



# Journal de développement

Arthur Pejoiane

Corentin Chrisment

Pier Chapon

Romain Malka

## 1 Phase 1

Le but de cette phase est de créer un moteur physique simple de gestion de particules à partir des notions vues aux semaines 1 à 4.

- ☒ Implémenter une classe Vecteur3D avec toutes les méthodes appropriées: norme, norme au carré, normalisation, multiplication par un scalaire, addition, soustraction, produit par composantes, produit scalaire, produit vectoriel, etc.
- ☒ Implémenter une classe de test ou des test unitaires pour la classe Vecteur3D.
- ☒ Implémenter une classe Particule comprenant des accesseurs pour l'attribut InverseMasse.
- ☒ Implémenter un intégrateur (intégration d'Euler et/ou intégration de Verlet) afin de mettre à jour la position et la vitesse de chaque particule à chaque frame.
- ☒ Réaliser un petit jeu de tir balistique où on peut choisir entre différents projectiles (e.g. balles, boulets, laser, boule de feu) et effectuer un tir. Chaque projectile doit avoir sa propre vitesse initiale et sa propre masse. Le frottement doit être négligeable (près de 1). La trajectoire des projectiles doit être clairement visible à l'écran.



## Semaine 1-4

Lors de cette première phase de développement, nous avons réussi à implémenter **les principales exigences mathématiques** sans rencontrer de difficultés majeures. Nous avons opté pour une classe **Vector3** sous forme de **template**, ce qui nous offre une plus grande **flexibilité** quant au type de données utilisé. Bien que les flottants soient notre principal type, cette approche nous permet d'anticiper d'éventuels changements de type au cours du projet.

Étant donné que nous n'avons pas eu de problèmes particuliers au niveau mathématique, nous en avons profité pour faire l'intégration de **Verlet** en plus de celle d'**Euler** afin de constater les variations entre les deux intégrations vues en cours.







Pour l'affichage des éléments à l'écran, nous nous appuyons sur le toolkit **Openframeworks**. Ce dernier nous permet non seulement de gérer l'affichage, mais aussi d'utiliser des fonctionnalités préexistantes comme la **gestion du taux de rafraîchissement**. Cependant, nous avons rencontré un **problème de synchronisation** avec le contrôle de version lors de son intégration. Nous envisageons donc d'explorer d'autres solutions de génération de projets pour fluidifier notre workflow.

Nous avons aussi développé une classe **Object** afin de simplifier l'affichage des images dans les prochaines phases.

À l'avenir, nous prévoyons également de refactoriser certaines de nos implémentations, en particulier dans le cadre de l'intégration des **corps rigides**.

## Phase 2

Le but de cette phase est de poursuivre le développement du moteur physique réalisé lors de la phase 1 en lui ajoutant les notions vues au chapitres 4, 5 et 6. Ainsi, le moteur permettra la gestion d'amas de particules.

-  Implémenter, à l'aide d'une interface, des générateurs pour les forces suivantes : gravité, friction et ressorts. Notez que tous les types de ressorts vus en classe doivent être implémentés.
-  Implémenter un système de détection de collisions. Ce système devra détecter les types de collisions suivants : interpénétrations, contacts au repos, tiges et câbles.
-  Implémenter un système de résolution de collisions basé sur les impulsions.
-  Réaliser un petit jeu où l'on déplace un amas de particule, i.e. *blob*. Chaque particule est représentée par un cercle. Les particules sont reliées entre elles en utilisant des ressorts classiques. Par contre, ces liens ont une limite d'élasticité au-delà de laquelle ils se comportent comme des câbles.
-  Votre blob doit pouvoir se séparer en plusieurs morceaux et se fusionner.
-  Ajouter un HUD où l'on affiche en temps réel le nombre de particules attachées au Blob. Chaque nouvelle valeur rebondit avec un mouvement harmonique simple amorti lorsqu'elle apparaît.

## Semaine 4-6

Nous avons suivi l'ordre des exigences établies pour accomplir les différentes tâches. L'implémentation des interfaces s'est déroulée sans grande difficulté. Cependant, l'intégration des ressorts, en particulier la gestion des collisions, a été plus complexe.






Concernant les ressorts, nous avons créé deux classes distinctes : **ParticleSpring** pour les ressorts reliant deux particules et **ParticleSpringFixedPoint** pour les ressorts entre une particule et un point fixe.

Afin de gérer efficacement nos nouveaux systèmes, nous avons créé une classe **World**. Cette classe est responsable de l'initialisation et de l'exécution en boucle des fonctions clés de nos différents systèmes.

Pour la gestion des collisions, nous avons développé une classe **SphereCollider** qui évalue si deux sphères se touchent via une classe **CollisionSystem**. Toutefois, nous rencontrons encore un problème d'adhérence anormale entre certaines sphères, donnant un effet de "glue", et nous n'avons pas encore trouvé de solution satisfaisante. Après refactorisation, le problème persiste. Nous avons tout de même implémenté des **callbacks**. Ces derniers permettent d'associer des fonctions spécifiques lors des collisions entre particules. Utile lors de la mise en place du jeu.

### Phase 3

Le but de cette phase est de spécialiser le moteur physique des phases 1 et 2 par l'ajout de la physique rotationnelle, implémentant ainsi les notions vues au chapitres 7, 8 et 9. Ce faisant, le moteur permettra la gestion des corps rigides. Notez toutefois que la gestion des collisions n'est pas à implémenter dans cette phase. En particulier, vous devez :

-  Implémenter une classe `CorpsRigide` avec les attributs et méthodes pertinentes.
-  Implémenter les classes `Matrix3`, `Matrix4` et `Quaternion` avec les attributs et méthodes pertinentes.
-  Implémenter une classe test ou des tests unitaires pour les classes `Matrix3`, `Matrix4` et `Quaternion`.
-  Implémenter un intégrateur physique complet permettant la mise à jour des objets de type `CorpsRigide`.
-  Réaliser un jeu de tir balistique 3D où des objets non sphériques (e.g. boîtes) sont lancés en l'air en leur appliquant une force ailleurs que sur leur centre de masse. Le centre de masse ainsi que le mouvement de rotation doivent être clairement visibles.

### Semaine 9-11

Dans cette troisième phase de développement, nous avons commencé par intégrer les **matrices** et les **quaternions**. Cette étape nous a permis de réaliser des **tests unitaires** pour valider nos implémentations avant de passer aux calculs physiques. Nous avons choisi d'implémenter des matrices sous une forme **NxM dynamique**, ce qui nous dispense de créer plusieurs classes **Matrix** pour chaque dimension spécifique. Cette approche améliore la flexibilité et simplifie la gestion des différentes tailles de matrices.

Ensuite, nous nous sommes concentrés sur l'**intégrateur physique**, conçu pour gérer la rotation des **corps rigides** en utilisant nos implémentations des quaternions et des matrices. Cette étape n'a pas été sans difficulté : nous avons rencontré un problème lié au centre de masse des corps rigides, qui provoquait une rotation anormale. Après une analyse approfondie, nous avons réussi à résoudre ce dysfonctionnement.

Pour répondre aux exigences manquées des phases précédentes, nous avons activé l'option **GUI** d'OpenFrameworks afin de développer une interface utilisateur pour la démonstration du jeu. L'interface est désormais plus complète : elle permet de modifier la masse du projectile, l'angle du canon et la puissance du tir. De plus, nous avons ajouté une fonctionnalité de rotation de la caméra 3D, permettant d'observer la rotation des cubes sous différents angles.