Célestin Moënne-Loccoz
Eva Minot
Nicolas Cavani

Journal de bord - Phase 2

1. Démarrage de la phase

Lors de la première séance de cours dédiée au démarrage de la deuxième phase du projet, nous nous sommes penchés tous les trois sur la déclaration des classes ParticleForceGenerator et ParticleForceRegistry, qui sont les deux classes centrales pour l'implémentation et l'utilisation des générateurs de forces. Le but était que nous partions tous avec une même compréhension du projet et une même base pour les générateurs. Puis, nous nous sommes réparti le travail:

- Célestin et Eva s'occupent des définitions de ParticleForceGenerator et ParticleForceRegistry
- Nicolas s'occupe de travailler sur l'interface graphique

Et, pour une répartition équitable des tâches, nous nous sommes chacun attribués deux générateurs de force à créer:

- Nicolas s'occupe de ParticleGravity (gravité) et ParticleSpring (ressort classique entre 2 particules)
- Eva s'occupe de ParticleDrag (traînée) et ParticleBungeeSpring
- Célestin s'occupe de ParticleBuoyancy (flottaison) et ParticleAnchoredSpring

2. Implémentation des générateurs de force

Comme expliqué dans la partie 1, nous nous sommes répartis les tâches pour cette partie de la phase. Au niveau de la déclaration de ParticleForceGenerator et ParticleForceRegistry, nous n'avons pas rencontré de difficulté particulière, le cours étant bien expliqué pour ces classes. Pour la création des différents générateurs de forces, aucune difficulté particulière n'a été rencontrée. Eva et Célestin se sont occupés des leurs rapidement, ce qui leur a permis de commencer à se pencher sur l'implémentation des contacts en attendant que Nicolas finisse les siens et continue l'interface graphique.

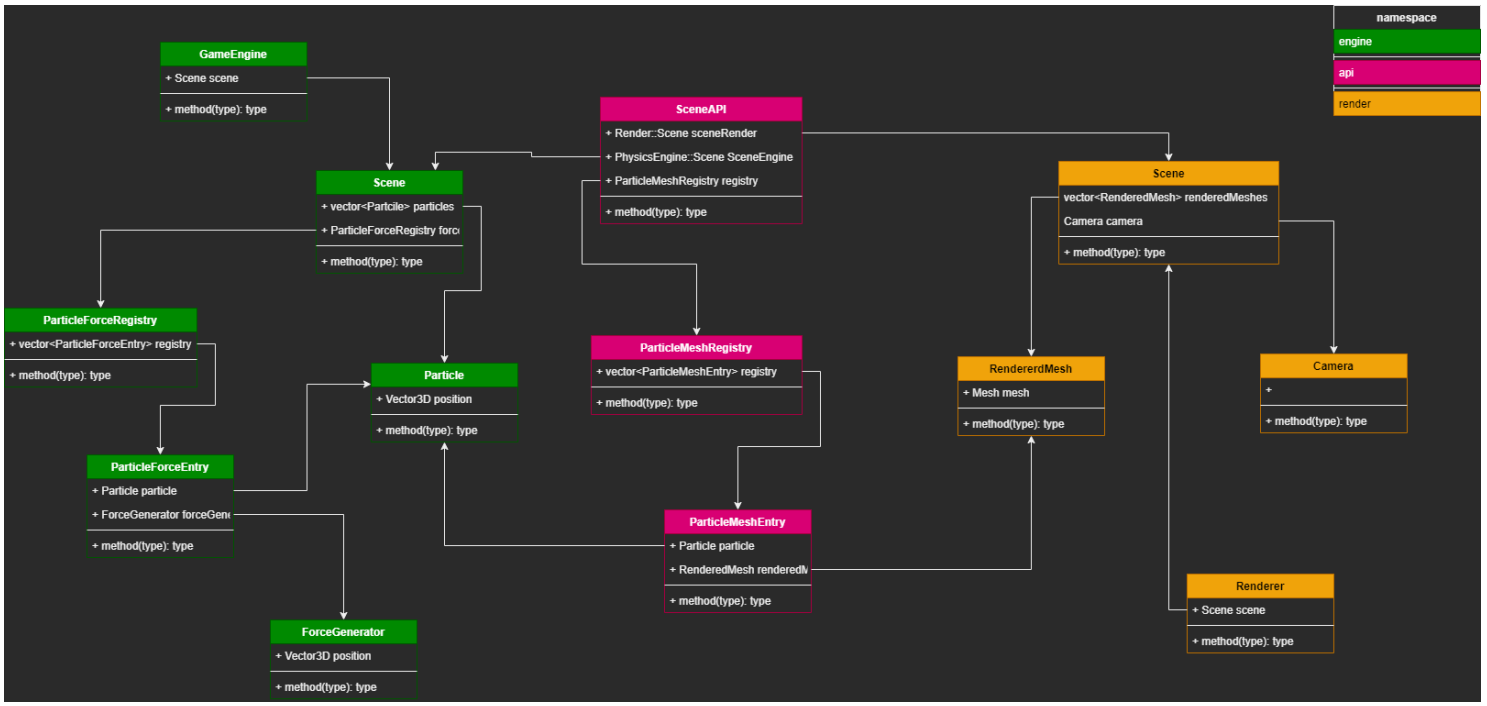
3. Implémentation des contacts

Eva et Célestin se sont occupés de cette partie, car Nicolas avait déjà beaucoup à faire au niveau de l'API. Après la séance de cours sur les contacts, nous avons commencé à implémenter le générateur de contact (ParticleContactGenerator) et le solveur de contact (ParticleContactResolver), en s'inspirant du cours. Nous n'avons pas rencontré de difficulté particulière ici. Ensuite, nous avons implémenté la classe ParticleContact, qui vérifie l'interpénétration et qui calcule les vélocités résultantes. Encore une fois, pas de grande difficulté rencontrée car les formules sont bien détaillées dans le cours.

Une fois toute la base pour créer les contacts achevée, nous avons créé les différents générateurs de contact demandés: le câble (ParticleCable), la tige(ParticleRod) et le mur (WallContactGenerator). Cette partie a été plus compliquée, notamment au niveau des tests. C'est surtout le WallContactGenerator qui nous a posé problème parce qu'il fallait ajouter les contacts au repos que l'on n'avait pas encore implémentés, et nous avons eu un peu de mal à trouver une solution visuellement et logiquement satisfaisante. Au final, dans ParticleContact dans la méthode resolveVelocity, nous vérifions si l'accélération de la particule est uniquement dûe à la gravité: si c'est le cas, on vérifie si la vélocité résultante du contact que l'on calcule est inférieure à un seuil (que l'on a défini à 8*m_restitution, valeur qui nous paraissait satisfaisante car elle permet quelques rebonds avant de stopper la particule en fonction du coefficient de restitution). Si c'est le cas, on considère que la particule fait de trop "petits" rebonds et qu'elle doit donc s'arrêter de rebondir, donc on la met au repos. Sinon, on la laisse rebondir. Lorsqu'on met une particule en état de repos, on lui ne lui applique plus la gravité pour qu'elle n'interpénètre plus la surface. Si une autre force que la gravité s'applique, à un moment donné, à une particule en repos, celle-ci sort de son état de repos et "répond" à la force qu'elle reçoit (et on lui réapplique la gravité).

Enfin, nous avons pu nous atteler à la création de CableSpring, un générateur de force/contact hybride, qui se comporte comme un ressort classique entre 2 particules, mais qui génère un contact de type câble lorsqu'une limite d'élasticité préétablie est dépassée. Pour ce CableSpring, nous avons eu un peu plus de difficultés car il a fallu adapter notre code précédent pour permettre à un générateur de force de pouvoir également générer des contacts. Mais une fois cela fait, nous avons pu le tester et vérifier son bon fonctionnement, puis nous avons créé le blob.

4. Création de l’API



Concernant l’api entre l’engine et l'affichage, voici une capture du diagramme UML d’une partie de notre code. Nous voulons conserver un code cpp propre, générique, adaptable et réutilisable. C’est pourquoi nous avons séparé le code dans différentes libraires chacune dans leur propre namespace. Cela nous permet d’avoir une représentation d’une scène pour la partie engine qui contient les Particle et d’une scène pour la partie graphique qui contient les RenderedMesh que l’on affichent. A la manière du ParticleForceRegistry, nous avons un ParticleMeshRegistry qui permet de lier chaque Particle a un RenderedMesh. Nous avons aussi une classe SceneAPI, qui permet de faire le lien entre les deux scènes. Cette dernière est la scène que l’utilisateur peut utiliser pour ajouter des particules supplémentaires à la scène.

5. Interface utilisateur

Lorsque vous lancerez le programme mainPhase2.exe vous aurez la possibilité d'interagir avec la scène via ImGUI. Les boutons “Run Simulation” et “Pause Simulation” font se qu’ils indiquent en lançant ou mettant en pause la boucle de calculs dans la GameEngine. Vous devez appuyer au moins une fois sur “Run Simulation” pour que les particules se mettent en mouvement. Dans l’onglet “Blob Creation" vous aurez la possibilité de créer des blobs, de gérer le nombre de particules qui constituent le blob a créer et toutes les différentes constantes des ressorts et des forces qui s’appliquent sur les particules.
Une fois qu’au moins une particule est présente dans la scène, vous avez la possibilité de déplacer la première particule qui à été créée. Si celle-ci fait partie d’un blob (car vous avez généré un blob de plus d’une particule) le blob entier se déplacera, grâce aux forces qui lient les particules. Finalement en créant une seule particule (un blob a une seule particule) vous pourrez observer le mécanisme de particule au repos.

6. Reprise de la phase 1 et apprentissage de OpenGL et C++

Pour lancer notre projet vous avez le choix entre mainPhase1.exe et mainPhase2.exe.
Nous avons amélioré le programme de la phase1 qui était dernièrement seulement en console à cause du fait que l'apprentissage de OpenGL ne fut pas si simple, d'autant plus que je souhaitais réellement comprendre le fonctionnement bas niveau de la librairie. Nous avons donc abstrait dans des classes les concepts opengl de VertexBuffer, IndexBuffer, VertexArray, Mesh, Texture et Shader. Cela nous permet ensuite à l'aide d'une Window, d'un Renderer, d'une Camera et les RenderedMesh d'ajouter facilement des éléments à la scène et de les afficher. Pour le moment les particules que nous rendons sont des octaèdres avec une texture de flamme (initialement pour la fireball de la phase1, mais manque de temps ensuite pour gérer différentes textures par éléments). Nous planifions d'implémenter la possibilité d'importer un modèle 3D depuis blender, ce qui nous permettra de rendre les formes que nous souhaitons dans notre programme.
Concernant les différentes notions que l'on a pu apprendre en pratiquant, on retrouve l'utilisation des smart pointers. La distinction entre shared, unique et weak pointers. Nous avons aussi utilisé des namespaces nous permettant de déclarer deux Scene différentes, une pour l'engine et une pour le rendu graphique, pratique pour la compréhension et la cohérence du code. Le concept de matrice MVP, model view projection pour rendre des objets en 3D. De plus, grâce à notre architecture, il est tout à fait possible de gérer la position, la rotation et la scale de chacun des éléments que l'on affiche, vous constaterez que dans mainPhase1, la pyramide tourne sur elle même ! Finalement, toujours concernant la cohérence et la lisibilité de notre code, nous avons décidé d'utiliser clang format pour uniformiser le coding-style de tout le projet.

7. Ce qu'il reste à faire

Améliorations que nous souhaitons apporter :

- importation de mesh 3d générées depuis blender.
- gestion des textures pour les différents types de projectile.
- améliorer la création des projectiles pour que des forces cohérentes leur soient appliquées (Fireball, Bullet, et Laser n'ont plus de sens sans forces).
- abstraire la notion de blob pour regrouper les particules qui le compose permettant de créer et de déplacer des blobs facilement, ainsi que de pouvoir les fusionner et les diviser.
- améliorer le rendu de la grille au sol
- créer des murs avec une position et des dimensions, pour pouvoir les placer dans la scène et les afficher (actuellement le WallContactGenerator créer un contact sur un plan entier x, y ou z ; ce qui n'est pas intéressant dans le cadre de la création d'un jeu).
- pleins d'autres choses