Rapport Du TPL de POO - Simulation de Systèmes Multiagents:

Equipe 25 Sur Teide

November 19, 2021

1 Introduction

En définissant un agent comme une entité autonome qui agit dans un environnement déterminé, Un système multi-agents sera un système composé d'un ensemble d'agents actifs, interagissant avec leurs voisins et avec leur environnement selon certaines règles.

Ce principe offre donc plusieurs possibilités intéressantes de modélisation de grands groupes d'entités autonomes, à l'instar des sociétés humaines et animales... Ainsi, il trouve des applications dans plusieurs domaines comme le les sciences humaines, la télévision, les jeux vidéos...

On s'intéresse donc dans ce sujet à la simulation de ces systèmes multi-agents.

2 Jouons à la Balle

2.1 L'hiérarchie des classes:

Les diagrammes dans le sujet représentent l'hiérarchie des classes que nous avons adopté pour toute la partie en relation avec les balles.

Du coup, on remarque qu'on a bien respecté la structure vers laquelle le sujet nous a guidés, en créant :

- La Classe Balls qui est responsable de la gestion de tout ce qui relève de l'entité "Liste de Balles", en gérant son effectif, la translation de ses éléments, l'élaboration d'une chaîne de caractère appropriée représentative de son état...,
- la Classe EventBalls qui est représentatif d'un évènement de la simulation. Implémentant la Classe mère Event, elle contient bien une méthode next qui translate les élements de la liste Balls (nouvel état), et une méthode restart qui nous renvoit vers l'état initial,
- La Classe BallsSimulator qui vient implémenter l'interface "Simulable" et qui est responsable de créer et gérer la Simulation d'une entité de Balls, et qui est contrôlée par le gestionnaire d'évènements créé suite aux instructions du sujet,

2.2 Tests Effectués En cette Partie:

Pour tester Cette Classe, nous avons utilisé Un simple test : TestBallsSimulator, qui trace un nombre à déterminer de balles en des positions quelconques et les fait translater de distances dx et dy manipulables (ici en l'occurence pris égaux à 10, 10).

2.3 Rebondissement:

Afin de gérer le rebondissement des balles lorsqu'elles atteignent les parois de la fenêtre, nous avons choisi de ne pas modifier les coordonnées des balles en question, mais de les conserver afin de garder une certaine intégrité des calculs prochains si jamais ils impliquent les coordonnées x et y de la balle. On a par contre introduit de nouveaux paralètres, définissant la "ScreenLocation", i.e : la position apparente de la balle sur l'écran, qui se calcule par de formules simples (figurant sur la méthode next sur BallsSimulator)

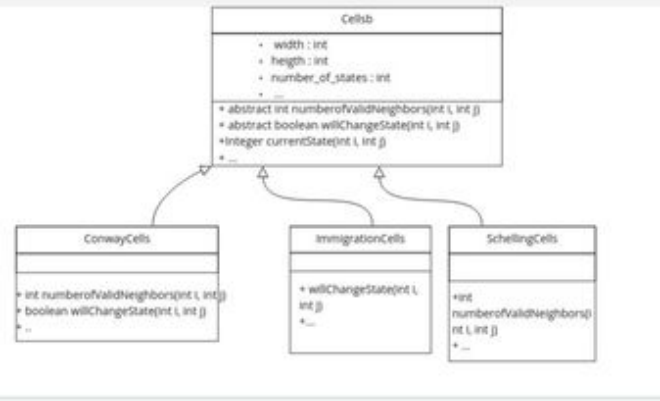

Figure 1: Hiérarchie pour La Partie des Automates Cellulaires (devrait être en paragraphe suivant)

2.4 Conclusion:

Grosso modo, Cette partie sert de "blueprint" de ce qu'on devra faire plus tard. Sa simplicité cache tout le principe derrière l'hiérarchie des classes dans les parties prochaines... On retrouvera surtout la structure : Entité/ Event/ simulateur/ Test dans tout le reste du projet.

3 Les Automates Cellulaires:

3.1 Hiérarchie Et Conception:

Les trois jeux de la partie 3 ont beaucoup en commun. Ainsi, le choix de la création d'une classe abstraite Cellsb dont on héritera pour dessiner la grille/les cellules de chacun des jeux s'est vite imposé.

On a définit dans cette classe 3 constructeurs (vu que chaque jeu a ses propres caractéristiques), des méthodes qu'on utilisera dans les trois jeux ainsi que des méthodes abstraites qui seront définis dans chacun des cas.

En Détails : On a décidé d'utiliser une liste de couleurs et deux dictionnaires qui identifieront chaque cellule (i, j) par la chaîne de caractère i + "," + j et dont la clé sera un entier i qui représente l'index de la couleur de la cellule dans la liste des couleurs. Que ce soit pour le premier jeu de Conway (les états vivant et morts sont codés respectivement par 1 et 0) ou dans le jeu de Schelling où les états vacants ne sont autres que des cellules associée à l'entier 0 (qui lui est l'indexe de la couleur blanche), ce choix de structure a permis une bonne factorisation du code.

Mais pourquoi aurons-nous besoin de deux dictionnaires? La réponse se trouve dans la nature même du caractère des cellules. En effet, chaque cellule à l'instant t ne dépend que de l'états de ses voisins à l'instant t-1. Un deuxième dictionnaire pour sauvegarder les états des cellules à l'instant t-1 et éviter tout problème que pourrait engendrer une modification brusque des états des cellules voisines d'une cellule donnée.

Trois classes ont hérités de la classe abstraite Cellsb, à savoir: ConwayCells, ImmigrationCells, SchellingCells. On a définit dans chacune de ces classes les méthodes suivantes:

- numberofValidNeighbors(): retourne le nombre de voisins qui satisfont les règles de chaque jeu;
- willChangeState(): retourne un booléen selon le "numberofValidNeighbors";
- changeState(): change l'état (l'indexe de la couleur) de la cellule (i, j) et la dessine;

Ensuite, il faut simuler les jeux. Et pour cela, trois classes ont été créées implémentant l'interface Simulable où l'on a définit les méthodes next() et restart()) ainsi que notre choix de position initiale des cellules qui prendra en considération un maximum de cas possibles.

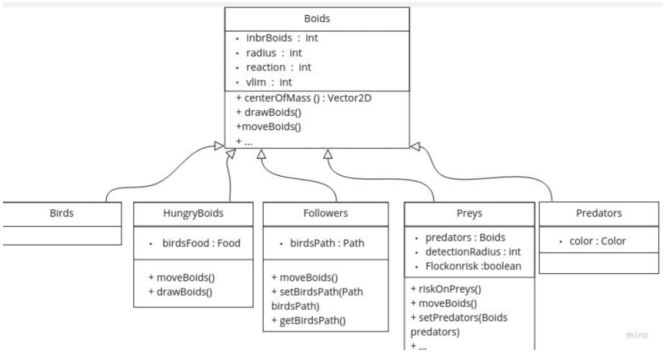

Figure 2: Hiérarchie pour cette partie

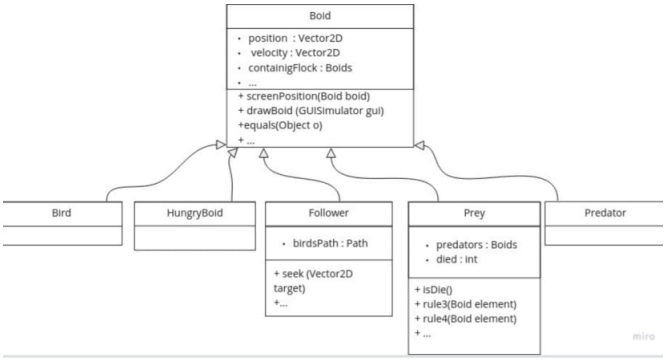

Figure 3: Hiérarchie pour cette partie

3.2 Les Tests:

On a réalisé un test pour chacun des Jeux (voir images des résultats en Annexe)

Après différents tests sur la simulation de Conway, on a remarqué que (à part dans le triste cas où toutes les cellules finissent par mourrir), le jeu converge toujours vers des motifs réguliers. (Voir annexe)

En ayant pris différentes valeurs du seuil et du nombre d'habitations, nous avons souvent ségrégation lorsque le seuil prend la valeur 3. (4 lorsque le nombre d'habitations est faible).

4 Les Boids:

4.1 Hiérarchie Et Classes:

Encore une fois, le diagramme suivant représente l'hiérarchie des classes utilisées dans cette partie:

Ainsi, la classe Boid est une classe qui est abstraite, représentative d'une seule entité, et liée à la classe Boids qui représente l'essaim en entier. Ces deux classes abstraites seront par la suite implémentées par les classes :

- Birds qui est assez générique, et dont les éléments sont régis par les 3 règles principales définies par Reynolds, et qui assurent L'alignement, la cohésion, et la séparation.
- HungryBirds qui représente une classe de boids qui, en plus de former un essaim, ils ont faim, et sont attirés par des particules de nature Food. Pour celà on a introduit une nouvelle règle qui se base sur la quête (implémentée par la fonction seek).
- Followers qui sont un groupe de boids qui, en plus de former un essaim, qui suit un chemin de nature Path. Pour Garantir ce fait de proximité dela trajectoire dessinée, on utilise encore une fois la notion de quête d'un point de la trajectoire (grâce à la fonction seek), à chaque fois que le boid essaie de s'éloigner.

- Prey et Predator qui sont implémentées pour représenter l'intéraction entre deux groupes de boids dont l'un essaie d'attaquer l'autre. Les proies avancent en essaim assez nombreux, avec une vitesse supérieure à celle des prédateurs (ces derniers sont 2 fois moins réactifs). Et à chaque fois qu'un de ces derniers se rapproche d'une proie de l'essaim, ce dernier subit un anti-floc, tel que ses éléments essaient de se disperser au maximum. Ensuite, à chaque fois qu'un prédateur touche une proie, cette dernière meurt (disparait).

4.2 Les Tests Effectués:

Dans Cette partie, on a essayé de tester chacun des types de Boids qu'on a pu créer indépendemment, et de garder leurs paramètres (Nombre d'essaims et d'éléments par essaim) les plus manipulables possibles. Ainsi, on a:

- TestBirdsSimulator qui implémente des groupes d'essaims différents, de couleurs et de vitesses distinctes, qui se déplacent dans le même milieu indépendemment les uns des autres.
- TestHungryBoids qui représente plusieurs groupes d'essaims, chacun avec sa propre nourriture qu'il vise.
- TestFollowersSimulator qui représente plusieurs groupes d'essaims forcés à rester proches d'une trajectoire, et qui font donc des allers retours horizontaux sur ce Path
- TestPreyspredators qui représente un groupe de proies et un autre de prédateurs, intéragissant comme décrit précédemment.

4.3 Remarques Générales:

- Une fois le rebondissement des boids sur la paroi implémenté, on remarque que une fois que la vitesse des particules dépasse un certain seuil, ces dernière ne sont plus régies par les règles qu'on a implémentées, car leur vitesse domine désormais... elles se disperssent du coup, ne ressemblant plus à un essaim. Ainsi, on trouve ça nécessaire d'associer une vitesse limite à chaque essaim.
- Le dessin des triangles s'est fait par l'implémentation d'une sous classe de GraphicalElement, en utilisant la fonction drawPolygon. Ensuite leur orientation se fait grâce à l'utilisation d'un vecteur direction représentatif du send de leur vitesse (sur l'écran)
- En faisant plusieurs tests différents, on remarque qu'à chaque fois qu'on a une force d'attraction (HungryBirds, Followers....), plus le paramètre "maxForce" est grand, plus les éléments de l'essaim attiré convergent plus vite vers la cible. Ce qui est normal vu que la force d'attraction sera plus grande. Il ne faut pas dépasser un certain seuil apr contre.

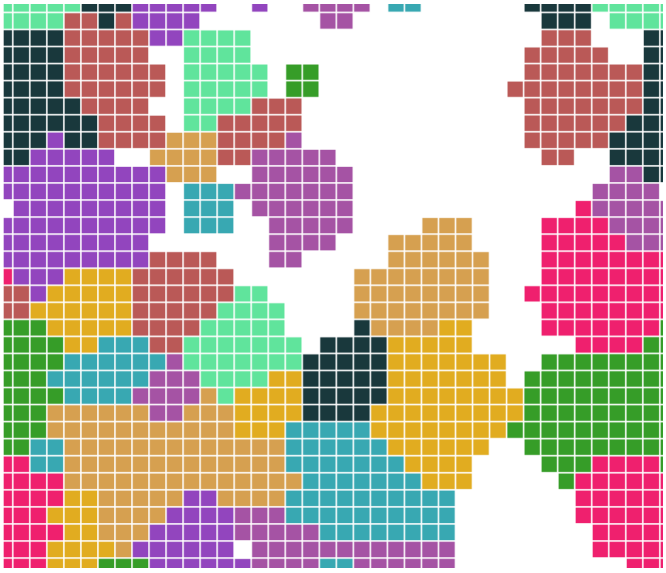

Figure 4: Schelling Testé pour 10 habitations et un seuil de 3:

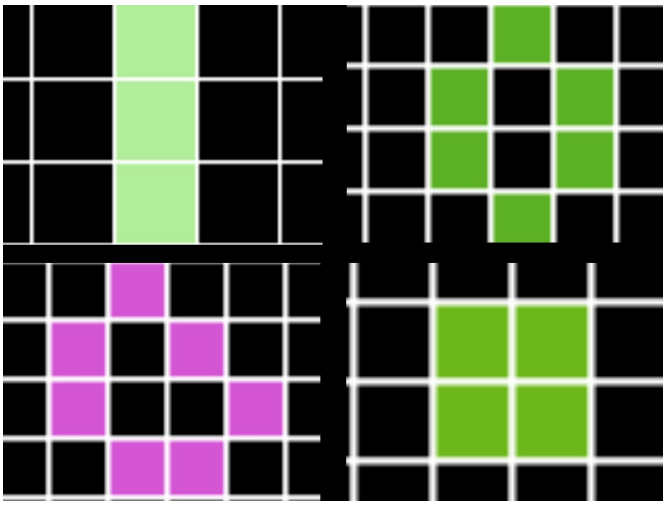

Figure 5: 4 Motifs Classiques résultant du jeu de Conway

5 Annexe