gère le cycle de vie de tous les objets, forces et contraintes. Elle centralise la logique en ajoutant les forces, en appliquant les mises à jour, en intégrant les vitesses et positions et en réinitialisant les forces. *World* permet également l'initialisation de la scène.

D - CONCLUSION :

Ce sprint a été beaucoup plus complexe sur les points de vues organisationnel et technique. Malgré cela nous avons su trouver des solutions pour pallier ces problèmes en s'appuyant sur les points forts de chacun des membres du groupe. Il faudra garder ce rythme et cette organisation pour les deux derniers sprint de ce projet afin de ne pas se laisser surpasser et de répondre au mieux aux attentes.

E - ANNEXE :

Tableau de bord du sprint 2 :

TABLEAU DE BORD_2

Membre(s) du groupe	Tâche effectuée	Date	Temps de travail (en heure)
Florian	Force de friction	15/10/2025	3
Emeline	Force ressort	12/10/2025	2
Emeline & Hippolyte	HUD	10/10/2025	3
Florian	World	14/10/2025	2
Dimitri	Gestion des collisions	10/10/2025	4
Hippolyte	Détection des collisions	09/10/2025	2
Emeline	Blop & déplacement	13/10/2025	5
Hippolyte	Tiges et cables	20/10/2025	2
Florian & Hippolyte & Dimitri	Tests et résolutions des forces & collisions	20/10/2025	5
Hippolyte	Réorganisation de l'architechture du projet	10/10/2025	1
TOTAL TEMPS	30		

III - Sprint 3 :

A - DIFFICULTÉS :

Problème 1 : Compréhension de l'utilisation des nouveaux outils mathématiques
→ Le nouveau concept de quaternions a été pour nous un blocage au niveau de la compréhension de son utilisation. Le cours étant plus dense sur ces nouvelles notions, il était un peu compliqué de retirer ce qui nous était vraiment utile.

SOLUTION : Poser des questions au professeur pour avoir plus d'informations. De plus, nous nous sommes réunis afin de retravailler le cours ensemble ainsi que de reprendre les parties de code *Matrix3*, *Matrix4* et *Quaternions* pour que tout le monde soit en accord avec.

Problème 2 : Outil collaboratif hors service
→ Des pannes sur git nous ont empêché de mettre en commun notre travail aux horaires que nous avions réservés pour travailler tous ensemble. De plus, le fait que la majorité du travail à effectuer est centré autour de la classe corps rigide, il a été difficile de répartir le travail sans outils collaboratifs appropriés.

SOLUTION : Nous nous sommes regroupés dans une salle pour travailler et nous avons tenté de travailler ensemble sur un même poste, nous avons également mis en commun nos fichier en nous les transférant directement via USB.

B - ASTUCES DE PROGRAMMATION :

Astuce 1 : Création de classes distinctes pour chaque forme géométrique.
Astuce 2 : Pré-calcul du tenseur d'inertie inverse en espace local.

C - JUSTIFICATIONS DES CHOIX :

Le fait de créer une classe par forme, tout en héritant de la classe *RigidBody*, nous permet d'ajouter des attributs propre à chaque forme, notamment pour les dimensions, et de mieux gérer le rendu visuel :

- *Box* : *dimensions* qui est vecteur avec largeur, hauteur, profondeur
- *Cylinder* : *radius* et *height*
- *Axe* : *handleDimensions* et *headDimensions*

Nous stockons deux matrices pour l'inertie : *invInertiaTensorBody* (constante) et *invInertiaTensor* (variable). Cela nous permet de créer le torseur d'inertie qu'une seule fois, leur de la création de l'objet, puis nous appliquons une transformation afin d'obtenir l'inertie dans le monde.

D - CONCLUSION :

Ce sprint a été nettement plus complexe sur le plan de la compréhension des notions mathématiques. Malgré cela, nous nous en sommes bien débrouillés dans l'implémentation et l'utilisation des fonctions, en nous appuyant sur les points forts de chacun des membres du groupe. De plus, en nous basant sur nos erreurs précédentes, nous avons su pallier les problèmes d'organisation.

E - ANNEXE :

Phase 3

Membre(s) du groupe	Tâche effectuée	Date	Temps de travail (en heure)
Dimitri	Matrix3	06/11/2025	2
Dimitri	Matrix4	06/11/2025	3
Emeline	Quaternions	10/11/2025	2
Florian	Tests unitaires	10/10/2025	3
Hippolyte	CorpsRigides (attributs et méthodes pertinentes)	14/10/2025	4
Hippolyte	CorpsRigides (intégrateur physique)	10/10/2025	2
Emeline	Rendu visuel	15/11/2025	3
Emeline	Logique de tir	15/11/2025	2
Hippolyte, Florian, Emeline	Debuggage	18/11/2025	3
TOTAL TEMPS	**24**		