Simulation Orientée-Objet de systèmes multiagents

1 Choix de conception et tests

1.1 Balles

On a créé une classe `Balls` qui contient la liste des balles (stockée sous forme de points) ainsi que leurs positions initiales. On a écrit des méthodes classiques sur `Balls` comme par exemple accéder au nombre de balles, définir les coordonnées des balles, mais aussi une méthode `transaction`, et `reInit` qui replace les balles à leurs positions initiales.

Ensuite, on a testé notre classe en prenant 3 balles et en essayant de les translater. On a évidemment aussi essayé, une fois déplacées, de replacer les balles à leurs positions initiales. On obtient le résultat suivant.

```
[java.awt.Point[x=0,y=0], java.awt.Point[x=3,y=2], java.awt.Point[x=5,y=0]]
[java.awt.Point[x=-1,y=7], java.awt.Point[x=2,y=9], java.awt.Point[x=4,y=7]]
[java.awt.Point[x=0,y=0], java.awt.Point[x=3,y=2], java.awt.Point[x=5,y=0]]
```

1.2 Simulateur de balles

Pour cette classe qui implémente `Simulable`, on a définit comme attribut `balls`, de type `Balls`, qui contient donc les positions actuelles et initiales des balles. On a aussi comme attribut la liste `dir` des vecteurs vitesses de ces balles, ainsi que `dir_init` qui correspond aux vitesses initiales.

Ainsi, quand on va replacer les balles au centre, on va bien pouvoir à la fois les remettre à leurs positions initiales tout en ayant les mêmes vitesses initiales. On ajoute aussi `window` de type `GUISimulator` pour pouvoir dessiner. On ajoute `width`, `height` et `colors` car on les utilise dans notre algorithme même s'ils ne semblent pas indispensables mais cela facilite notre algorithme.

On définit ensuite comme demandé les méthodes `next` et `restart` en faisant bien attention de bien recalculer les nouvelles positions et vitesses des balles et on fait aussi attention à bien *reset* `window` avant de redessiner.

Ensuite, on teste notre classe en créant 6 balles et on lance la simulation.

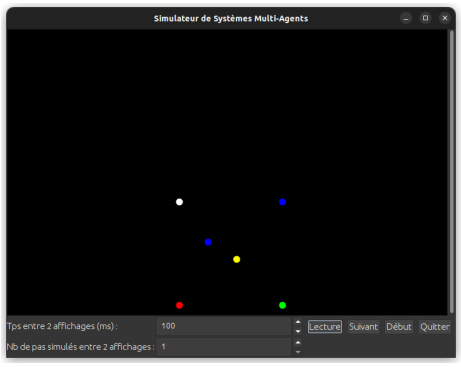

FIGURE 1 – Résultat de la simulation de `TestBallsSimulator`

1.3 Jeu de la vie de Conway

On crée tout d'abord une classe `Cellule` qui nous donne le nombre d'états possibles de la cellule en question et son état actuel. On le fait car on voit que, dans le prochain jeu, on aura besoin de plusieurs états et non seulement deux comme ici puisque la cellule ne peut être que vivante ou morte.

Ensuite, on crée notre classe `Grille` qui implémente `Simulable` et qui va stocker la grille en cours et la prochaine. Cela nous permet, lors du calcul de la prochaine grille, de bien tenir compte des anciennes cellules, et qu'on ne prenne donc pas en compte une nouvelle. Cela n'aurait pas été possible si l'on écrasait directement la valeur. On stocke aussi la grille initiale, histoire de pouvoir revenir au cas initial. Enfin, on a aussi les tailles `n`, `m` et `window` en attribut (pour pouvoir dessiner toujours).

On y définit aussi les méthodes `dessiner`, `next` et `restart`. `dessiner` et `next` sont en `abstract` car on va les écrire spécifiquement en fonction du jeu (`Conway` ou `Immigration`).

Dans le cas du jeu `Conway`, on étend la classe `Grille` en écrivant `dessiner`. Pour cela, on parcourt toute la grille et on change juste la couleur du rectangle affiché en fonction de l'état de la cellule (morte ou vivant). Pour la méthode `next`, on parcourt aussi toute la grille mais on fait bien attention à placer la nouvelle valeur calculée dans `grilleApres`, histoire que `grilleAvant` ne soit pas modifiée lors du calcul. On raisonne par `modulo` pour traiter le cas des bords. Ce n'est qu'à la fin de tous les calculs qu'on va recopier la `grilleApres` dans la `grilleAvant` : dans notre cas, on le fait au tout début de la fonction `dessiner`.

On teste ainsi notre classe avec une grille de taille 10×10.

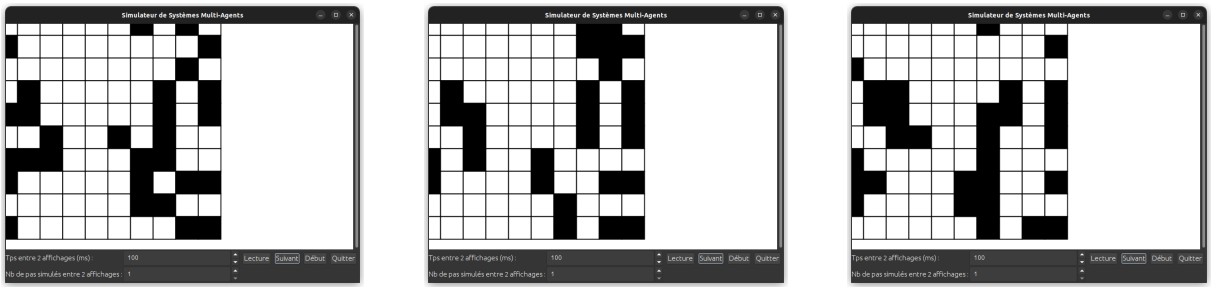

FIGURE 2 – Résultat de la simulation de `TestConway` avec 3 états consécutifs

1.4 Jeu de l'immigration

C'est exactement le même principe que précédemment sauf qu'au lieu d'avoir seulement 2 états par cellule, on en a n. Le seul changement est qu'on stocke le nombre d'états car on l'utilise dans la méthode `next`.

On teste aussi notre classe avec une grille de taille 15×10, chaque cellule pouvant avoir 5 états.

FIGURE 3 – Résultat de la simulation de `TestImmigration` avec 3 états consécutifs

1.5 Modèle de Schelling

Cette classe est, comme précédemment, une extension d'une grille, sauf qu'on ajoute comme attribut une file qui contient l'ensemble des habitations vacantes. On a aussi besoin d'une file temporaire `tmp`. En effet, lorsqu'on va calculer la prochaine file d'habitations vacantes, on a besoin de stocker dans `tmp` les habitations qui ont été quittées, et qui donc sont devenues libres.

On considère qu'une habitation laissée libre ne peut pas directement être habitée, on doit attendre un tour pour qu'elle puisse se remplir à nouveau (après un `next` donc). Ainsi, les habitations qui vont être quittées vont être mises dans `tmp` pour ne pas qu'elle puisse être habitées directement (ce qui aurait été le cas si on la mettait dans la file des habitations vacantes). On va bien évidemment supprimer une habitation vacante de la file pour que la nouvelle habitation puisse y habiter. Ce n'est qu'à la toute fin, qu'on va remettre tous les éléments de `tmp` dans la file des habitations vacantes pour qu'elles puissent être habitées à partir du prochain tour.

Lors de notre calcul pour savoir si une habitation va déménager ou non, on a besoin de compter le nombre de couleurs différentes (sans les doublons évidemment), donc on choisit la collection `Set` (pour qu'on puisse manipuler des ensembles). On précise aussi, que s'il y a n couleurs, il y aura $n+1$ états (0 pour une habitation non habitée, et enfin de 1 à n pour les n couleurs).

On teste notre classe avec 6 états (3 couleurs, et habitation vide) et $K = 3$ (seuil).

FIGURE 4 – Résultat de la simulation de `TestSchelling` avec 3 états consécutifs

1.6 Boids

On crée une classe `Boid` qui va représenter un essaim avec ses coordonnées, ainsi que son vecteur vitesse. On écrit les fonctions classiques qui permettent d'accéder et de modifier les attributs privés. On réécrit aussi la méthode `equals` de la classe `Objet` car on va l'utiliser plus tard.

Ensuite, on crée la classe `BoidsSimulator` avec comme attributs principaux la liste de boids actuelle, la liste de boids à l'état initial, ainsi que `window` pour pouvoir dessiner. On rajoute

aussi `height`, `width` et `nb` (nombre de boids) pour faciliter notre code.

Pour cette classe, on code les 3 règles classiques : cohésion, séparation et alignement. On rajoute aussi la règle `bound_position` pour éviter que les essaims sortent de l'écran. Comme notre classe hérite de l'interface `Simulable`, on réécrit également les méthodes `next` et `restart`. On rajoute aussi une méthode pour dessiner.

Pour dessiner, on créé une classe `Triangle` qui implémente `GraphicalElement` et qui permet à partir des coordonnées (`x` et `y`) ainsi que du vecteur vitesse (`velX` et `velY`) de dessiner l'essaim. Pour cela, on recode la fonction `paint` de `GraphicalElement`.

Enfin, on teste notre simulation avec 100 essaims, `height = 500` et `width = 500`. On précise aussi que initialement, on prend comme vecteur vitesse le vecteur de coordonnées (50, -50). Pour ce qui est des coordonnées initiales, on les choisit aléatoirement en faisant bien attention à ce qu'ils soient dans l'écran.

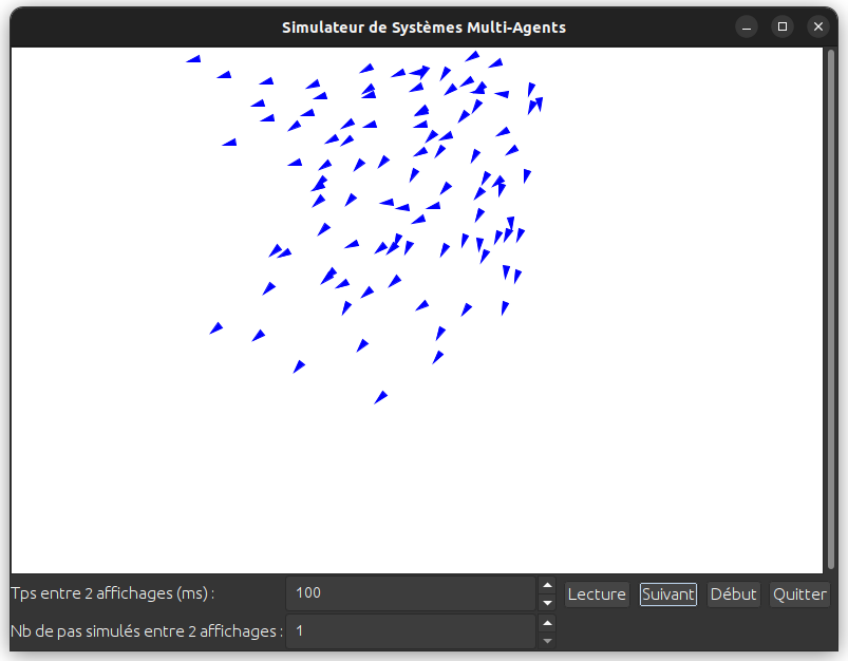

FIGURE 5 – Résultat de la simulation de `TestBoidsSimulator`

2 Diagrammes de classes

Les diagrammes des classes implémentées sont présentés en figures 6 et 7.

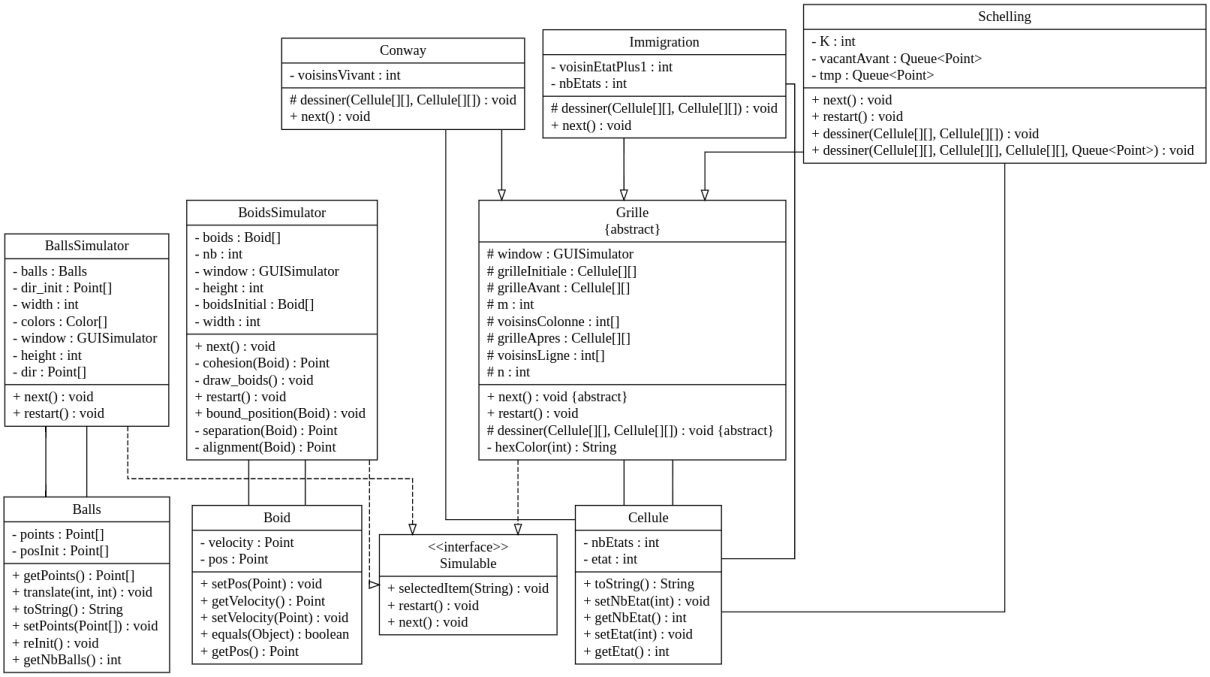

FIGURE 6 – Diagramme de classes

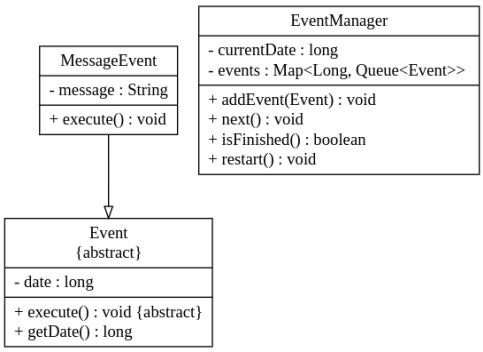

FIGURE 7 – Diagramme de classes