Compte-rendu du TPL de Programmation Orientée Objet

Dans ce TPL, nous avons cherché à modéliser différents systèmes collectifs liés à l'intelligence artificielle, appelés les systèmes multiagents. Il s'agit là d'un système d'agents autonomes qui sont capables d'interagir entre eux, ainsi qu'avec l'environnement dans lequel ils se trouvent. Ici, nous allons réaliser une simulation de différents modèles d'agents cellulaires, à savoir un système de balles, des automates cellulaires (Game Of Life, immigration et modèle de Schelling) et des boids.

Pour réaliser ces simulations, nous avons utilisé une bibliothèque d'interface graphique utilisateur (GUI) fournie par les professeurs.

1ère partie : Les balles

Dans cette première partie, nous modélisons un système de balles qui se déplacent en ligne droite (ici en diagonale) et qui rebondissent contre les parois de la fenêtre. Pour cela, on crée deux classes : une classe *Ball*, qui définit l'objet en lui-même, et une classe *Balls*, qui définit et donne le comportement d'un groupe de balles dans l'environnement graphique.

Pour implémenter correctement la classe *Ball*, nous avons créé une classe *Vector2d* qui représente un vecteur dans le plan, et qui est utilisée pour modéliser la position et la vitesse d'une balle. Cette classe nous servira très souvent pour les autres simulations.

La classe *Balls* représente un groupe d'un certain nombre d'objets *Ball* dont la taille et la vitesse sont définies, et dont la position est aléatoire dans la fenêtre. La méthode *translate* permet de déplacer toutes les balles dans la même direction, en prenant en compte les rebonds sur les parois. La méthode *reInit* replace les balles à leur emplacement initial.

BallsSimulator permet d'afficher un objet *Balls* dans l'interface graphique et de gérer son évolution dans le temps à l'aide de l'évènement BallMoveEvent qui effectue la translation au temps *t* et crée un nouvel évènement pour le temps *t+1*.

La classe de test *TestBalls* permet de s'assurer du bon fonctionnement des classes *Ball* et *Balls* en affichant les coordonnées des balles d'un objet *Balls* au cours du temps et après réinitialisation. La classe *TestBallsSimulator* effectue ce test dans l'interface graphique.

2ème partie : Les automates cellulaires

Après avoir codé le système précédent, nous nous sommes tournés vers l'implémentation d'un système d'automates cellulaires. Pour cela, comme demandé, nous avons codé une grille contenant des cellules possédant chacune leur propre état, et qui évoluent toutes en fonction de leurs voisins. Nous nous sommes alors intéressés à 3 exemples d'automates cellulaires. Pour implémenter ces automates, nous avons d'abord mis en place deux classes, une classe *Cell* et une classe *Grid*.

La classe *Cell* ne contient que l'état de la cellule car c'est la seule valeur propre à une cellule. La classe *Grid* est une classe abstraite représentant une grille de cellules ayant une taille définie et une taille de cellule définie. La méthode *getNeighbors* permet d'obtenir la liste des cellules voisines de la cellule étudiée. Les méthodes *setState* et *getState* se trouvent dans la classe *Grid* pour que l'état des cellules soit accessible et modifiable à partir de leurs coordonnées.

- Le Jeu de la Vie de Conway :

Pour implémenter ce jeu dont vous trouverez les règles sur le sujet dans notre simulateur, nous avons donc implémenté une classe *GoLGrid*, qui met en place la grille de cellules en définissant son état (qui est un entier) en fonction d'un booléen. La méthode *aliveNeighbors* permet de compter le nombre de voisins vivants d'une cellule.

GoLSimulator permet de simuler graphiquement le jeu de la vie à partir d'un état aléatoire. Cette fois-ci, et pour tous les simulateurs d'automates cellulaires, la méthode init appelle la méthode initGrid de la grille correspondante, et génère un état aléatoire différent à chaque appel.

- Le Jeu de l'Immigration :

Pour implémenter cet automate cellulaire dans le simulateur, nous avons créé la classe *ImmGrid*, héritant de la classe abstraite *Grid*, et donnant l'accès à chaque cellule de la grille, son état actuel ainsi que l'état maximal atteignable avant de revenir à l'état initial. Nous avons aussi implémenté une classe *ImmEvent* pour cette simulation, permettant d'actualiser les différentes cellules en fonction de leur voisinage à chaque étape t, et de créer l'étape t+1 .

- Le modèle de ségrégation de Schelling :

Pour implémenter cet automate cellulaire, nous avons créé une classe *SchellingGrid* de la même manière que les grilles précédentes. Cependant, à la différence des deux automates cellulaires précédents, il n'y a pas la notion de changement d'état d'une cellule selon un cycle défini, mais le besoin de savoir quelles cellules de la grille sont vides afin de pouvoir déplacer les "familles" selon les règles définies. Pour cela, on définit une méthode *getVacantSpots* dans la classe *SchellingSimulator*, qui nous permet de récupérer toutes les cellules vacantes de la grille, et d'y déplacer les familles qui déménagent. De plus, nous avons forcé l'ajout d'au moins une cellule vacante à l'initialisation de la grille, afin que la simulation ne soit pas bloquée dès le début. Enfin, nous considérons que dès qu'une famille déménage, la cellule devient inhabitée, et une autre famille peut donc venir s'y installer au même tour (nous ajoutons donc cette cellule dans la liste des cellules vacantes).

On remarque alors que pour un seuil supérieur ou égal à 3, la ségrégation est quasi-systématique et très importante.

3ème partie : Les Boids

Pour implémenter les boids, nous avons créé une classe abstraite *Boid*, qui porte tous les attributs communs à tous les boids (qu'ils soient standards, proies ou prédateurs), et définit les méthodes communes à tous les boids (détermination des voisins à partir de l'angle et de la distance de vue, forces définies par les règles). De plus, un boid est représenté graphiquement par un objet de la classe Triangle, que nous avons créé en lui donnant un Vector2d correspondant à la position exacte du boid qui est au centre du triangle, une base et une hauteur (ou une taille), afin de générer les points correspondants aux arêtes du triangle. La méthode rotate permet, comme son nom l'indique, d'appliquer une rotation à ce triangle, avec pour paramètre l'angle de rotation (calculé par la méthode angle de la classe boid).

Ensuite, pour étudier le comportement d'un essaim sans autre contrainte que celles exprimées juste au-dessus, nous avons créé une classe *CommonBoid* qui hérite de la classe *Boid*, et qui redéfinit la méthode *addForces* (qui détermine le mouvement du boid) en ajoutant les 3 forces des règles.

Nous avons par la suite mis en place un système de prédation avec deux essaims de boids, l'un étant la proie de l'autre et cherchant à fuir ces derniers, sachant que les prédateurs vont poursuivre les proies. Le système de proie/prédateur fait donc appel à deux nouvelles classes *PredatorBoid* et *PreyBoid*, héritant toutes deux de *Boid*, et définit une nouvelle force par classe (force de traque pour les prédateurs, force de fuite pour les proies).

Enfin, on crée deux classes *CommonBoidSimulator* et *PreyPredatorSimulator*, qui permettent de vérifier le bon fonctionnement des deux classes associées. Nous avons aussi défini un système de "rebond" sur les bords de la fenêtre de simulation, permettant à cette dernière de durer indéfiniment. Il se trouve dans la classe *BoidEvent* qui gère aussi les événements dans le temps. Les classes *TestBoidSimulator* et *TestPredPreySimulator* servent à tester les deux classes précédentes dans l'interface graphique.

4ème partie : La gestion des événements

Afin de pouvoir gérer les différents évènements qui peuvent se passer dans les différents systèmes multiagents, nous avons également intégré un gestionnaire d'événements permettant d'organiser les évènements qui arrivent en fonction de leur date d'arrivée, et sont triés dans une table de hachage selon la date. Nous avons alors créé une classe *EventManager*, qui prend comme attributs la table de hachage ainsi qu'un long *CurrentDate* qui correspond à la date actuelle. Nous avons également implémenté plusieurs méthodes permettant de manipuler le gestionnaire

d'événements (ajout d'un événement, redémarrage, suppression d'un événement de la liste, etc).

Après avoir défini le gestionnaire d'événements, nous avons créé une classe abstraite *Event* qui prend comme attributs une date, et possède une méthode abstraite d'exécution. Ainsi, pour chaque événement correspondant à une simulation précise, on crée une classe d'événements qui hérite de cette classe abstraite, à laquelle on va ajouter des attributs propres à la simulation (à savoir la grille et les états pour les automates cellulaires, ou l'ensemble des boids ou balles pour les autres simulations). La méthode abstraite est donc réécrite (via un Override) pour mettre en place les règles d'évolution des systèmes (par exemple, pour le jeu de la vie la méthode *execute* de la classe *GoLEvent* nous donne la nouvelle grille après évolution d'un cran, en prenant les règles de reproduction et de mort définies dans la partie consacrée à cette simulation).

5ème partie : Tests effectués

Pour vérifier si nos différentes classes Java fonctionnent de la manière souhaitée, nous avons implémenté différents tests pour chacune des parties du projet (pour vérifier si l'interface graphique fonctionne, si les événements sont gérés correctement par le gestionnaire, et si chacune des simulations donne le résultat escompté). Par exemple, pour les boids, nous avons implémenté deux tests pour vérifier si les boids se comportent normalement, un test standard avec un seul essaim qui se déplace dans l'interface graphique, et un autre où deux essaims sont présents, un premier étant la proie du second, et qui cherche donc à le fuir. De plus, pour le système de balles décrit en première partie, nous avons implémenté des balles disposées aléatoirement sur la fenêtre, qui se déplacent en diagonale et qui rebondissent sur les bords de la fenêtre. Enfin, pour les automates cellulaires, nous avons généré aléatoirement la grille et la position des différentes cellules puis lancé les simulations afin de vérifier si les simulations fonctionnent correctement. Tous ces tests sont modifiables à votre guise en changeant les constantes de ces classes de test, puisque ces valeurs sont retransmises aux classes de calcul des simulations.

Partie 6 - Coding style

Le principe d'encapsulation et de masquage des attributs a bien été pris en compte. Cependant certains attributs sont publics, mais cela est justifié. Par exemple, les vecteurs position et vitesse d'une balle sont publics car leur accès généralisé est nécessaire, c'est d'ailleurs pour cela que les attributs x et y de la classe Point de Java sont publics.
L'idée est la même pour les vecteurs position et vitesse des Boids, ainsi que leur couleur, et pour les attributs x et y des Vector2d.
Rendre ces attributs privés en ajoutant des mutateurs/accesseurs aurait le même effet que de les laisser publics.
L'attribut state de la classe Cell est public lui aussi, car chaque grille doit pouvoir y accéder et le modifier. La modification de cet état passe ensuite uniquement par les classes héritant de Grid grâce aux fonctions getState et setState.