Rapport Projet Programmation Orientée Objet (JAVA)
Groupe Teide n° : 4

1) Balls et Balls simulator:

(fichiers : Balls.java, BallsSimulator.java, TestBallsSimulator.java)

Notre choix pour stocker les balles s'est porté sur une LinkedList de java.util pour ajouter et modifier les balles rapidement.
Aussi, le choix du nombre de balles se fait directement sur BallsSimulator.java car dans le constructeur de cette classe, on passe en paramètre le nombre de balles.
Les balles ont une trajectoire et une position aléatoires, et pour ce fait, on initialise un objet Random et on utilise la méthode nextInt() de cette classe.
Pour ce qui est du rebondissement des balles, on utilise deux vecteurs dx_vector et dy_vector pour changer la position de chaque balle à appel de next().

Pour tester BallsSimulator : make TestBallsSimulator

2) Jeu de la vie:

(fichiers : JeuDeLaVie.java, Grille.java, TestJeuDeLaVie.java)

On a commencé à implémenter comme tous les groupes le Jeu de la Vie indépendamment des autres jeux. Mais ce n'est évidemment pas la version finale présente sur le code source rendu.
ET pour cela il faut don parler de la classe mère d'en même temps le Jeu de la Vie, Le jeu d'Immigration et le modèle Schelling : Grille.java.

Cette classe regroupe tout ce qu'il y a en commun entre ces trois jeux et essaye de factoriser au maximum les méthodes.
Par exemple le nombre d'éléments est un paramètre pour cette grille aussi pour déterminer le nombre de couleurs à ajouter.
Il y aussi une fonction voisinage qui regroupe dans une table les voisins d'une cellule en particulier et dessiner() qui dessine les cellules.
Pour là aussi, tous les paramètres sont à choisir dans TestJeuDeLaVie.java comme le nombre de cellules dans la grille et la taille d'une seule cellule.

Pour tester JeuxDeLaVie : make TestJeuDeLavie

3) Jeu de l'immigration :

(fichiers : JeuImmigration.java, Grille.java, TestJeuImmigration.java)

Il n'y a pas trop de différence entre le Jeu de La Vie et ce jeu si ce n'est les règle et la possibilité d'augmenter le nombre de couleur, ce qui est géré directement dans la classe
Grille.java. En ce qui concerne les couleurs, elles sont ajoutées aléatoirement en jouand sur les composantes r,g et b d'une couleur.
Par exemple:

```
r = (k * 100 + 200*nbr_case_y + 56)% 255;
g = (5*(k*200 + 400 * nbr_case_x + 56))% 255;
b = (k*500 + 600 + taille_case_x + 56) % 255;
```

Avec des nombres complétement aléatoires pour pouvoir ajouter des couleurs et modulo 255 pour ne pas le dépasser.

Pour tester JeuImmigration : make TestJeuImmigration

4) Modèle de Schelling :

(fichiers :JeuSchelling.java, Grille.java, TestSchelling.java)

Le modèle de Schelling se bas lui aussi en majeure partie sur Grille.java où une fonction placeVide() qui renvoie une LinkedList<Point> indiquant les maisons vacantes, et se choix se justifie par le fait que c'est un vecteur qui varie à chaque tour, ce qui rend la tâche d'ajouter (.add()) et d'enlever (.remove()) plus facile. Le seuil est passé lui aussi en paramètre en plus au nombre de couleurs. Les couleurs sont là aussi choisies aléatoirement même on aurait voulu limiter les couleurs à celles des éthnies pour mieux voir l'hypothèse de l'économiste s'appliquer.
Les maisons à occuper à chaque tour sont elles aussi choisies aléatoirement grâce un Math.random() qu'on utilise pour déterminer l'indice de la maison à remplir.

Pour tester Schelling : make TestSchelling

5) BoidsSimulator:

Pour la partie des boids, on a 6 fichiers :

Bird.java :

Type de boids qui a sa propre fonction **updateVelocity()** pour bien déterminer les règles qui régis un groupe d'oiseaux.

Insect.java :

Même chose que Bird tout en étant une proie des oiseaux.

Vector.java :

Classe qui implémente des méthodes utiles pour les vecteurs pour simplifier le code après :
Plus(), moins(), dot(), fois(), distanceTo(), absolue() et **div().**

Animal.java :

Classe mère de **Bird** et **Insect**, regroupe toutes les règles qui peuvent régir un groupe d'animaux.
Cette classe a comme attributs deux objets de classe **Vector** (position et velocity) qui ont eux même un tableau de deux valeurs comme attribut (x,y et Vx,Vy).
Pour les règles implémentées : les trois basiques (**cohesion(), separation()** et **alignement())**
On a dès lors ajouté d'autres règles :
boundPosition(), strongWind() et **limitVelocity()** (comme décrites dans le pseudo-code donné).
predating() : Fonction qui sur le même principe de **cohesion(),** pousse les oiseaux à chasser les insectes en suivant leur centre de gravité.

Boids.java :

Classe principale qui gère l'interaction entre les boids et exécute les règles. Implémente notamment la fonction **predate()** qui illustre les oiseaux dévorant les insectes. Elle supprime à chaque

tour les insectes trop près d'un des oiseaux et met à jour les attributs donc insectes et nombre_insectes de la classe à chaque fois.

Triangle.java :

Classe qui nous permets de représenter les boids sous forme de triangles.

TestBoids.java :

Classe test de notre simulateur.

Pour augmenter l'interaction entre les oiseaux et les insectes, on a aussi rajouter de la séparation chez les insectes envers les oiseaux.
Il est à noter aussi que les règles sont appliquées entre les boids et leurs voisins dans une certaine distance, et les fonctions qui implémentent ça sont dans la classe Boids.java.

Si vous voulez changer de paramètre pour la simulation, il faut le faire directement dans les paramètres de la fonction move_insects() et move_birds() dans next() de la classe Boids.java. Et pour voir les règles qui définissent chaque Boid, rdv les classes Bird.java et Insect.java.

Remarque : il aurait été plus facile de représenter dans BOid.java les boids sous forme d'une collection au lieu d'un simple vecteur, surtout dans la détermination des voisins mais le temps nous nous a pas suffit pour le faire.

Pour tester Boids : make TestBoids (Insectes en rouge et Oiseaux en noir)

6) Event et EventManager :

Rien de spécial à dire sur ces classes, elles sont implémentées et fonctionnelles sur toutes les classes du projet les nécessitant.