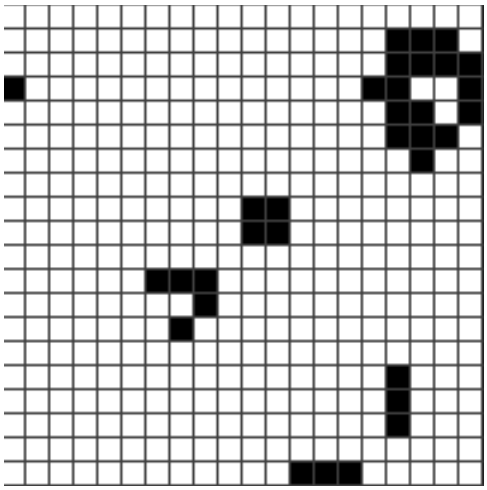
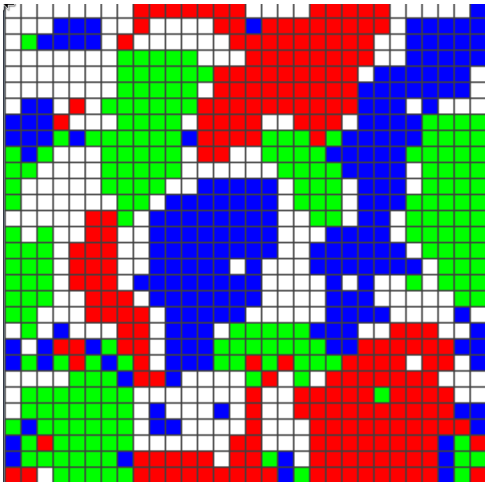
Programmation Orientée Objet Rapport du TPL

Equipe 26

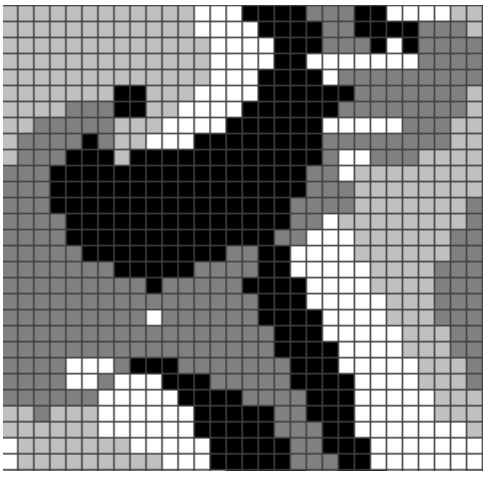


(b) Game of Life

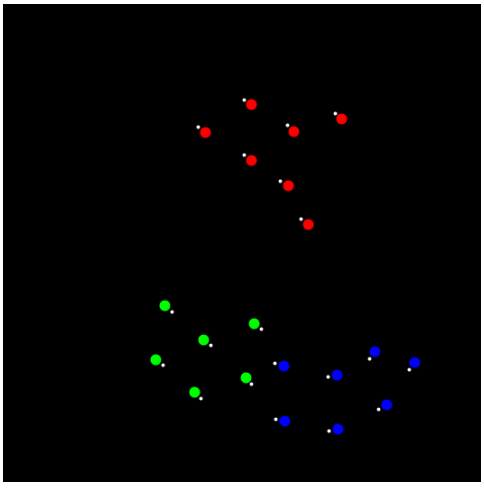


(c) Segregation de schelling

(c) Game of Life and Schelling



(e)



(f) Boids

(f) Schelling and Boids

Figure 1: Les résultats de la simulation

1 Avant-propos

1.1 Résultats et Rendu

En théorie, nous sommes arrivés au bout du projet java. Les performances sont très bonnes (voir excellentes) et les applications robustes. On vous fournit, chers lecteurs, avec ce rapport plusieurs fichiers bonus :

- Le **README** de notre projet vous disant comment simuler et lancer les tests
- Un annexe à la fin du rapport représentant un **digramme UML** de notre projet
- Une documentation complète de toutes (eh oui toutes) les classes Java. Vous la trouverez dans le répertoire **/documentation**
- Le fichier **xml** du coding style

RQ: Le coding style respecté est celui du **sun_check.xml**. Certaines règles de ce coding style sont :

1. Les variables doivent être en CamelCase :
   (a) variable nommée h ou x autorisée (h pour height et x pour l'abscisse)
   (b) variable nommée *initAlive* autorisée
   (c) variable nommée *init_alive* interdite
2. Une variable qui peut être **final** doit être **final**

Vous pouvez vérifier avec l'outil check_style.

1.2 Méthode de travail

Nous avons profité du fait d'être une équipe de trois étudiants pour mettre un point d'honneur à faire tester notre code par une personne tierce.

Durant toute la durée du projet, chaque partie de code était push sur une branche git « test » pour être testé et commentée par l'un ou l'autre (ou les deux) des membres de l'équipe. Après correction des éventuelles erreurs ou amélioration de la méthode, le commit pouvait être push sur la branche master.

Cette méthode nous a garanti un code robuste et nous a permis d'avoir un retour constructif sur chaque partie du projet à l'aide d'une vision la plus globale possible. De plus, certaines parties ont été entièrement développées en « pair programming », toujours dans la même optique de soutien au sein de l'équipe.

2 Choix de conception

2.1 Packages

Étant donnée la diversité des tâches demandées au cours de ce TP, nous avons décidé de découper les différentes parties en packages dans notre projet.

Ainsi, nous avons **5** packages relatifs aux différentes parties du TP, à savoir «balls», «conway», «immigration», «schelling» et «boids». Nous ajoutons à ceux-ci deux autres packages, contenant respectivement les classes/structures communes à toutes les parties («glob») et les tests effectués dans le cadre du TP («tests»).

2.2 Simulateurs

La classe asbtraite Simulateur au centre du diagramme **UML** implémente l'interface Simulable et contient un GUISimulator qui lui permet entre autres d'accéder à la fenêtre d'affichage.

De cette classe héritent cinq simulateurs propres à chaque partie. Ils sont globalement responsables de l'affichage de la fenêtre et des boutons d'interaction. Ce sont également les seules classes faisant appel à des fonctions d'affichage d'éléments («addGraphicalElement»).

Ainsi, les calculs concernant les interactions entre les différents acteurs du programme sont **complètement séparés** de leur affichage. Cela assure une gestion cohérente de l'application ainsi qu'une meilleure lisibilité du code.

2.3 Évènements

L'application finale gère un système d'évènements ponctuels. Ceux-ci sont pris en charge par un « EventManager » appartenant au Simulateur abstrait de base.

Ce manager a pour rôle de créer de nouveaux évènements et les faire interagir avec le programme. Pour ce faire, il dispose d'un attribut correspondant à une liste d'évènements. Ces évènements peuvent être de trois types différents : «EventBoids», «EventBalls» et «EventGrid». Le manager va donc créer un évènement, le rajouter à sa liste et, lorsque son temps sera venu, le communiquer au Simulateur dont il fait partie, qui le traitera en fonction.

3 Quelques Notions de POO utilisées

L'abstraction : Comme vous pouvez le voir sur le diagramme UML, on a utilisé des classes abstraites comme par exemple Simulateur. Cela a permis de bien factoriser le code.

Le polymorphisme : On a bien utilisé dans notre code des polymorphismes d'objets. Par exemple on utilise l'**ImmigrationGrid** en la manipulant avec une référence de type **ConwayGrid** (voir UML).

L'héritage : Celui-ci est présent partout dans notre code pour le rendre plus structuré et plus compréhensible tout en respectant toujours le principe d'encapsulation.

Les collections : On utilise plein de collections Java. Par exemple, les Arraylist, Hashmap, PrioritQueue etc.

Les Interfaces : Les interfaces sont utilisées dans notre projet comme par exemple l'interface Comparable dans la gestion des événements.

4 Tests sur nos applications

Dans le fichier « TestSimulator », on peut voir une batterie de simulateurs prêts à être lancés.

En modifiant les paramètres de ces simulateurs, on peut aisément tester différentes situations pour chaque partie du TP. L'autre fichier de test permet d'afficher textuellement notamment les position des balls ou encore l'état des events pour s'assurer de leur bon fonctionnement.

5 ANNEXE

voir page suivante

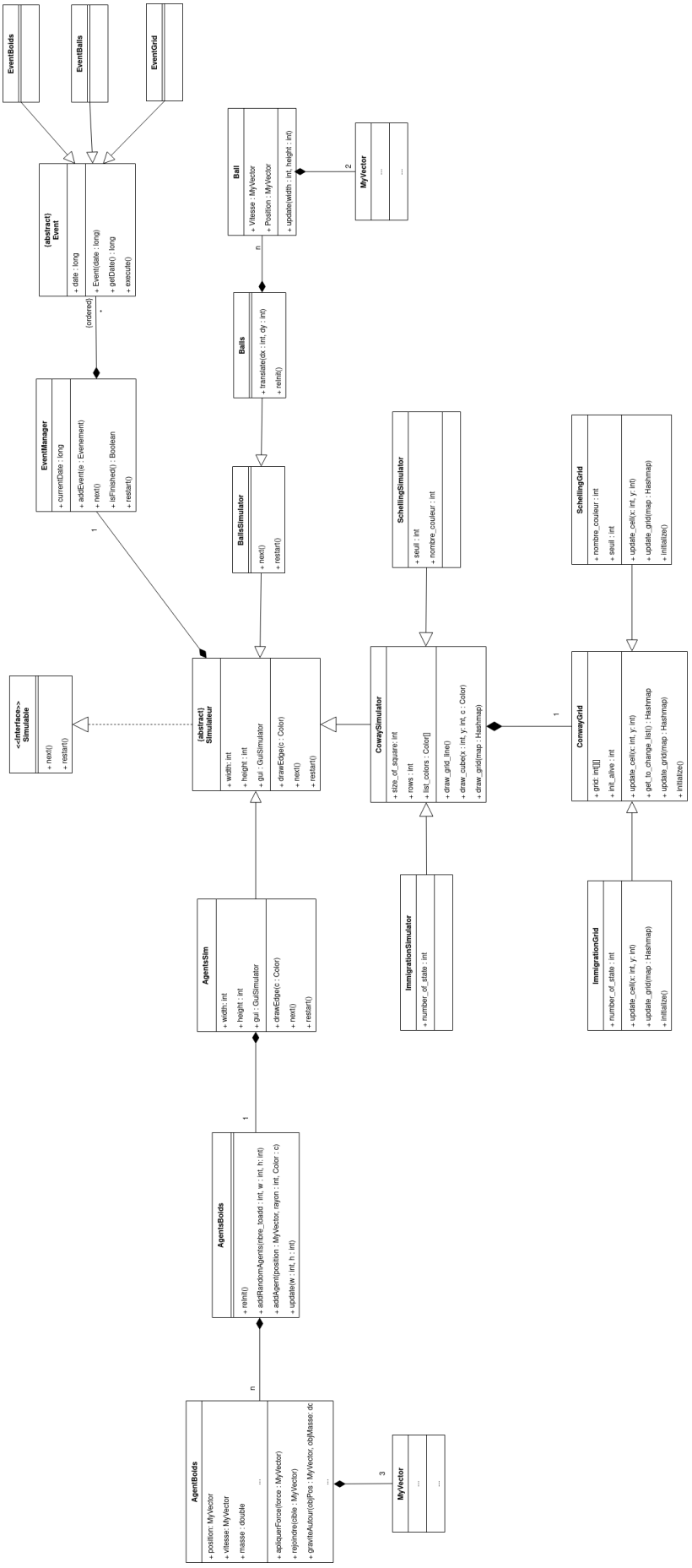


Figure 2: Diagramme UML du projet