

1 Automates Cellulaires

1.1 Jeu de la vie de Conway :

Comme vous pouvez le constater, nous avons utilisé des classes assez 'génériques' pour qu'elles soient facilement réutilisables pour les autres jeux. De plus, on peut remarquer que notre conception est presque identique à celle proposée pour le jeu de balles. En effet, ici nous manipulons un tableau de *int* et non plus un tableau de *Point*. Ainsi pour l'adapter à d'autres jeux, il suffirait de modifier les correspondances entre les états et les couleurs et les règles d'actualisation.

Pour tester ces classes, nous avons utilisé deux classes de tests : l'une pour faire une simulation textuelle dans un premier tant, par la suite, quand la classe de calcul pour tester la simulation graphiquement.

1.2 Jeu de l'immigration :

Les seules différences avec le jeu de la vie sont

- Qu'une cellule peut prendre un nombre n_e d'états et non plus seulement 2.
- Les règles de changement d'états.

Du fait que notre conception est générique, nous avons eu qu'à créer une nouvelle classe fille de *Cellules* qui implémente une nouvelle méthode *actualiser()* propre au jeu de l'immigration.

Nous avons fait le choix de créer une correspondance entre les *Entiers* qui représente l'état d'une cellule et une *Couleur* qui sera affichée dans sa simulation graphique. Pour ce faire nous avons utilisé une table de correspondance *Map<Integer, Color>*.

1.3 Le modèle de Schelling :

Pour cette partie, en plus d'avoir implémenté la nouvelle règle d'actualisation, nous avons utilisé une collection Java pour lister les habitations vacantes : *LinkedList*. De plus, nous avons considéré l'état -1 comme étant l'état d'une habitation vacante pour faciliter les calculs et faisant bien attention que ce nouvel état ne soit pas considéré comme une couleur lors du calcul des voisins. Lorsqu'une famille déménage, elle emménage dans une habitation vacante prise au hasard dans la liste. Lors de l'initialisation, toutes les habitations vacantes prévues sont créées dans l'ordre dans les premières cellules. Ce nombre de cellules vacantes dépend du facteur de ségrégation k , du nombre de lignes et du nombre de colonnes, selon la formule : $\frac{\text{lignes} \times \text{colonnes}}{5 \times (k+1)}$, trouvée empiriquement.

2 Un modèle d'essaims : les boids

2.1 Troupeaux de boids

Pour implanter un simulateur de boids nous avons pensé que le plus efficace serait de réutiliser ce que nous avons fait sur les balles. Ainsi nous avons qu'à étendre les propriétés des balles. En effet, un boid est une balle à laquelle on ajoute une information sur sa vitesse. Nous avons donc défini un tableau de *Vect2D* qui est ni plus ni moins qu'une extension de la classe *Point* par souci de clarté entre un point qui repère la position d'un boid et un vecteur qui repère sa vitesse. Un boid est donc repéré par sa position et sa vitesse. Ainsi pour que l'ensemble des boids se comporte en essaim, nous avons décidé d'appliquer les 3 règles de bases : la cohésion, l'alignement et la séparation. Chacune de ces règles est affectée d'un gain kC , kA , dS qui permettent de donner plus ou moins d'importance au groupe et ainsi de donner des comportements différents aux boids.

2.1.1 Un gestionnaire à événements discrets :

Afin de gérer les événements dans *EventManager*, on utilise comme collection Java une *PriorityQueue* afin de classer les événements selon leur date d'exécution de façon croissante. La classe *CompareurDate* implémente cette relation de comparaison. La classe abstraite *Event* et ses sous-classes permettent de définir concrètement la nature des événements.

2.1.2 Modification de votre simulateur :

Grâce à la hiérarchie des classes utilisés, il est facile d'utiliser les événements à la place de l'ancienne gestion. Il suffit alors de créer un sous-événement *BBEvent* pour les balles et les boids et *CellulesEvent* pour tous les types de jeu reposant sur les cellules.

2.1.3 Simulation de plusieurs groupes de boids :

Nous avons généralisé ce problème en proposant aussi de simuler plusieurs groupes de balles, les boids étant des balles, cela ne posait pas de problème. Pour cela, il suffit simplement d'utiliser un tableau de balles partout où on les utilise, c'est à dire dans *BEvent* et dans *BBSimulator* afin d'avoir un simulateur qui contient ces groupes de balles/de boids. Les événements sont alors communs : à la création du groupe on choisit leur délai d'attente entre deux execution, et leur événements s'ajoutent à la queue globale des événements.

3 Bilan

Pour conclure ce projet, nous allons faire un bilan de notre travail :

- Ce projet nous a tout d'abord paru difficile à répartir entre les deux membres de notre équipe. Mais en s'organisant et en s'efforçant de diviser les tâches nous avons pu gagner beaucoup en efficacité.
- Du point de vue de l'organisation du code, nous avons utilisé Git qui nous a permis de gérer au mieux notre code.
- Le projet que l'on vous donne est selon nous abouti dans le sens où l'on a réussi à répondre à l'intégralité du sujet. Cependant avec un petit plus de temps nous aurions aimé ajouter quelques améliorations.
 - En effet, pour la simulation des boids, nous ne donnons pas d'angle de vision au boids, mais seulement une distance tout autour d'eux.
 - Le deuxième point à améliorer serait la possibilité de faire interagir les différents groupes de boids entre eux. Option que nous n'avons pas eu le temps de créer.
- Pour le reste lancer notre jeu et laissez vous guider.