# Journal de Développement :

Développement d’un Moteur Physique en C++

## LES CHOIX

Etape 1.

**Outils de développement :** Visual Studio, OpenGL (avec librairies Glew et Freeglut)

L’IDE de développement sélectionnée est Visual Studio car nous l’avions déjà installé chacun de notre côté. Concernant la librairie graphique, nous utilisons OpenGL car Martin et Clémence l’avaient déjà utilisée l’année précédente dans d’autres projets sous QtCreator.

Nous avons toutefois dû adapter son utilisation à Visual Studio, via les bibliothèques *glew* et *freeglut*.En effet, QtCreator encapsulait OpenGL de telle manière que son utilisation ne requérait rien de plus que l’IDE de développement.

Etape 2.

On a représenté les blops par un ensemble de particules liées entre elles. On a ajouté une piscine et un mur afin d’effectuer quelques tests. Mis à part ces choix au niveau de la méthode pour tester la physique du jeu, la partie logicielle est la même.

Etape 3.

Malgré l’impossibilité de construire un cMake compatible avec la librairie freeglut, nous avons décidé de continuer à utiliser celle-ci car nous ne voulons pas perdre du temps à chercher une autre librairie graphique 3D, que nous devrons apprendre à maitriser de 0.

Pour le code, nous avons beaucoup repris le code précédent, en ajoutent les nouvelles classes nécessaires, supprimant celles inutiles et en adaptant au fur et à mesure celles réutilisables, en prenant soin de tout commenter.

## LES DIFFICULTÉS

Etape 1.

Nous avons rencontré diverses difficultés durant le développement :

* L’utilisation d’une librairie graphique de bas niveau requiert plus d’effort pour l’affichage, et avons donc passé du temps à programmer l’affichage et rajouter une couche d’abstraction à notre code.
* Comme *freeglut* ne supporte pas bien l’encapsulation de ses fonctions de callback (la librairie étant en C, elle émet des erreurs si on appel des fonctions appartenant à des instances d’objets), nous avons dû mettre en place un hotfix avec une instance statique de la classe gérant l’affichage graphique. Ce n’est pas une solution élégante mais nous avons préféré ne pas perdre trop de temps sur l’affichage, car ce n’est pas l’objectif premier de notre travail.
* Le C++ étant un langage nouveau pour Elie, il lui a fallu apprendre la syntaxe et autres spécificités du langage au fil du projet.
* Nous avons également rencontré des certains problèmes liés aux pointeurs, notamment liés à deux classes se partageant une instance de Vector3D, problèmes assez récurrents lors du développement d’un programme en C++.

Etape 2.

Pour la partie force, nous avons rencontré des difficultés pour la dernière force, celle de faible ressort. Celle-ci n’a pas été implémentée finalement car nous avons préféré se concentrer sur les contacts.

En ce qui concerne ces derniers, nous avons rencontré des difficultés au niveau de la répartition des effets : l’interpénétration a plus d’impact sur la résolution des contacts que les impulsions. Nous n’avons pas implémenté le fait qu’une vitesse trop faible soit ignorée mais au finalnous n’avons pas d’effet de vibration au niveau du sol.

Nous avons également changé le frame rate dynamique cependant celui semble avoir des soucis au niveau des librairies de graphisme qu’on utilise : la variation des frame rate (donc l’augmentation d’objets à traiter) fait varier un peu les mouvements.

Etape 3.

La principale difficulté était liée à la compréhension d’un outil mathématique qu’on avait jamais utilisé auparavant : les quaternions. Une fois l’outil compris, il fallait l’adapter au code et donc à l’affichage en 3D. Par exemple il fallait effectuer des changements de représentation pour faire les rotations lors du dessin et passer des degrés aux radians. Pour l’instant ces étapes n’ont pas posé de soucis au niveau des arrondis sauf pour la rotation autour de l’axe y où l’on peut observer des problèmes lors du calcul de l’angle d’Euler associé (pour certaines orientations l’angle n’existe pas).

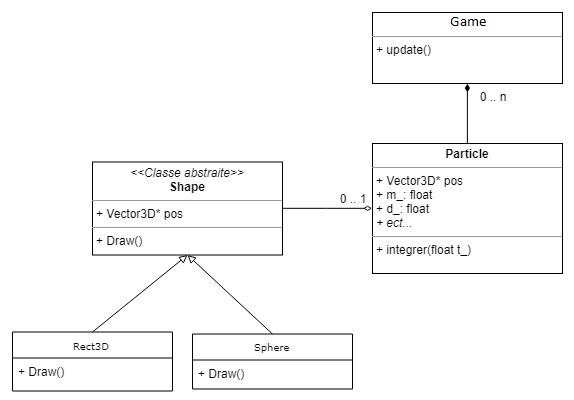
L’autre difficulté était de faire la collision pour une simple cas. On a d’abord essayé avec l’application d’une force à un point fixe mais on a eu le même problème que pour les collisions précédentes car la collision est plutôt semblable à une impulsion. On a donc repris les étapes du précédent résolveur de collisions en simplifiant le procédé et en ajoutant une petite étape pour la rotation (assez similaire à l’étape de modification de la vitesse).

## CHOIX D’IMPLEMENTATION

Etape 1.

La séparation a été faite entre la logique et l’affichage, une classe **Game** gère la majeure partie de la simulation, et possède une liste d’objets **Particle** qui contient les données importantes à la physique, qui peuvent chacun avoir un objet **Shape** gérant leur affichage.

Un UML simplifié du système :



Rect3D et Sphere sont des classes filles de Shape implémentant une fonction Draw() qui trace via OpenGL leur forme à la position de la particule.

La classe Jeu fait le lien entre les différentes classes, fait les appels de l’intégrateur sur chacune des particules puis dessine la scène.

Etape 2.

Pour les forces, nos choix sont assez similaires au cours. Cependant, pour les contacts nous avons choisi de reproduire la structure du registre de force avec un **ContactResolver** qui contient une liste de **ParticleContact** et qui va la parcourir pour résoudre l’ensemble des contacts.

Etape 3.

Pour les tests on a utilisé un héritage de la classe RigidBody qui s’appelle Box. Pour l’instant le jeu ne gère que des box. On a séparé l’ensemble des box du jeu en deux listes pour les deux test de cette étape car les box du 2ème test ne sont pas soumises à la gravité (constamment sur le sol). Une fois le système de collision mis en place, on conservera seulement une seule liste de RigidBody. Pour implémenter la collision, comme préciser lors des difficultés de l’étape 3, on a modifié la vitesse et la rotation de manière un peu similaire à la résolution de collision de l’étape 2. Pour la modification de la rotation de la box 1, on a utilisé la formule suivante :