Projet Java: Simulateurs

Projet Programmation Orientée Objet

Hugo ELHAJ-LAHSEN, Maxime NEMO, Simon PITOIS

Principes de conception

Notre conception est un *entity-component system*.

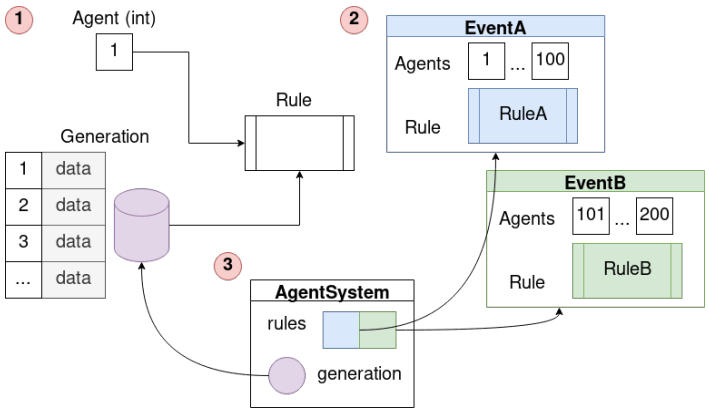


Figure 1: Conception du projet

On a choisi de représenter chaque agent par un identifiant. Une Generation stocke les données de chaque agent. Une Rule est du code s'appliquant a un agent et qui peut également impacter la Géneration.

Les Events vont créer le changement dans le système. Chaque Event possède une portée d'agents ainsi qu'une Rule qu'il applique. Quand l'event est executé par le système, il applique la règle à tous les Agent sur lesquels il agit.

L'AgentSystem invoque les Events le temps venu et affiche la génération actuelle.

Avantages et difficultées rencontrées

En pratique, nous avons utilisé la *dependency inversion*. Chaque système implémente des classes représentant les responsabilités nécéssaires (un Generation, une ou plusieures Rules, un Display qui affiche la génération) qui implémente des interfaces bien définies. On les passe à l'AgentSystem qui dépend de ces interfaces.

Cependant, cette abstraction rend parfois le code difficile. Cette conception n'était pas notre première idée: nous l'avons refait plusieurs fois, et nous avons fini sur une conception surement plus compliquée que nécéssaire.

En terme de performance, nous faisons une copie entière de la génération à chaque fois que nous faisons un next. En termes pratiques cette copie n'est pas nécéssaire tout le temps et fait des soucis de performance.

Squelette du projet

Un diagramme UML général au projet entier à été généré à partir des fichiers sources mais celui-ci ne rentrant pas dans ce PDF, il est disponible en annexe (Diagrammes_UML/project_java_diagram.png), nous regarderons dans ce rapport, des UML détaillés pour chaque parties.

La documentation Javadoc est disponible dans le dossier doc/.

Balls

Diagramme UML dans Diagrammes_UML/Balls_diagramme.png

Ici nous avons commencé par tout implémenter dans une seule et même classe. Puis, lorsque nous avons commencé à mettre en place les autres simulateurs et les autres agents à traiter, nous avons décidé de créer des classes communes mais qui font chacune leur travail : La ball en elle-même, sa création, les règles qui s'appliquent dessus (fonctions translate et apply qui permettent d'itérer les positions des balles) et enfin l'affichage des balles.

Les trois interfaces présentes dans agentsystem sont communes à toutes les simulations.

Nous avons essayer notre simulateur avec plusieurs balles placées à des coordonnées différentes (ici 18). Lors de la simulation, on peut voir que les balles rebondissent bien contre le bord, en prenant la direction adéquate. *Un gif est disponible dans Images_test/balls_test.gif*

Pour aller plus loin, on pourrait essayé d'implémenter les chocs entre les balles, néamoins cela nécessiterai d'avoir accès aux positions des autres balles pour toutes les balles. Ce simulateur se raprocherai alors plus de celui des boids que nous verrons après.

Jeu de la vie de Conway

Diagramme UML communs à toute la partie cellular dans Diagrammes_UML/Cellular_diagramme.png Diagramme UML de Conway dans Diagrammes_UML/Conway_diagramme.png

Pour cette simulation nous avons créé les classes et interfaces qui permettent de générer un quadrillage (sous la forme de carrés autour d'un point-centre) et de pouvoir régler la taille, les largeur et longueur… (GridGeneration/IGridGeneration) Nous avons aussi créé les interfaces (ICellTypes) permettant de créer les types de cellules (différentes selon les simulations) et les règles de simulation s'appliquant à chaque tour (IGridRule). De plus nous avons fait en sorte que la grille soit une sphère. Donc que les cases sur les bords soient comptées adjacentes aux cases de l'autre côté.

Nous avons fais des tests avec plusieurs positions de départ et un nombre de cellules en vie différent. *Un gif des différents cas testé est disponible dans Images_test/conway_test.gif*

On retrouve plusieurs cas ici :

-Un état stable (en bas à gauche), les cellulesse maintiennent en vie chacune mais aucune naissance n'apparait.

-Un état périodique (en haut à gauche), des cellules naissent et meurent mais on observe un cycle périodique. A un moment les cellules reviennent à leur emplacement de départ et recommencent.

-Un état de limite stable (en bas à droite), Ici les cellules évolue jusqu'à atteindre un état stable et se comportent comme le cas 2.

-Un état à limite nulle (en haut à gauche). Ici les cellules se dispersent et meurent petit à petit jusqu'a disparaitrent complètement.

Le jeu de l'immigration

Diagramme UML de l'immigration dans Diagrammes_UML/Immigration_diagramme.png

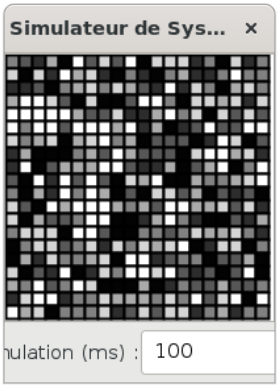
La conception du jeu de l'Immigration est très ressemblante à celle de Conway. Même si le sujet ne le spécifiait pas, nous avons choisit de rendre les bords adjacents (grilles sphérique). Nous avons gardé la même structure que le simulateur précédent avec la définition des cellules, celle de la grille et les règles qui s'appliquent dessus.

Concernant les tests, nous avons essayé plusieurs cas différents sur des tailles de grilles fixes (la taille influence peu l'évolution de la simultaion):

-Peu d'états différents (2 états) : pour deux états le programme "clignote". Les états s'inversent puis reviennent à leur état initial.

-Nombre moyen d'états différents (entre 3 et 6) : Les états se regroupent, plus lentement, puis au bout d'un moment une ou deux couleurs (nécessairement non succesives) devient prédominante et le programme s'arrête. La vitesse à laquelle les agents deviennent uniforme dépend beaucoup des conditions initiales et du nombre d'états.

-Nombre d'états différents important (plus de 6) : Il y a trop d'états pour avoir un changement de masse. On observe quelques modifications puis le programme s'arrête car le nombre important d'états empèchent les agents d'avoir 3 voisins de couleur superieure.



Ici nous avons la fin de la simulation pour 7 états. Les différentes couleurs sont loins d'êtres uniforme comme on l'obtient pour un nombre d'états inférieur.

Pour un nombre d'états inferieur, le gif est disponible dans Images_test/immigration_test.gif

On voit aussi que l'évolution des couleurs se fait teintes par teintes. Dans le gif, par exemple, on voit que le noir se transforme en blanc d'abord, puis à la fin, le blanc commence à se transformer en gris clair etc… jusqu'à obtenir une couche quasi uniforme de gris foncé.

###Schelling

Diagramme UML de Schelling dans Diagrammes_UML/Schelling_diagramme.png

Pour créer le simulateur de Schelling, nous avons commencé par utiliser la même classe GridGeneration. Mais nous avons eu des difficultés sur la gestion des voisins. Nous avons donc décidé de créer des tableaux différents pour Schelling, (tableaux des agents et des maisons vides). Mais la factorisation de notre code nous à permis de bien le faire, désormais Schelling dépend directement de l'interface IGridGeneration et profite de toutes les autres classe implémentées.

Notre programme semble fonctionnel. En effet, lors de la simulation, on observe bien l'effet de ségrégration montré par Schellling. Dans certains cas, lorsque le nombre de voisins différents nécessaires pour se déplacer est trop grand, on peut avoir des cas ou il y a peu de déménagements et donc pas de ségrégation.

Un gif d'une simulation à 6 couleurs de familles différentes

On voit bien la ségrégation qui s'opère, les maisons vacantes se situant pour la plupart à la limite entre les taches de couleurs.

Boids

Diagramme UML de Schelling dans Diagrammes_UML/Boids_diagramme.png

La conception des boids utilise la conception en règle multiples. On a choisi de créer plusieures règles différentes pour chaque groupe de boids, en spécifiant une portée de boids. Par exemple, une première règle s'applique pour les boids 0 à 200, une autre de 200 a 400. Mais on stocke tous les boids dans le même objet generation, ce qui est utile par exemple pour faire la détection de colision, ou il faut être au courant de tous les boids.

Nous avons fait des boids avec des vitesses et des visions différentes, qui se repoussent les uns les autres. Nous avons des prédateurs qui chassent les proies. Une force s'applique des que les boids sortent du champ, qui va les rediriger dans le champ.

Nous avons testé ce programme de beaucoup de manières différentes. Nous créé plusieurs types de boïds ayant chacun leur champ de vision, leur vitesse limite. Certains peuvent être des prédateurs (ici les verts) et des proies (les jaunes).

Voici une image de notre simulation après quelques secondes :

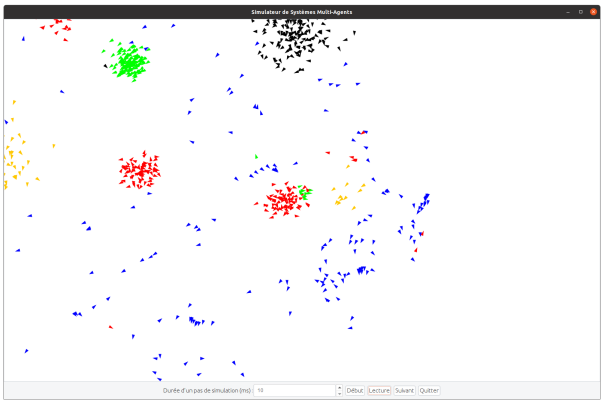


Figure 2: test_boids

-impact de la vitesse: Plus un Boïds peut être rapide, et plus son comportement est hasardeux. Si le champ de vision ou les forces qui s'appliquent sont relativement faibles, les boïds ont tendance à vivre leur vie et à ne pas faire très attention à leur semblable (Les boïds bleus). A l'inverse, les boïds lents ont plus tendance à se regrouper. (Ici les rouges)

-impact de la distance de vue Les boïds ayant le meilleur champ de vision ont tendance à se regrouper très vite et à former un seul groupe pendant toute la simulation. (En noir)

-impact proie prédateur Nous avons implémenté le comportement proie prédateur. Chhaque prédateur chassant une proie spécifique. Dans ce test, les boïds verts chassent les oranges, on le vois bien de par leur position, seulement les boïds orange sont bien plus rapides donc s'échappent facilement. La prochaine étape aurait été d'implémenter la destruction des boïds proies lorsqu'ils sont ratrapés par les chasseurs.