Projet POO 2023-2024
Simulation Orientée-Objet de système multiagents

HCG

Introduction - TestBall

Le démarrage du projet et sa prise en main s'est fait à l'aide d'une simulation d'une groupe de balles qui bougent sur une fenêtre graphique.

Nous avons décidé de représenter les balles par des points dans un espace 2D puisqu'une classe point2D existait déjà en java et possédait toutes les méthodes qu'il nous était nécessaire d'utiliser pour le mouvement des balles et que la fenêtre graphique est un environnement 2D.

1 - Jeux cellulaires

Pour garantir une approche générique et maximiser la réutilisation du code, nous avons pris en considération la création de classes optimales. Dans cette optique, nous avons conçu une classe abstraite appelée ***CellGameEngine***. Cette classe est dotée d'un constructeur qui permet de spécifier la taille de la grille de jeu, le nombre initial de cellules, ainsi que le nombre d'états possibles pour chaque cellule.

Cette classe abstraite nous permet de définir la méthode firstGeneration() qui va initialiser les différentes simulations qui hériteront de ***CellGameEngine***. Pour avancer dans la simulation nous utilisons la méthode next() définie dans ***Simulable***, de même pour la méthode restart() qui nous permet de relancer une simulation.

En utilisant cette classe abstraite comme base, nous pouvons créer différentes implémentations de moteur de jeu de cellules en dérivant de celle-ci. Chaque implémentation spécifique peut ainsi fournir ses propres règles de jeu et fonctionnalités supplémentaires, tout en profitant des fonctionnalités de base déjà fournies par la classe abstraite.

Cette approche de conception nous permet de factoriser efficacement le code commun et de rendre nos classes plus flexibles et extensibles.

Nous modélisons la grille de cellule par la classe ***Grid***, d'instanciation unique pour chaque jeu et possédant un tableau à 2 dimensions de cellules. Les cellules

justement sont représentées par la classe ***Cell***, et possèdent entre autres un état courant et précédent ainsi qu'un nombre de voisins.

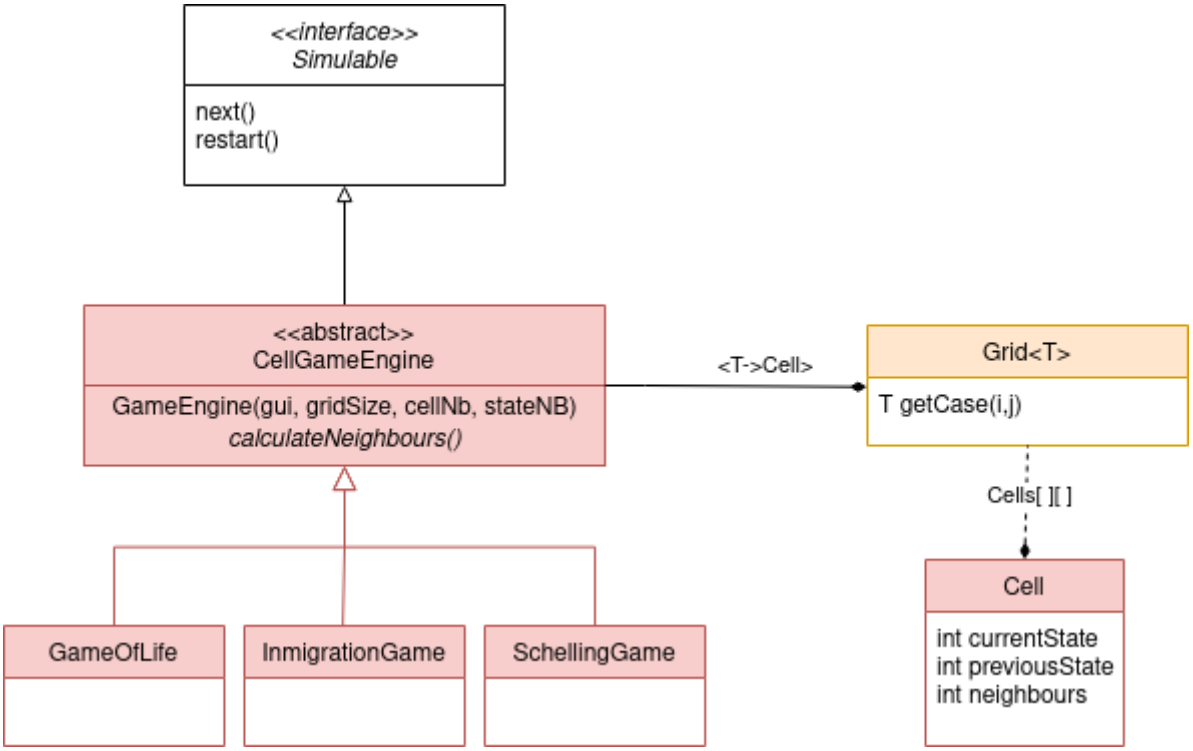


Figure I. Diagramme de classes de la partie jeux cellulaires

A- Le jeu de la vie

Le jeu de la vie, réalisable dans la classe ***GameOfLife***, peut être initialisé avec une dimension de grille (gridSize) et un nombre de cellules vivantes (cellNumber) qui auront des positions aléatoires dans la grille.

B- Le jeu de l'immigration

Le jeu de l'immigration, réalisable dans la classe ***InmigrationGame***, est similaire au jeu de la vie à la différence près que le nombre d'états possibles pour une cellule est déterminable par l'utilisateur.

Dans notre programme, il faut changer la valeur de *stateNumber* passée en paramètre du constructeur de l'objet ***InmigrationGameEngine***. Il est aussi toujours possible de choisir la taille du côté de la grille.

En lançant le programme, les cellules se stabilisent parfois en un modèle de spirale.

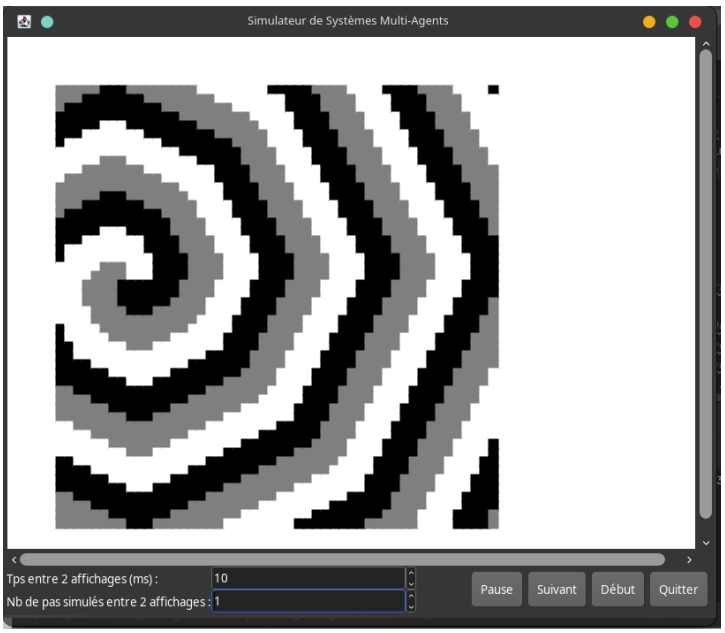


Figure II. Capture d'écran du jeu de l'immigration stabilisé

C-Le modèle de Shelling

Le modèle de Schelling, réalisable dans la classe **SchellingGame**, est lui aussi similaire au jeu de la vie et au jeu de l'immigration puisque nécessite lui aussi une grille et un nombre de cellules.

Cependant, nous avons besoin en plus de stocker la liste des cellules vides et celles colorées afin d'effectuer les déplacements aléatoires. Nous avons décidé d'utiliser des *HashSet* puisque la méthode empêche les dédoublements.

Enfin nous avons créé une liste permettant de stocker les différentes couleurs utilisées et générées de manières aléatoires puisque nous devons pouvoir accéder aux couleurs de manière efficace.

2- Les boids

Nous avons conçu des boids qui sont inspirés par l'idée des poissons. Les boids sont divisés en deux groupes, représentés par la classe BoidShoal (banc de boids).

Tout d'abord, il y a les poissons "gentils", qui sont passifs et cherchent simplement à survivre. Ils implémentent les comportements de cohésion, de séparation et d'alignement. Ces poissons sont des sardines de différentes couleurs et tailles. Ensuite, il y a les poissons prédateurs, les requins, qui chassent les sardines. Leur vitesse augmente à mesure qu'ils ont faim.

La représentation graphique des poissons est réalisée en utilisant la méthode "paint" de la classe abstraite Boid, qui implémente l'interface GraphicalElement.

Les poissons évoluent dans un environnement qui est sujet à des changements. Par exemple, à la manière d'un fond marin régi par les courants, la classe FlowField décrit le champ vectoriel de l'espace qui influence le déplacement des boids. Ce champ vectoriel utilise la classe Grid définie précédemment pour créer un tableau en deux dimensions de vecteurs. Nous avons décidé de rendre la classe Grid générique, de sorte qu'il est possible de choisir le type d'objet contenu dans les cases.

Dans le cas du FlowField, les objets contenus dans les cases sont des Vector2, une classe spécialement créée pour modéliser un vecteur en deux dimensions et toutes les opérations associées. Le champ vectoriel oscille entre une grille de vecteurs aléatoires et des vecteurs orientés vers le centre, selon un intervalle aléatoire.

Par défaut, nous avons choisi d'inclure un seul requin, qui est relativement gros, mais il est tout à fait possible d'envisager la possibilité d'ajouter plusieurs requins. Le jeu se crée en spécifiant uniquement le nombre de poissons pacifistes en tant que paramètre de l'objet FishBoidsEngine.

La manière dont la partie boid est modélisée permet d'imaginer facilement l'ajout d'autres espèces de poissons ou de changer de paradigme d'entités (par exemple, passer à des insectes).

3- L'Event Manager

L'**EventManager** est une classe qui permet de gérer différents événements et les exécuter dans un ordre prédéfini.

Elle regroupe 2 données: une date qui est initialisée à 0 et une File de Priorité (PriorityQueue) d'Event.

De plus, elle possède deux méthode principales:

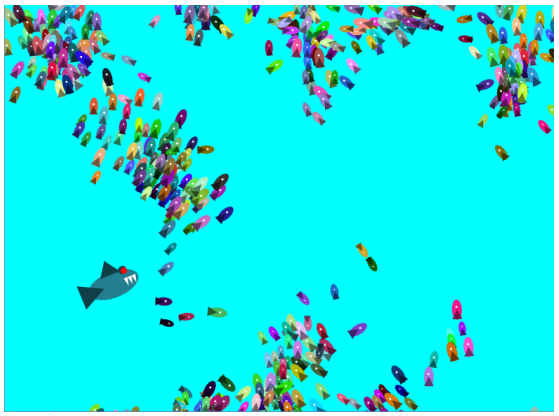
-**next** qui permet d'exécuter tout les événement qui doivent l'être à la date actuelle et d'incrémenter la date

-**restart** qui remet à zéro la date et vide la PriorityQueue.

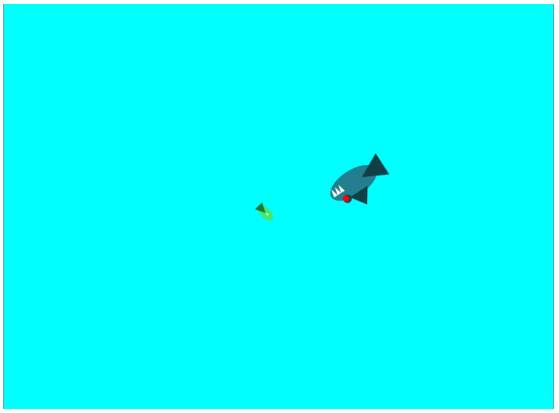
Qu'est-ce qu'un Event. Un Event est une classe abstraite que l'on a créé représentant la forme que devrait prendre un évènement, c'est-à-dire, une date d'exécution et une méthode exécuter pour effectuer son évènement.

Pourquoi avons-nous choisi de prendre une file de priorité? Nous avons fait ce choix puisqu'il nous permet de pouvoir regarder et donc exécuter que les éléments qui doivent s'effectuer à la date actuelle, sans avoir un coût d'ajout trop élevé.

A- Annexes

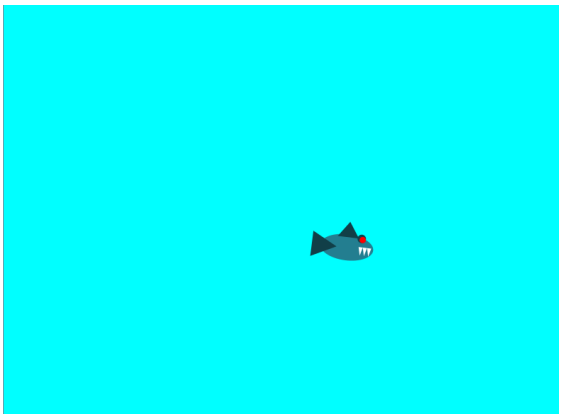


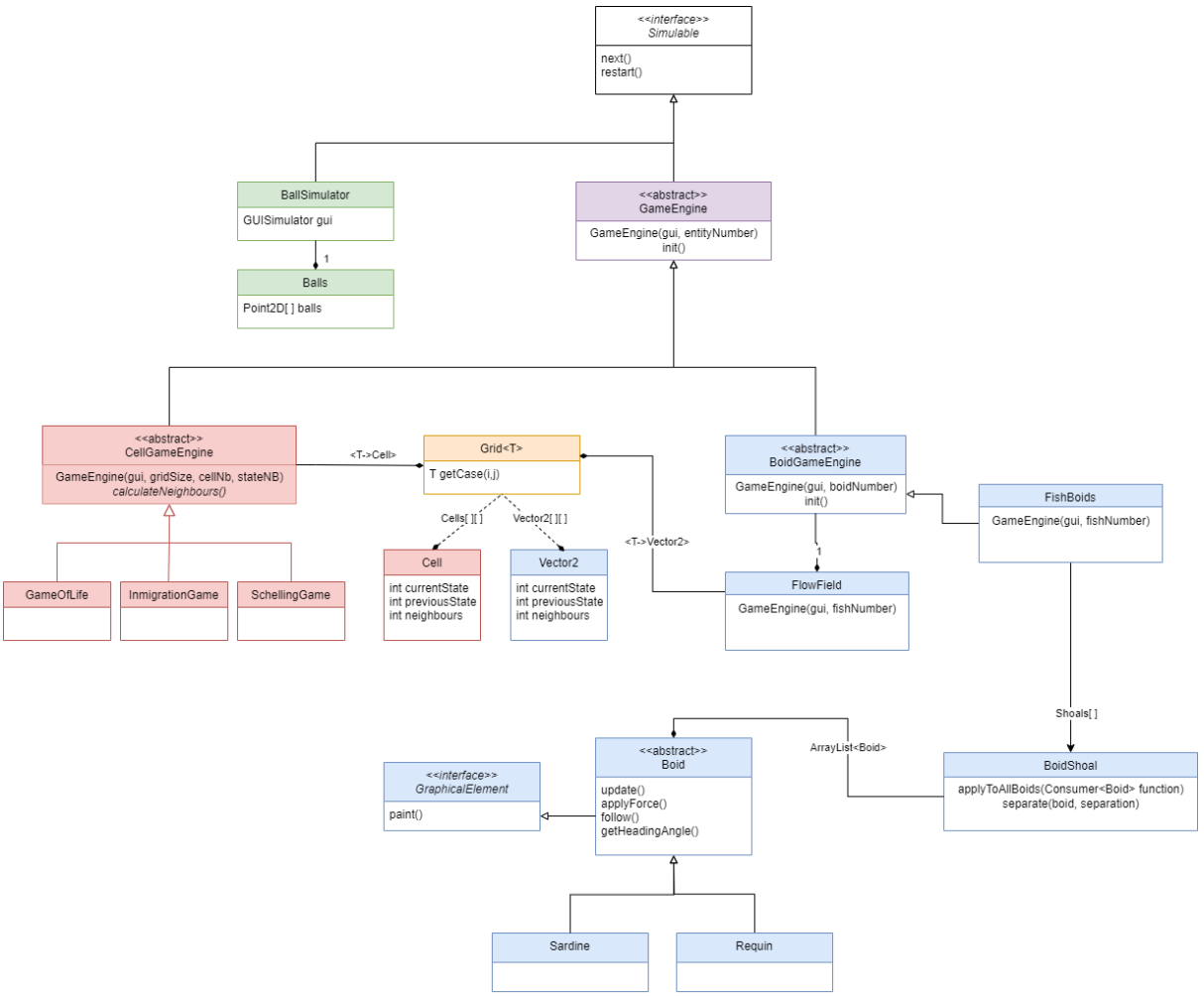
Annexe I. Festin



Annexe II. Instant avant désastre

Annexe III. Solitude





Annexe IV. Diagramme UML de classes du projet