PROJET POO2

Simulateur 2D de gravité

UFR MATHS-INFO
HAEHNEL Jonathan & PAPILLON Marc
L3 Informatique - S5A

I/ Etat des lieux :

a) Fonctionnalités implémentées :

Description de la fonctionnalité	Type
Collision entre un rond-rond	Primaire
Collision entre un rond-rectangle	Primaire
Affichage visuel des formes dans un canvas	Primaire
Modification du coefficient de gravité	Primaire
Ajout/Suppression de nouvelle forme	Secondaire
Mise en place des rebonds	Secondaire
Objet utilisant des images	Confort
Possibilité de colorier une forme et sa bordure	Confort
Accélérer le temps	Confort
Ajouter une forme avec un système de “glisser-déposer”	Confort
Mise en place d’un système de maps (objet fixe)	Confort
Déplacement d’un objet	Confort

b) Fonctionnalités non implémentées :

Nous avions prévus d’ajouter quelques autres fonctionnalités, cependant, par manque de temps, nous n’avons malheureusement pas pu les intégrées.

Description de la fonctionnalité	Priorité
Collision entre un rectangle-rectangle	Primaire
Rotation d’un cercle et d’un rectangle	Primaire
Lancement d’un objet avec la souris	Secondaire

c) Répartition des taches :

Personne	Tache
HAEHNEL Jonathan	• Création de l'interface graphique • Mise en places des formes (Shape) • Création du système de maps • Placement/Suppression d'un nouvel objet • Rédaction du rapport
PAPILLON Marc	• Gestion des collisions et du canvas • Mise en places des formes (Shape) • Rédaction du rapport

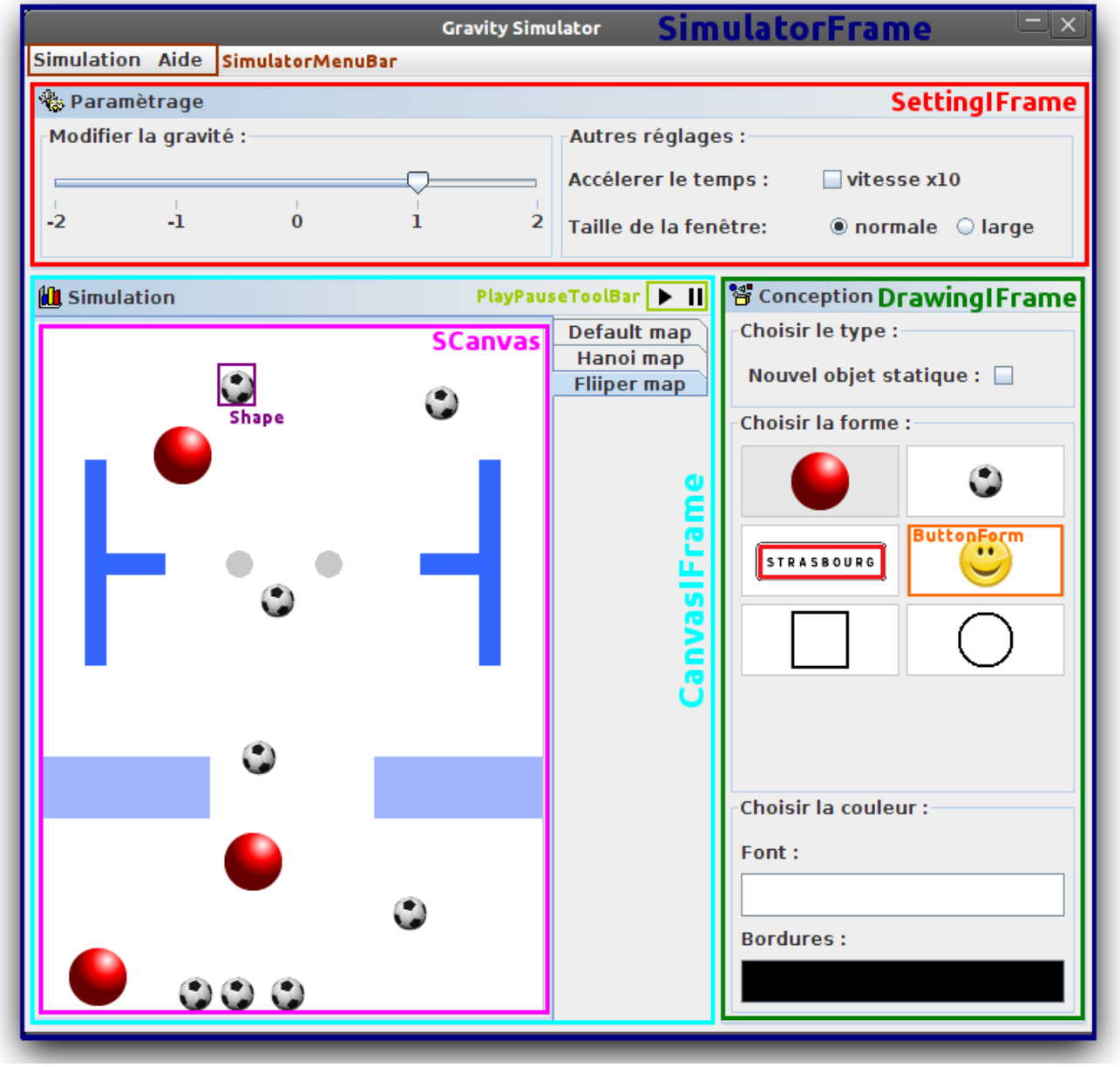
II/ Interface graphique :

a) Aperçu global

Nous avons choisis de faire une interface **agréable à utiliser et ergonomique.** Pour rendre l'application plus esthétique, nous avons utilisé une librairie externe (*JGoodies*) pour la mise en place de composants graphiques supplémentaires (par exemple : une classe IFrame)

Toutes les classes permettant la construction de l'interface graphique sont situées dans un unique **package "IHM"** qui un total de 8 classes Java.

Ci-dessous, une capture d'écran de l'interface graphique avec les classes correspondantes :

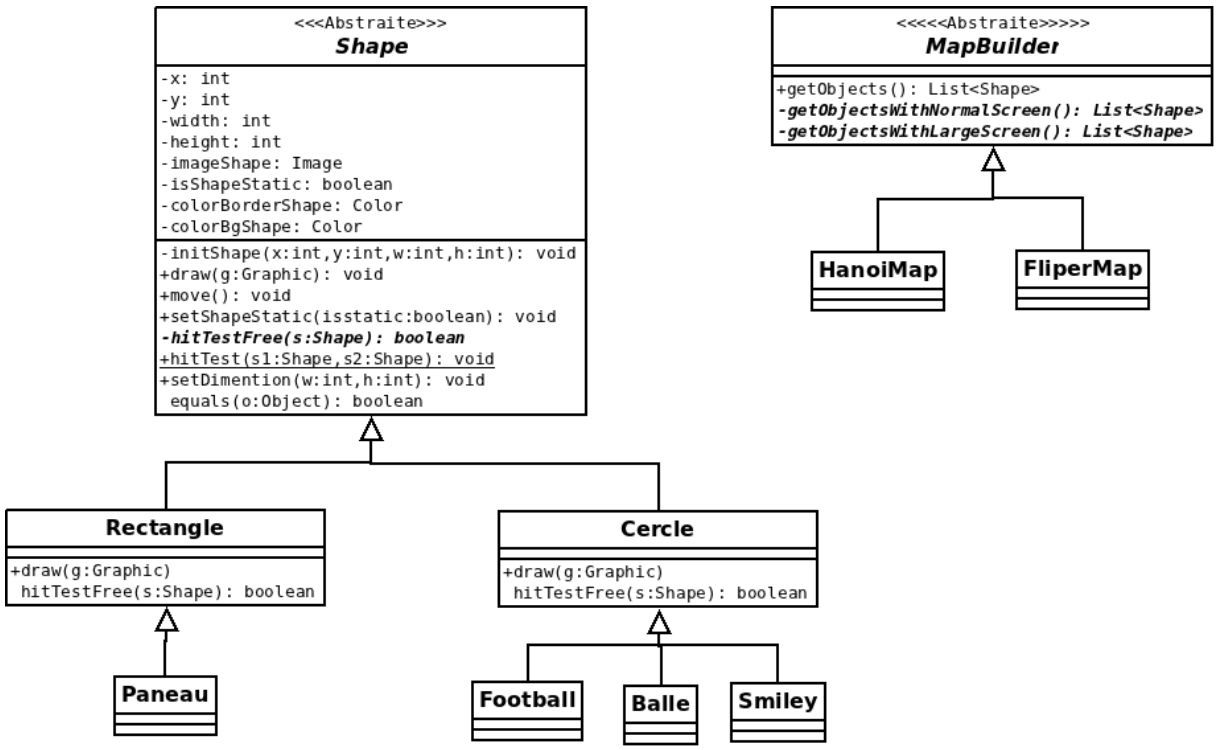


On remarque tous de suite que l'interface est **découpé en 3 parties** (trois IFrame) :

- La conception : L'utilisateur peut y choisir ses formes à ajouter avec les couleurs de son choix.
- La simulation : Cette zone (canvas) permet de visualiser directement les objets en mouvement. En cliquant, vous pouvez placer ou supprimer des objets de la simulation. Vous pouvez également choisir une map. *(cf. II/ c)*
- Le paramétrage : Dans cette partie, vous pouvez modifier des paramètres de bases du moteur graphique comme par exemple, faire varier la valeur de la gravité.

B) Gestion des formes

Pour mettre en place les formes, nous avons décidé de créer une classe par forme et d'utiliser un héritage. Une taille de la forme est soit déterminée directement par des données numériques, soit par la taille de l'image de forme.
Le package concerné par les formes est "**Shape**".



C) Concept des maps

Afin de rendre les simulation plus intéressante, nous avons rajouter des **maps prédéfinies.** En effet, ce sont des ensembles d'objets fixes pré-inserés dans le Canvas. Chaque map peut posséder deux ensembles d'objets :

- Pour l'affichage dans une fenêtre normale (350 pixels X 475 pixels)
- Pour l'affichage dans une fenêtre large (800 pixels X 475 pixels)

Le package concerné par les maps est "**Maps**".

III/ Gestion des collisions :

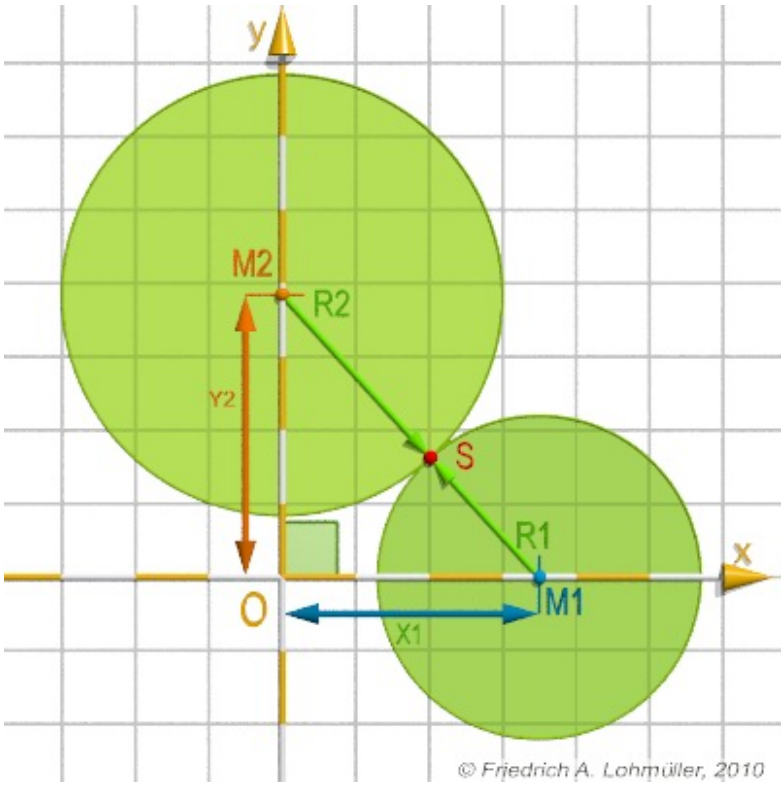
Avant de parler des collisions, il est nécessaire de savoir comment est implanté le système permettant aux objets de se déplacer. Notre choix s'est porté sur l'utilisation de vecteurs et d'ajouts de force telle que la gravité.

Ainsi chaque "Shape" à son vecteur vitesse et ses coordonnées x et y. A chaque frame on ajoute le vecteur vx à la coordonné x de l'objet, et de même pour le vecteur vy à la coordonné y.

Nous considérons la gravité comme un vecteur mais qui ne peut avoir qu'une influence dans l'axe verticale, il n'était donc pas nécessaire de le considérer comme un vecteur dans notre programme mais comme un simple chiffre (double ou float) qui s'appliquera au vecteur vy de nos objets.

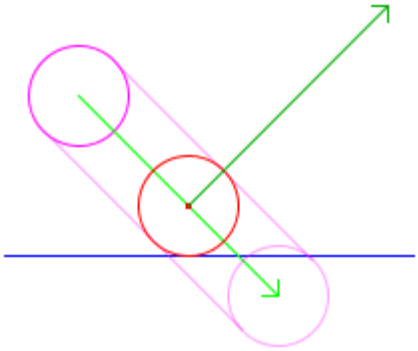
a) Collision rond-rond:

Les collisions rond-rond sont certainement les moins coûteuses pour l'ordinateur car de simples calculs sans test préalable permettent d'obtenir des résultats satisfaisants. Pour détecter si une collision a lieu, la solution la plus simple est de calculer la distance entre les deux balles, si cette distance est inférieure ou égale à l'addition des deux rayons de ces ronds, il y a une collision entre les deux. Pour ce faire le théorème de pythagore est utile : Racine de (distance x^2 + distance y^2).



La prochaine étape consiste au repositionnement des balles après que la collision est eu lieu. Cette étape, bien que compliqué, est nécessaire car la collision peut être détectée après qu'elle est réellement eu lieu. Si c'est le cas et qu'aucun repositionnement n'est fait, les objets risque de se "rentrer dedans" à l'infini et de bloquer l'un dans l'autre.

Notre méthode pour le repositionnement consiste à calculer le point d'impact des deux balles (midPoint) puis on remettre en place les objets tels qu'ils étaient lors de la collision. Ceci nécessite des calculs prenant en compte la normale et le rayon des objets concernés.



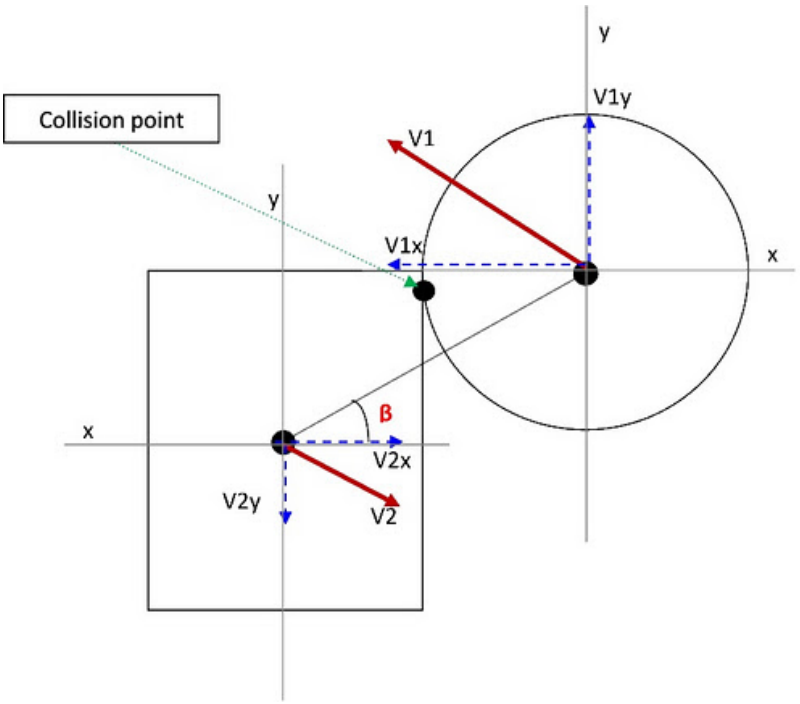
Repositionnement d'une balle après une collision

La dernière étape est le calcul des nouveaux vecteurs. Ici il faut prendre en compte les anciens vecteurs et les projetter grâce à la normale (qui se calcule en fonction de la distance entre les deux objets). Puis on met à jour les vecteurs de chaque objet et la fonction de collision est terminée.

Cette méthode est précise et ne nécessite pas d'encapsuler nos cercles dans des "bounds".

B) Collision rectangle-rond :

La collision rectangle - rond est différente de celle citée plus haut car nous ne pouvons pas utilisé de formules similaires. En effet pour savoir la distance entre les deux objets cela nécessiterait plusieurs étapes comme le calcul du plus proche point des objets sur le segment le plus court qui les relie. Puis des tests seraient nécessaire pour savoir quelle partie du rectangle le rond touche.



Pour optimiser les perfomances et limiter les calculs il est donc plus simple de considérer un rond comme un rectangle. Ainsi il suffit de tester chaque “cas” possible en fonction de la position d’un objet par rapport à l’autre.

Nous avons découper les cas de cette façon :

- Si le vecteur vx de la balle est positif
    - Si la balle ne touche pas un côté de la balle
        - Si le vecteur vx de la balle est négatif
        - Sinon …
    ...
- Si le vecteur vx de la balle est négatif
    ...
- Si le vecteur vx est nulle
    …

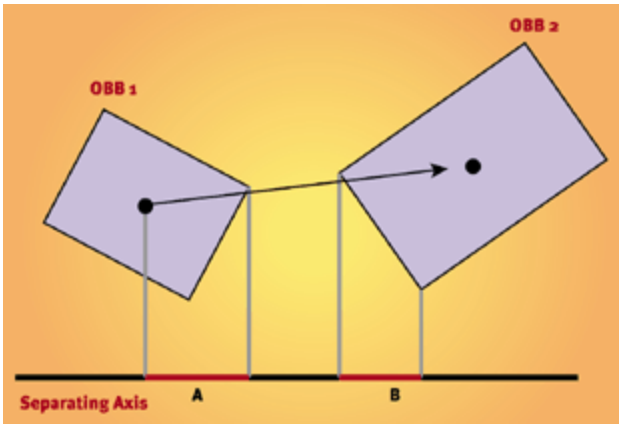
Cette décomposition de cas peut sembler fastidieuse mais elle permet de repositionner instinctivement la balle par rapport au rectangle et d’orienter son vecteur vitesse facilement. Évidemment ce n’est pas la méthode la plus précise car la balle est encapsulé dans un rectangle mais elle n’est pas très coûteuse car il ne s’agit que de quelques tests simples.

c) Collision rectangle-rectangle:

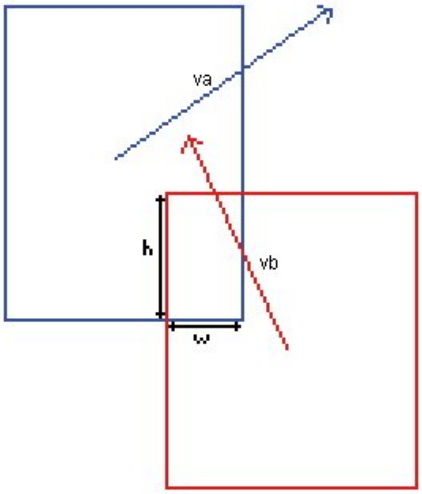
Nous n'avons malheureusement pas eu le temps de mettre en place le système de collision entre rectangle et avons donc choisi d'en faire des objets fixes. Les balles peuvent donc "rouler" ou de rebondir sur les rectangles que nous mettons en place.

Malgré cela plusieurs choix était à notre disposition pour gérer ces collisions.

Un système prenant en compte les rotations : separating axis test. Théoriquement c'est simple, si une ligne qui sépare les deux objets existe, ils ne sont pas en collision.



Une deuxième solution, plus simple mais qui ne prend pas en compte les rotations d'un rectangle. Il s'agit comme pour les balles de calculer le point d'impact afin de repositionner les rectangles puis de calculer les nouveaux vecteurs.



Cette méthode ne semblait pas très compliqué à mettre en place puisque proche de la méthode que l'on utilise pour la collision rond - rond, mais il semblait étrange que deux rectangles puissent se "cogner" sans pour autant changer de rotation.