Choix de conception

Ce projet contient différentes simulations graphiques comprenant :

1. Une modélisation de balles afin de prendre en main les fonctionnements de la bibliothèque GUISimulator.
2. Des simulations cellulaires prenant racine dans le principe du jeu de la vie de Conway
3. Une simulation de boids plus complexes utilisant les acquis des simulations cellulaires pour simuler des agents dans un environnement continu et non plus discret.

Jouons à la balle

La classe **Balls** :

L'attribut central est **points**, implémenté par une *ArrayList*. On a choisi une collection *List* car on a besoin de pouvoir ajouter des balles autant qu'on en veut sans nombre prédéfini. L'implémentation en tant qu'*ArrayList* est dû à la nécessité de la méthode *get* lors de la translation des balles. L'implémentation par *Array* étant la plus efficace lorsque cette méthode est nécessaire. La raison de l'implémentation pour les attributs **dx** et **dy** est la même.

La classe **BallsSimulator** :

Cette classe hérite de **Balls** et implémente l'interface **Simulable**. Elle implémente donc les méthodes *next* et *restart* et. Ses attributs sont **gui** et **ballsColor** qui sont nécessaires pour la simulation.

Des automates cellulaires

- Les classes fondamentales:

La classe **Cell**:

Différents constructeurs ont été mis en place afin de faciliter la création des différentes grilles pour le jeu de la vie et ses variantes.

Un maximum de calculs de génération concernant des modifications d'états de cellule ont été déplacés dans cette classe afin de garantir la sécurité de l'attributs **state**.

Cette classe héberge les règles intrinsèques aux cellules pour les différentes simulations. On retrouve les règles de vie et de mort du jeu de la vie, d'augmentation d'état dans la variante de l'immigration ou encore les règles de déménagements dans le modèle de Schelling.

La classe **Board**:

Cette classe est plus complexe et héberge toutes les méthodes nécessaires aux calculs des générations.

L'attributs central est **cellBoard**, implémenté par une matrice de cellules sous forme de *ArrayLists*. Ce choix a été motivé par la nécessité de parcourir des indices de listes, la structure de données est donc apparue comme évidente. Or, de plus, on peut repérer un avantage supplémentaire car la taille de la matrice étant fixe, il n'y a pas besoin de redimensionner les tableaux, l'opération coûteuse des tableaux redimensionnable est donc esquivée.

On repère de la même manière que pour la classe Cell, différents constructeurs pour les différentes simulations, ainsi que différents modifieurs pour ces mêmes applications.

On retrouve également des méthodes relatives aux calculs des voisins, nécessaires aux calculs de générations. Ces méthodes nécessitant des manipulations de coordonnées en 2D, nous avons choisi d'utiliser la classe *Point2D.Double* qui permet de gérer avec une seule méthode les problèmes d'effets de bords (cf. méthode **InBounds**).

Pour finir, afin de gérer les calculs de générations, une matrice de cellule miroir stockant les données sur les voisins et utilisant également les *ArrayLists* a été défini pour les mêmes raisons que pour l'attribut **CellBoard**.

- Les classes spécifiques:

La classe **ConwayBoard**:

Le jeu de la vie est implémenté par une grille utilisant la même structure de donnée que **Board**, puisqu'elle hérite de cette classe (comme toutes les autres classes hérite de cette dernière nous n'allons donc plus le préciser par la suite).

Or elle s'appuie également sur un *HashSet<Points2D.Double>* notamment au moment de sa création afin de prédéfinir les cellules vivantes. Ici un *HashSet* a été choisi afin de garantir l'unicité des *Points2D.Double* mais également afin de pouvoir accéder en temps constant aux cellules vivantes par la méthode **contains** de *HashSet*. Ici le *HashSet* est également suffisant car le jeu de la vie ne fonctionne qu'avec des cellules vivantes ou mortes, donc si elles sont dans le *HashSet* alors elles sont vivantes sinon non.

Cette classe, et les futurs classes également, implémente l'interface **Simulable**, on retrouve donc les méthodes **draw**, **next** et **restart**.

La classe **ImmigrationBoard**:

La spécificité de cette classe provient de sa création par une *HashMap<Point2D.Double, Integer>*, cette fois ci une *HashMap* est nécessaire afin d'associer à une cellule (ou plutôt ici ses coordonnées), un certain état.

La classe **SegBoard**:

Cette classe est très similaire à la classe **ImmigrationBoard**, les structure de données impliquées sont identiques, et le seul attribut ajouté est le seuil de ségrégation **segSeuil**.

Note : Nous avons essayé de faire hériter **SegBoard** de **ImmigrationBoard** mais n'avons pas réussi à le faire fonctionner avec l'event manager. Cela aurait été intéressant dû à la très grande similitude entre les deux classes.

Une simulation de boids en environnement continu

La dernière partie du projet consiste en la modélisation d'une population d'agents évoluant dans un espace continu, en appliquant des règles inspirées du modèle de Reynolds. Contrairement aux automates cellulaires, les boids ne dépendent plus d'une grille discrète mais manipulent directement des coordonnées réelles. La conception a donc nécessité un découplage plus propre entre calculs physiques, comportement collectif et affichage.

- Les classes fondamentales

La classe Element
Cette classe définit l'ensemble minimal des attributs nécessaires au mouvement d'un agent : position (héritée de `Point2D.Double`), vitesse et direction. Elle sert de socle générique : aucune logique comportementale n'est intégrée ici, ce qui permet de spécialiser ensuite plusieurs familles d'agents (boids classiques, prédateurs…) sans polluer la base.

La classe Boids
Elle implémente les trois règles canoniques du flocking :

- *Alignement* : tendance à s'orienter selon la moyenne pondérée des vitesses des voisins ;
- *Séparation* : force de répulsion forte à courte distance pour éviter les collisions ;
- *Cohésion* : force faible attirant l'agent vers le centre local du groupe.

Afin d’obtenir des mouvements fluides et réalistes, l’ensemble de ces forces est calculé sous forme vectorielle. Les pondérations décroissantes selon la distance réduisent les ruptures de direction et évitent la formation de sous-groupes artificiels.

Un choix de conception important est l'utilisation d'un **double tampon** (`nextDirection`, `nextVelocity`). Tous les boids calculent leur prochain état à partir de l'instant courant, puis appliquent les changements simultanément lors d'un second passage. Cette méthode empêche qu'un boid mette à jour son mouvement en se basant sur des voisins déjà mis à jour, phénomène qui crée habituellement du "jitter" ou des comportements incohérents.

Enfin, un module d'**évitage de murs** introduit une force douce lorsque l'agent s'approche des bords, ce qui remplace un rebond brutal par un virage progressif beaucoup plus naturel.

La classe PredatorBoid

Elle hérite de `Boids` tout en surchargeant la règle d'orientation. Le prédateur identifie sa cible la plus proche, puis corrige progressivement sa direction via une interpolation angulaire (`blendAngles`). Cette technique évite les retournements instantanés, et la légère accélération intégrée crée un comportement de poursuite crédible. Ce choix confirme l'intérêt de structurer le projet autour d'une classe de base simple (`Boids`) permettant des extensions spécialisées.

- Les structures globales

La classe Swarm

Elle encapsule une liste d'agents et gère la séquence de mise à jour : calculs d'abord, application ensuite. Ce choix garantit la cohérence interne du système et permet d'instancier plusieurs groupes indépendants (par exemple sardines vs prédateurs). L'initialisation repose sur un choix simple : placement aléatoire et vitesses directionnelles variées, ce qui donne immédiatement une dynamique non triviale.

La classe PredatorSwarm

Elle surcharge uniquement la méthode d'initialisation pour instancier des `PredatorBoid`. L'architecture par délégation permet d'éviter toute duplication et maintient la cohérence entre groupes.

Orchestration par événements

Comme pour les automates cellulaires, la simulation temporelle complète est pilotée par un **EventManager**. L'intérêt est particulièrement fort dans le cas des boids : différentes populations peuvent évoluer à des fréquences distinctes. Dans notre implémentation, les boids sont mis à jour à chaque tick, tandis que les prédateurs se déplacent plus lentement. La PriorityQueue garantit l'exécution dans le bon ordre temporel et simplifie l'introduction de nouveaux groupes ou comportements.