UQAC

Journal de développement

Groupe 5 - Kiryu

Amaury Mehlman
David Ponton
Clark Randall
Pierre Teodoresco

Phase 1 : Construction d'un moteur élémentaire de gestion de particules

I. Classe Vector3D

La classe **Vector3D** assume le rôle clé de la représentation d'un type vecteur qui comprends donc les trois composants $(x,\ y,\ z)$, ainsi qu'un ensemble d'opérations usuelles manipulant ces trois composants.

Pour un certain nombre d'opérations, nous avons choisi d'utiliser la surcharge d'opérateur offerte par C++ pour faciliter plus tard leurs utilisations. On retrouve par exemple la multiplication par un scalaire et le produit vectoriel qui utilisent tous deux l'opérateur * (et la version avec affectation ***=**) en changeant le type du deuxième opérande (flottant ou **Vector3D**).

D'autres opérations existent sous la forme de méthodes classiques. De manière non exhaustive, on y retrouve la normalisation, le produit scalaire, le produit par composants etc…

Finalement, pour accroître sa compatibilité avec OpenFramework, cette classe **Vector3D** propose dans son interface deux méthodes de conversion, qui permettent à partir d'un objet **Vector3D** de récupérer un objet ***glm::vec***. Il s'agit des méthodes ***Vector3D::v2()*** et ***Vector3D::v3()*** qui renvoient respectivement un ***glm::vec2*** et un ***glm::vec3***. Nous pouvons aussi imaginer, par la suite, devoir implémenter un ou plusieurs constructeurs de ***Vector3D*** qui prendraient en paramètre un ***glm::vec*** afin de faciliter le passage entre les deux.

II. Tests unitaires de la classe Vector3D

Nous avons écrit un ensemble de tests unitaires pour la classe **Vector3D**. Il s'agit d'une interface de tests contenant des fonctions appelant les différentes méthodes / opérateurs mis à disposition par la classe **Vector3D**.

III. Classe Particle

La classe **Particle** contient différents attributs et getters/setters à ces derniers. On retrouve les différents **Vector3D** pour la position, la vélocité, l'accélération et la somme des forces. Mais aussi l'attribut *inverseMass* permettant de donner une masse à nos particules tout en contournant le problème de masse infinie pour les particules statiques (elles auront ici un attribut *inverseMass* valant 0).

IV. Intégrateur d'Euler

Particle::integrate() est une méthode de la classe **Particle** qui utilise l'intégrateur d'Euler pour mettre à jour les vecteurs accélération, vélocité et position en utilisant la somme des forces subies. Dans notre code c'est donc une fonction membre de la classe **Particle** qui, pour être efficace, nécessite d'avoir renseigné les différentes forces subies par une particules à l'aide la méthode *Particle::applyForce()*.

V. Simulateur de tir balistique

L'ensemble du simulateur de tir balistique se trouve dans l'espace de nom **Ballistic**. Nous avons fait le choix d'utiliser simplement un espace de nom et non pas une classe parce que nous préférons limiter l'utilisation de classe à la définition de type (**Vector3D** et **Particle**).

L'interface **Ballistic** propose un ensemble de fonctions dont le nom est repris d'une partie des méthodes de la classe **ofApp** d'OpenFramework. On peut prendre l'exemple des fonctions ***Ballistic::update()*** et ***Ballistic::draw()*** qui ont respectivement la tâche des calculs de position en utilisant l'intégrateur de la classe **Particle** et l'affichage des différents informations à l'écran (les particules, la trajectoire ou encore du texte pour les commandes utilisateurs). En ce moment, l'espace gère plusieurs particules, mais n'affiche que la trajectoire de la dernière, cela est intentionnel car il n'est actuellement pas possible de supprimer une particule et donc l'affichage de toutes les trajectoires deviendrait encombrant. Notre échelle suppose que chaque pixel de l'écran représente 1cm, et donc la gravité est un vecteur de dimensions (0, -981, 0).