Rapport projet de programmation orientée objet

Simulation de systèmes multiagents

Équipe 81

12 novembre 2021

Sommaire

Introduction

Dans l'optique de respecter la taille maximal de 4 pages pour ce compte rendu, nous avons décidé de ne décrire que les méthodes que nous avons jugées nécéssaire d'expliciter. Pour plus d'information, chaque méthode est décrite avec précision en commentaires des fichier .java .

Le guide d'utilisation des tests effectués et des modifications qui peuvent être ajoutés pour modifier les simulations selon vos choix sont explicités dans le fichier README du projet. Il est donc conseillé de se cantonner à des modifications des paramètres mis en évidence dans ce fichier, pour garantir le fonctionnement optimal des programmes.

II) Simulation de balles

Dans cette partie, nous avions pour mission de concevoir un programme qui simulerait le mouvement d'un groupe de balles pour appréhender l'environnement graphique GUISimulator. Pour cela nous avons séparé l'aspect calculatoire et l'aspect graphique de la simulation avec une première classe, c'est une technique que nous appliquerons dans l'ensemble du sujet puisqu'il permet d'avoir un code plus clair et plus facile à appréhender.

DESCRIPTION DES CLASSES ET METHODES UTILISEES :

- Class Balls : modélise un ensemble de points et calcule les déplacements possibles
  - public Balls() : renvoie une liste vide de balles.
  - public LinkedList<Point> getpoint() : renvoie la liste des positions de toutes les balles.
  - public void add(Point p) : ajoute une balle p à la liste des positions des balles, avec un sens par défaut de 1 selon x et 1 selon y, et de position initiale identique à p.
  - public void translate(int dx, int dy, int xmax, int ymax) : translate l'ensemble des points de dx selon x et dy selon y en prenant en compte les rebonds possibles sur le parois en 0 et max sur x, et 0 et ymax sur y.
  - public void reInit() : repositionne tous les points à leurs coordonnées d'origine.
- Class BallsSimulator : Affichage graphique d'un ensemble de points dans un même simulateur graphique. Réalise donc l'interface Simulable et possède un attribut de type GUISImulator pour afficher les résultats dans le simulateur
  - public void next() : calcule la position suivante de chaque balle dans la simulation, efface leurs ancienne position et affiche la nouvelle.
  - public void restart() : charge la position initiale de chaque balle dans la simulation, efface leur ancienne position et affiche la nouvelle.

DESCRIPTION DES TESTS :

Le test TestBalls permet de s'assurer du bon fonctionnement de la partie calculatoire de la simulation en testant toutes les fonctions de la méthode Balls.

Le test TestBallsSimulator lance la simulation de 3 points se déplaçant de 10 en 10 sur x et y et rebondissant sur les bords de la fenêtre graphique.

III) Automates cellulaires

Dans cette partie, nous avions pour mission de concevoir une liste de programmes qui simuleraient des rectangles changeant d'état en fonctions de conditions différents selon le jeu de l'immigration (décrit dans les fichiers contenant "ImGame dans leurs noms "), le modèle de Schelling (décrit dans les fichiers contenant "Schelling dans leurs noms ") et le jeu de la vie de Conway (décrit dans tous les autres fichiers avec "Rectangles" dans leurs noms).

Pour cela nous avons factorisé le code en réalisant le jeu de la vie puis en réutilisant la classe Rectangles pour les deux autres simulations.

DESCRIPTION DES CLASSES ET METHODES UTILISEES :

- Class Rectangles : modélise un ensemble de rectangles ayant un état (symbolisé par un entier)initial, passé et courant.
  - public public Rectangles(int HeightRectangle, int WidthRectangle, int Heightgui, int Widthgui)() : Renvoie une liste de rectangles de taille HeightRectangle*WidthRectangle permettant de remplir la fenêtre graphique de taille Heightgui*Widthgui.
  - public void newState(): calcul du nouvel état courant selon l'état passé des voisins de la case
  - public void reInit() : chaque rectangle reprend son état initial (passé comme courant)
  - Class RectanglesImGame : hérite de la classe Rectangles en ajoutant **nbStates** le nombre d'états que peut prendre un rectangle, et **StatesVoisins** un tableau contenant le nombre de voisins ayant pour état 1+état_de_la_case_du_tableau. Et redéfinit la méthode nextState() pour respecter les règles du jeu de l'immigration
  - Class RectanglesSchelling : hérite de la classe Rectangles en ajoutant nbColor (similaire à nbStates) et vacants une file de rectangles n'étant pas occupés. Redéfinit la méthode newState() pour respecter les règles du modèle de Schelling
    - private void demenagement(int ligne, int colonne) : déménage le rectangle ligne colonne à une place vacante dans vacants si la file n'est pas vide.
- Class RectangleSimulator : Affichage graphique d'un ensemble de rectangles dans un même simulateur graphique. Réalise donc l'interface Simulable et possède un attribut de type GUISImulator pour afficher les rectangles de tailles HeightSize*WidthSize donnés en attributs dans le simulateur et un attribut Rectangles pour calculer l'état, la taille et la position de chaque rectangles.
  - setState1(int l, int c): met l'état initial du rectangle ligne l colonne c à l'état 1 et affiche la modification
  - public void next() : calcule l'état suivant de chaque rectangle dans la simulation, efface leurs ancienne position et affiche la nouvelle.
  - public void restart() : charge l'état initial de chaque rectangle dans la simulation, efface leurs ancienne position et affiche la nouvelle.
  - Class RectanglesImGame : généralise la simulation avec nbEtats (donnés en attributs du constructeur) et utilise la méthode setState(int l, int c, int n) au lieu de setStats1 pour mettre le rectangle ligne l colonne c à l'état n. Redéfinit les classes next() et restart()pour s'appliquer à des objets de classes RectanglesImGame.

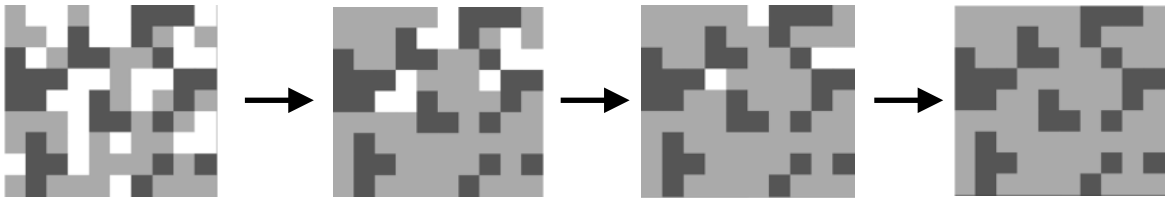
- Class RectanglesImGame : Même utilité de RectangleImGame mais pour s'appliquer à des objets de classe RectanglesSchelling.

DESCRIPTION DES TESTS :

Le test TestRectangleSimulator lance la simulation du jeu de la vie de Conway. Le jeu fonctionne et il est possible de modifier l'état initial de chaque rectangle.

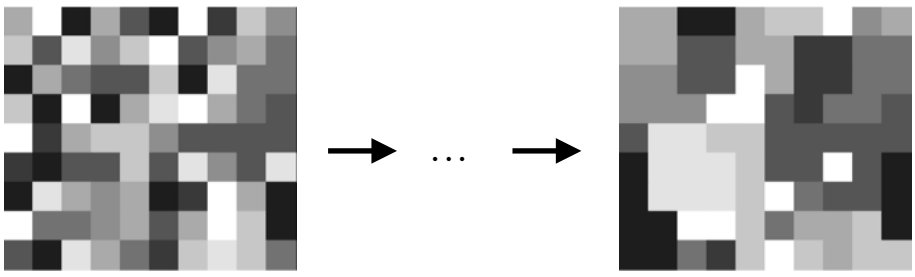
Simulation du jeu de la vie de Conway

Le test TestRectangleImGameSimulator lance la simulation du jeu de l'immigration. Le jeu fonctionne et il est possible de modifier le nombre d'état maximal et l'état initial de chaque rectangles.



Simulation du jeu de l'immigration

Le test TestRectanglesSchellingSimulator lance la simulation du jeu de l'immigration. Le jeu fonctionne et il est possible de modifier le nombre d'états maximal, le seuil K de voisins et l'état initial de chaque rectangles.



Simulation du modèle de Schelling

IV) Simulation d'un système multiagents

Dans cette partie, nous avions pour mission de concevoir un programme qui simulerait le comportement d'un groupe d'individus. Ainsi, il était question de réutiliser nos connaissances issues des simulations précédentes, pour créer le cadre de la simulation tout en y rajoutant une composante : L'enjeu supplémentaire de cet exercice était de simuler l'influence des forces qui régissent l'interaction entre les individus.

Pour modéliser les forces, nous nous sommes basés sur l'ouvrage *the nature of code* écrit par Daniel Shiffman, et notamment le chapitre consacré aux systèmes multiagents. Le principe est toujours le même : lorsqu'une force extérieure agit sur le boid (on appelle les individus ainsi), cette force s'ajoute à l'accélération. La force résultante appliquée sera alors un vecteur orientant le déplacement de l'individu.

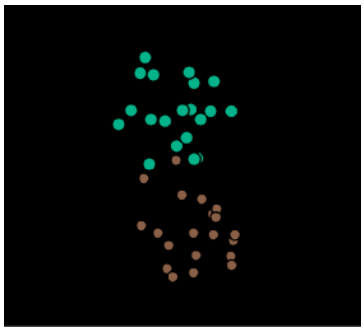
DESCRIPTION DES CLASSES ET METHODES UTILISEES :

- Class Boid : modélise un seul individu.
  - Public void limit() : limite la norme du vecteur de vitesse
  - Public float dist(Boid b) : renvoie la distance entre le boid considéré et le boid donné en paramètre
  - Public void update() : à chaque frame de la simulation (chaque temps d'actualisation de la position), la vitesse est ajoutée à l'accélération. En effet la vitesse correspond à la distance parcourue par unité de temps. L'accélération, quant à elle, modélise le gain de vitesse par unité de temps.
  - Pulic void applyForce(Vecteur force) : l'accélération correspond à l'ensemble des forces subies par un objet divisé par la masse de l'objet (qui est unitaire ici). Ainsi, une force appliquée sur l'objet se traduit par la somme de la force et de l'accélération.
  - Public Vector seek(Vecteur target) : sert à diriger le boid vers une cible
  - Public Vecteur separate(ArrayList<Boid>) : sert à séparer le boid considéré des autres boids, afin que les boids restent à une certaine distance les uns des autres.
  - Public vecteur cohesion(ArrayList<Boid> listBoid) : assure la cohésion du groupe, c'est à dire la tendance des individus à suivre leurs voisins proches.
  - Public Vecteur align(Boid> listBoid): garantit que les individus adaptent leur allure à celle du groupe.
  - void flock(ArrayList<Boid> boids) : applique les trois forces précédemment évoquées (cohésion, séparation et alignement) au boid considéré. Lorsque cette méthode est appelée à chaque pas de temps, les forces de cohésion d'alignement et de séparation, calculées à partie de la position des boids voisins passés en paramètre sont appliquées au boid. Celles ci sont pondérées par un coefficient.
- Class Flock : modélise un groupe d'individus. Peut en ajouter (méthode add(Bird b)) et tous les supprimer (méthode clear())
- Class Event : Correspond à un événement avec une date et un pasDeTemps qui correspond au temps au bout duquel l'évènement se produit à nouveau.
- Class EventBoids : Évènement qui se réalise sur un attribut Flock boids et qui s'affiche dans GUISimulator gui.
  - public EventBoids execute() : met à jour la position des boids dans gui et renvoie un nouvel évenement ayant lieu à l'instant t+pasDeTemps

- Class EventManager : gestionnaire d'événements qui ajoute des événements à un arbre d'événements initiaux (listEventInit) et un arbre d'événements (listEvent) puis les trie grâce à une classe Comparator selon leurs date d'occurence. Appel ainsi les événements en fonction de sa date actuelle currentDate.
  - public void addFirstEvent(Event e) : Ajoute e à listEvent et listEventInit dans l'ordre croissant de date.
  - public void addEvent(Event e) : Ajoute e à listEvent dans l'ordre croissant de date.
  - public void next() : augmente de 1 currentDate, appel l'ensemble des événements ayant eu lieu avant cette date et ajoute à listEvent les nouveaux événements crées par la class EventBoids.
  - public void restart() : remet à 0 currentDate, remplace tous les événements de listEvent par les événements initiaux présents dans listEventInit.
- Class BoidSimulator : simulateur chargé de réaliser l'affichage graphique de la position d'une liste de Flocks dans un même simulateur graphique en suivant les ordres de l'EventManager. Réalise l'interface simulacre et redéfinit les méthodes next() et restart(). Possède donc un attributs GUISimulator, EventManager, une liste de Flock et une liste avec leurs positions initiales.

DESCRIPTION DES TESTS :

Le test TestBoidSimulator lance d'un modèle d'essaims. La simulation fonctionne, il est possible de modifier le nombre de Flocks et le nombre de Boids maximal par Flocks. Le nombre de boids par Flock et donc aléatoire entre 0 et nbBoids mais il est possible de le fixer à nbBoids en remplaçant *size =...* par *size=sizemaxOfFlock;*
(ligne 27 fichier BoidSimulator.java)



Interactions entre les boids de deux essaims