Programmation Orientée Objet

Simulation Orientée Objet de systèmes multi-agents

L'objectif du projet était de manipuler des systèmes multi agents : des entités autonomes qui interagissent entre elles pour atteindre des objectifs individuels ou collectifs. Nous nous sommes d'abord familiarisé à l'aide d'une interface graphique avec la notion d'automates cellulaire (grille de cellule ayant des états) avec plusieurs jeux comme : le jeu de conway, le jeu de l'immigration et le modèle de schelling. Nous nous sommes ensuite intéressés au modèle d'essaim : groupe d'agents cette fois ci libre dans l'espace 2D et interagissant entre eux avec l'orientation et la position des agents environnant.

Nous allons présenter dans ce document dans un premier temps nos choix de conception (architecture, classes) et dans un deuxième temps notre stratégie de tests et nos résultats.

Pour voir comment exécuter le code cliquez ici.

1. Conception

- **Automates Cellulaires**
    - Balle

Classe Balls : la classe gardant en mémoire l'ensemble des balles, leurs positions initiales et leur position courante.

Classe BallsSimulator : la classe implémente l'interface simulable pour la simulation

Classe EventBalls : classe représentant un événement de déplacement de balles. Elle étend la classe abstraite Event.

    - Pour conway et immigration des classes très similaires

Classe Cellule_nom_du_jeu : classe définissant une cellule et ses états possible (mort ou vivant pour conway par exemple)

Classe Plateau_nom_du_jeu : classe définissant l'ensemble des cellules du plateau (matrice de cellule), création des cellules vivantes définit dans le test_jeux, exploration du voisinage (décompte des cellules voisines différentes), affichage graphique. Elle étend la classe abstraite Plateau.

Classe EventNomDuJeu : classe représentant une étape d'un jeu. Elle étend la classe abstraite Event.

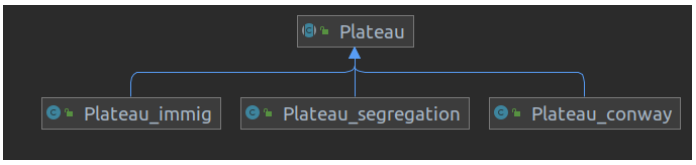
    - Pour Ségrégation

Pour Ségrégation nous avons utilisé la même idée que pour les jeux précédents. Mais nous avons utilisé des collections java (Map<K,V> et List<E>).

De plus, nous avons utilisé un dictionnaire dans la classe Plateau afin de trouver une case vide bien plus efficacement.

- Heritage

Classe Plateau : classe abstraite qui implémente un plateau et qui Plateau_nom_du_jeu, elle regroupe toutes les fonctionnalités communes des 3 jeux présents sur la Figure 1.



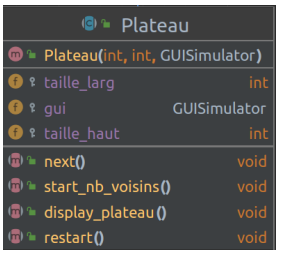
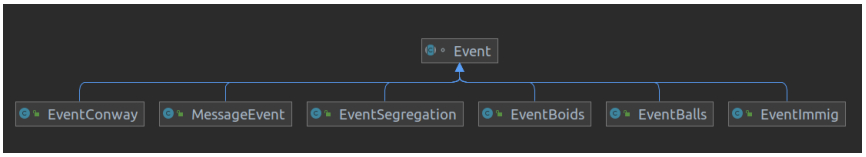


Figure 1 : Diagramme représentant l'héritage de la classe Plateau

Classe Event : classe abstraite elle est héritée par des sous-classes qui représenteront des évènements réels voir Figure 2



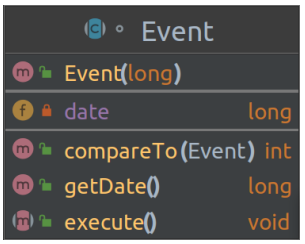
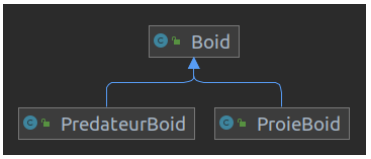


Figure 2 : Diagramme représentant l'héritage de la classe Event

- **Modèle d'essaims**

Classe Boid : Classe Boid qui définit les boids avec leurs paramètres respectifs (coordonnées, vitesse, limite_voisinage, …). Cette classe est utilisée pour créer plusieurs Boids et les insérer dans des listes de Boid.

Classe ProieBoid : Classe qui hérite de Boid. Les Proies sont des objets spéciaux de Boid. Elles ont des caractéristiques particulières dans cette classe, des caractéristiques de groupe.



Classe PredateurBoid : Classe qui hérite de Boid. Les Prédateurs sont des objets spéciaux de Boid. Ils ont des caractéristiques particulières dans cette classe.

Classe Simu_Boids : Classe qui simule le déplacement des Boids. Elle est utilisée pour créer, initialiser et afficher les Boids.

Classe TestBoids : Classe qui est utilisée pour créer la fenêtre gui et et appeler la classe Simu_Boids pour afficher les Boids.

Classe EventBoids : Classe qui execute l'évènement qui consiste à calculer les nouvelles coordonnées de tous les Boids en fonction des forces qui s'appliqueront sur lui.

Difficultés rencontrées :

Nous avons eu des difficultés en voulant créer différents types de Boids. Initialement, notre programme fonctionnait bien, mais pour différencier les groupes de Boid, nous avons utilisé l'héritage avec Boid. Chaque groupe de Boid est essentiellement un Boid, mais le problème est survenu avec le typage dynamique. Nous devions créer des listes contenant des Boids de différents types, chacun ayant des propriétés spécifiques. Pour résoudre cela, nous avons dû utiliser le typage Boid pour l'objet, mais l'instancier avec le constructeur spécifique à chaque type de Boid. Ainsi, même si nous appelons une méthode d'un type spécifique de Boid à partir d'un autre type, le programme recherche d'abord la fonction dans la classe Boid, puis utilise la version redéfinie dans la sous-classe correspondante. Cela nous permet d'utiliser la méthode appropriée selon le type de Boid.

Pour les tests de Boids, nous avons eu besoin de faire beaucoup de débogage au début puisqu'on s'était rendu compte, en générant aléatoirement des Boids, qu'ils avaient tendance à aller en haut à droite. En regardant à chaque tour le détail des forces, on s'est rendu compte que l'une d'entre elle nous emmenait toujours en haut à gauche.

2. Tests et résultats

- Automates cellulaires

Classe TestJeux_nom_du_jeu : permet la création du plateau, définition des positions des balles initiales et affiche de l'interface graphique

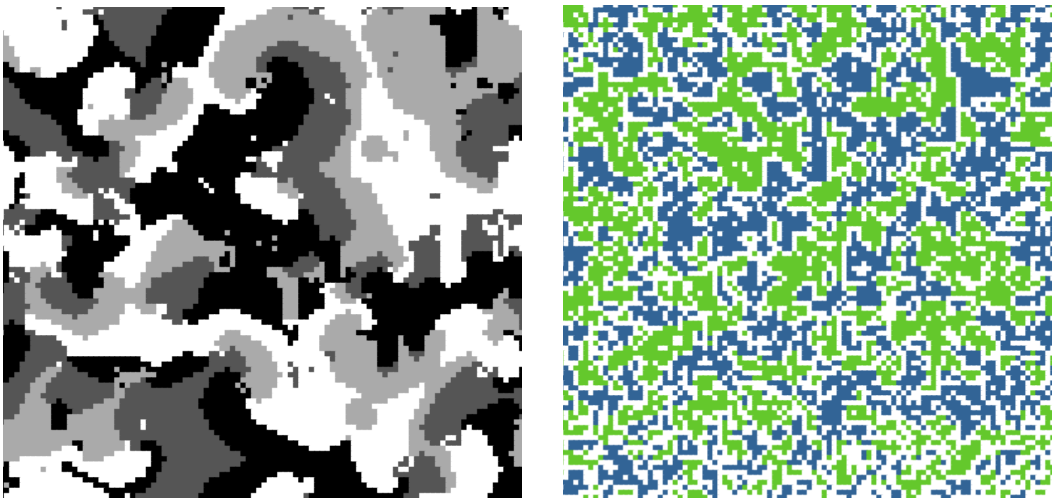


Figure 3 : Captures d'écran de l'affichage graphique de Immigration et de ségrégation

Pour conway nous avons pris des exemples sur internet afin de pouvoir les comparer aux nôtres et vérifier nos résultats. Ces tests nous permettaient de valider en partie notre programme sur des petits jeux de données.

On a ensuite amélioré notre code avec des tests aléatoires. Ce qui permettait de tester le fonctionnement de notre code avec de grosses quantités de données. En vérifiant pour

certain point que le comportement était bon. Ensuite nous avons fait tous nos tests de cette manière.

- Modèle d'essaims

Figure 4 : Captures d'écran de l'affichage graphique du modèle d'essaims

Nous avons créé un fichier de test qui génère aléatoirement deux types différents de Boid. Pour chacun d'entre eux, on génère aléatoirement ses coordonnées, sa vitesse (donc sa position).
En réponse à la question 11, on voit bien lors de nos tests que les prédateurs (les rouges) se déplacent bien vers les proies (les bleus). Ils ont été programmé de manière à ce qu'ils prennent en compte seulement la position de leurs proies et non des autres prédateurs.

Conclusion

Pour résumer nous avons réussi à implémenter le jeu de conway, le jeu de l'immigration, le modèle de schelling et le modèle d'essaims.

- Perspectives d'Amélioration :

Conway est un cas particulier d'immigration, nous aurions pu améliorer notre code en implémentant conway de façon à ce qu'il soit une classe fille d'immigration.
L'affichage des Boids aurait aussi pu être amélioré.

Pour tous les jeux d'automate cellulaire, on pourrait améliorer l'affichage parce que on s'est rendu compte que les cellules à gauche et en haut dépassent du cadre. On pense que c'est due à une mauvaise interprétation de la méthode Rectangle.

Exécution du Code

Si vous avez accès à IntelliJ IDEA : ouvrez le dossier en entier et exécutez le test correspondant au modèle que vous voulez tester.

Sinon à l'aide du terminal avec "./Makefile_nom-modele" après avoir avoir tapé la commande : "chmod +x Makefile_nom-modele"