Baptiste BABU
Louis CESARO
Jean FARINES
Yann ROYANT

Journal de bord
Phase 1

Astuces de programmation : Tests unitaires

Afin de nous assurer de la bonne implémentation de notre classe Vector3, nous avons implémenté une série de tests unitaires permettant de nous assurer de la fiabilité de toutes les méthodes touchant à notre classe Vector3.

Ainsi, les comportements associés à tous les opérateurs ont été testés afin de garantir la bonne exécution de nos programmes.

Difficultés rencontrées : Utilisation de SDL

Au cours de cette première phase, nous avons rencontré certaines difficultés à utiliser une interface graphique. Souhaitant faire fonctionner cette dernière sous Windows et plus particulièrement avec Visual Studio, nous avons cherché à compiler cette bibliothèque via un Cmake. Le problème concerne l'édition de lien entre notre exécutable et la bibliothèque.

Une solution partielle que nous avons pour l'instant utilisée, est de copier le fichier SDL2.dll dans le répertoire contenant notre exécutable à la main. Ce processus est automatisé dans nos règles de build.

Sous Linux, ce problème n'est pas rencontré car les librairies existent déjà sous Linux. Par conséquent, l'édition de lien de la librairie ne pose pas de problème.

Phase 2

Semaine 1:

Clarification de la Game Loop

Afin de rendre la boucle de jeu plus lisible dans notre code, nous avons implémenté des sous fonctions pour gérer l'initialisation de nos objets et de la fenêtre graphique.
Ainsi, l'initialisation de nos particules, au lancement du jeu, se fait via l'appel à la fonction *initParticles*.

Implémentation des forces et du registre

Au cours de la première semaine, nous avons implémenté l'interface ***ParticleForceGenerator*** ainsi que la classe ***GravityForceGenerator***. Dans cette dernière, le vecteur 3 de la gravité est défini de façon statique. Par conséquent, lors de l'instanciation de la force de gravité, la référence du vecteur de gravité est donnée à l'attribut au lieu d'instancier un Vector3 pour la gravité. Cela permet de limiter l'allocation de mémoire dans le cas d'un grand nombre de particules dans le jeu.

Parallèlement, nous avons implémenté la classe ***ForcesRegistery***. Afin de stocker l'ensemble de nos forces, nous nous sommes servis d'un ***vector*** de la bibliothèque standard. Par cette façon, nous avons à notre disposition un tableau dynamique dont la taille est recalculée lorsqu'on souhaite ajouter une nouvelle force au registre. Nous nous servons de la méthode ***push_back*** pour ajouter chaque nouvelle force à la fin de notre registre et nous n'avons pas le besoin de décaler les éléments déjà présent.

Semaine 2:

Implémentation des forces de Spring

Au cours de la seconde semaine, nous avons implémenté les forces relatives aux ressorts (***SpringForce***, ***AnchoredSpringForce***, ***BungeeSpringForce***, ***StiffSpringForce***, ***BuoyancyForce***). Pour chaque force, les particules sont passées par référence. Ainsi, nous avons les données des particules qui sont mis à jour à chaque modification. Les ***Vector3***, en revanche, sont copiés car on veut travailler avec des données fixes, ce qui serait impossible s'ils étaient passés par référence.

Semaine 3:

Le Blob

Notre personnage de jeu, pour cette phase, est un Blob composé de 19 particules. Chaque particule est soumise à des ***SpringForce*** liées aux particules adjacentes. Une

illustration de notre Blob est présent sur la Figure 1, où les nombres correspondent à l'indice de la particule dans le tableau correspondant au Blob.

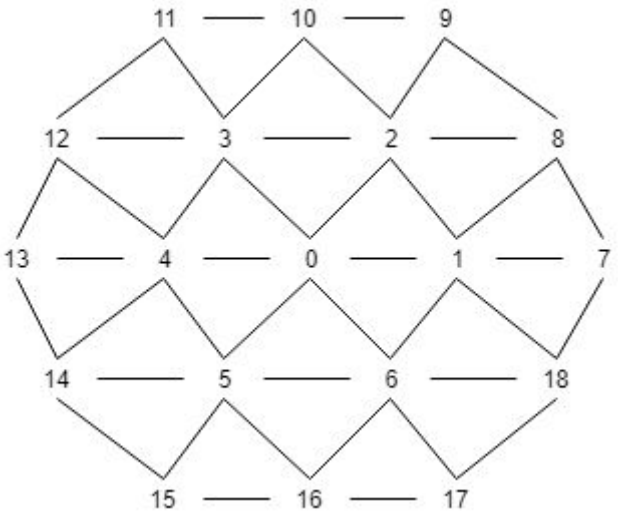

Figure 1 - Interactions des particules du Blob

Le Blob possède un booléen *assembled* permettant de gérer les forces appliquées aux particules. Si le booléen est à *false*, on ne tient pas compte des forces des ressorts. Dans le cas inverse, les ressorts sont appliqués et le Blob est sensé retrouver sa forme initiale. Le clic gauche de la souris permet de changer d'état.

Gestion des contacts

Afin de gérer les contacts, nous avons utilisé un ***vector*** comme pour le registre. De cette façon, nous pouvons ajouter un nombre aléatoire de collisions, représentées par des instances de ***ParticleContact***, sans avoir à gérer l'espace mémoire alloué pour notre conteneur.

Les classes permettant de créer expressément des collisions (***LinkContactGenerator*** et ***CableContactGenerator*** implémentant ***ParticleContact Generator***) possèdent une méthode *AddContact* qui prend le conteneur en paramètre. Elle permet d'ajouter une collision au conteneur. L'implémentation des deux classes ***LinkContactGenerator*** et ***CableContactGenerator*** est commencée mais n'est pas fonctionnelle à ce stade.

Résolution de la collision

Nous avons rencontré un problème lors de la résolution de collision. En effet, lorsque nous tentions de résoudre les collisions, la résolution ne tenait pas compte de la masse relative des particules impliquées. Ainsi, une particule de faible masse pouvait mettre en mouvement une particule plus lourde à la suite d'une collision.

Nous avons donc décidé d'utiliser la formule ci-dessous pour résoudre les collisions en tenant compte des masses :

$$v_a = \frac{C_R m_b (u_b - u_a) + m_a u_a + m_b u_b}{m_a + m_b} \quad v_b = \frac{C_R m_a (u_a - u_b) + m_a u_a + m_b u_b}{m_a + m_b}$$

(cf. https://fr.wikipedia.org/wiki/Collision_in%C3%A9lastique)

Dans cette formule, v_a et v_b sont les vitesses des particules après impact, u_a et u_b les vitesses avant impact, m_a et m_b leur masse respective et C_R le coefficient de restitution.

Gestion de plan

Nous avons implémenté la distinction entre plans et particules. Lors de la collision entre 2 particules, on instancie une ***ParticleCollision*** en passant les 2 particules en argument. Dans le cas d'une collision entre la particule et un plan, ***ParticleCollision*** est instanciée avec la particule passée en paramètre ainsi que 2 ***Vector3*** (le vecteur normal au plan et le vecteur de coordonnées d'un point du plan). Le booléen *isBplane* est alors passé à *True* afin de savoir comment gérer la collision.