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.

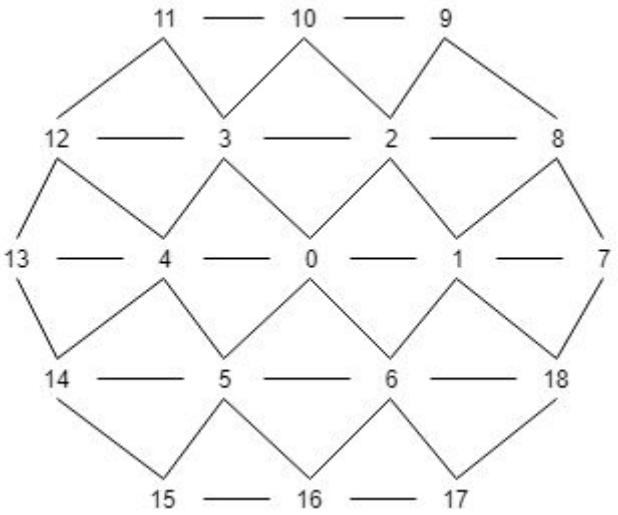

Figure 1 - Interactions des particules du Blob

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. Les flèches directionnelles du clavier permettent de le déplacer dans les 4 directions (la flèche du haut fait sauter le Blob).

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.

Phase 3

Semaine 1 :

Implémentation de la classe Quaternion et de ses tests unitaires

Au cours de la première semaine de cette troisième phase, nous avons implémenté la classe ***Quaternion*** avec les attributs et méthodes vus en cours. En plus de cela, nous avons implémenté les méthodes de *Log*, *Exp* et *Pow* pour le ***Quaternion***. Ainsi, si par la suite, nous avons besoin de réaliser des interpolations linéaires, nous pourrons nous servir de ces méthodes afin de simplifier les calculs.

Les éléments du quaternion ne sont pas stockés dans un tableau. Ainsi, ils sont en accès direct via attributs *w*, *x*, *y* et *z* de la classe.

Afin d'assurer l'exactitude de notre implémentation, nous avons implémenté une batterie de tests unitaires. Nous pouvons utiliser les ***Quaternions*** avec la certitude que notre implémentation ne contient pas d'erreurs.

Implémentation de la classe Matrix3

Nous avons également implémenté la classe ***Matrix3*** avec les attributs et les méthodes vus en cours. Les éléments de la matrice sont accessibles via l'attribut public *data* qui est un *std::Array<double>* ou via l'opérateur *[]* qui a été redéfini. L'accès aux éléments se fait avec les indices suivant :

0	1	2
3	4	5
6	7	8

Des tests unitaires ont été implémentés afin de nous assurer de l'exactitude de notre implémentation.

Semaine 2 :

Fixe des collisions et des impulsions de la phase 2

Précédemment, nous résolvions une collision uniquement une fois sans prendre en compte les possibles collisions que cette résolution générait. Nous avons corrigé ce point; à présent, nous appliquons notre résolution sur au plus 2n collisions, comme ce qui était souhaité.

Pour ce qui est des impulsions, nous avons repris notre première implémentation, basée sur les formules du cours, et avons corrigé cette dernière afin d'obtenir les résultats attendus.

Implémentation de la classe Matrix4

Au cours de la seconde semaine, nous avons implémenté la classe ***Matrix4***. Principalement basé sur l'implémentation de la classe ***Matrix3***, nous avons retiré les méthodes relatives aux ***Quaternions*** hormis le constructeur.

À l'instar des ***Matrix3***, nous accédons aux éléments soit via l'attribut public *data* qui est également un *std::Array<double>*, soit via la redéfinition de l'opérateur *[]*. Les éléments sont accédés via les indices suivant :

0	1	2	3
4	5	6	7
8	9	10	11

Comme pour ***Matrix3*** et ***Quaternion***, ***Matrix4*** possède des tests unitaires afin de vérifier son implémentation.

Implémentation de la classe RigidBody

Par ailleurs, nous avons implémenté la classe ***RigidBody***, équivalent 3D de la classe ***Particle***. Cette classe permet d'intégrer la rotation à nos objets. Nous avons, d'ailleurs, ajouté deux méthodes *WorldToBody* et *BodyToWorld*. Ces méthodes nous permettent de simplifier le passage du repère du monde au repère de l'objet et inversement.

Nous avons repris nos générateurs de forces de la phase 2 et avons appliqué de légères modifications afin de les utiliser avec des ***RigidBody***. En plus de cela, les générateurs de forces dûes à l'inertie et la rotation ont été ajoutés au projet.

Nous avons eu de légères difficultés à savoir comment utiliser l'attribut *transformMatrix*. En effet, nous avions du mal à savoir s'il s'agissait de cette matrice ou de son inverse à utiliser pour le changement de repère du monde vers celui du ***RigidBody***. Nous avons dû réaliser des tests fonctionnels afin de nous assurer de l'utilisation de cette dernière.

Semaine 3 :

Passage au rendu graphique 3D

Nous avons utilisé OpenGL avec SDL2 pour réaliser le rendu 3d. Nous avons associé à chacun des ***RigidBody*** à rendre un objet ***Cube*** décrivant les vertex de l'objet. Ces vertex sont translatés et rotationnés selon la position et l'orientation du ***RigidBody***. Nous

avons également ajouté un *shader* multicolor pour bien visualiser la rotation de l'objet dans l'espace.
Notre rendu 3d représente ainsi deux cubes (des voitures) entrant en collision : l'une continue sa route alors que l'autre subit l'impact qui lui confère une vitesse angulaire. De plus, la scène 3D contient un cube soumis à un couple de forces de résultante nulle : ce cube, non soumis à la gravité, effectue donc une rotation dans l'espace sans se déplacer.

Phase 4

Implémentation de la classe GameObject

Nous avons décidé pour cette phase de notre projet d'implémenter une nouvelle classe : ***GameObject***. Dans cette classe, nous définissons le type de nos objets de scène. En effet, un ***GameObject*** possède un ***Rigidbody*** et une liste de l'ensemble des ***Primitive*** de notre objet, ainsi qu'une ***BoundingSphere*** centrée sur l'objet, dont les dimensions sont définies arbitrairement par l'utilisateur.

Implémentation de la Broad phase

Pour implémenter notre **Broad phase**, nous avons décidé de nous servir de ***BSP Tree*** dû à la facilité d'implémentation. Par ailleurs, nous n'arrivons pas à voir comment utiliser les autres structures de données pour cette phase.

Au sein de notre classe ***BSP Tree***, nous avons défini des structures ***BSP Node*** et ***BSP Plane*** pour l'implémentation de notre ***BSP Tree***. Pour faire la détermination de nos collisions, nous ajoutons aux données de notre ***BSP Node*** un ***GameObject***. Si le ***Node*** possède plus de la capacité maximale de ***GameObjects***, nous réalisons une coupe au niveau du premier ***GameObject*** afin de générer 2 régions et on reproduit le processus.

Au cours de notre implémentation, nous avons rencontré un problème avec la détection de collision au cours de la **Broad phase**. En effet, lorsqu'un ***Node*** possédait plus de 2 ***GameObjects*** sur le plan de coupe, les collisions étaient dupliquées car les ***GameObjects*** étaient ajoutés dans les 2 noeuds fils. Pour résoudre, nous avons différencié les ***GameObjects*** provenant du même noeud des autres ***GameObjects***. En effet, nous testons les collisions des ***GameObjects*** d'un ***Node*** avec les ***GameObjects*** du ***Node*** parent mais sans tester les ***GameObjects*** du ***Node*** parent entre eux.

Implémentation de la classe Primitive

Pour nos primitives, nous avons fait une distinction entre une **boîte** et un **plan**. La classe ***Primitive*** est donc une classe abstraite possédant les attributs vus en cours. À cela, nous avons ajouté un attribut ID qui permet de différencier les pointeur vers une boîte de ceux vers un plan, vu qu'on ne manipule que des ***Primitive**** dans notre méthode *resolveCollision, qui est une* méthode virtuelle.

Nous avons implémenté les classes ***BoxPrimitive*** et ***PlanPrimitive*** qui étendent la classe Primitive. Chacune propose une implémentation différente de *resolveCollision*, permettant de gérer uniquement la collision entre une boîte et un plan.

Rendu graphique

La méthode de dessin des pavés a été changée pour que la classe de rendu ne contienne plus 8 vecteurs mais un seul vecteur des demi-dimensions du pavé. Cela limite les déformations du modèle au fil du temps.

Pour la dernière phase, les plans sont en réalité des pavés très fins que nous avons réutilisés pour éviter de changer la structure de la boucle principale.