Simulation Orientée-Objet de systèmes multiagents

TP en temps Libre de Programmation Orientée Objet

Ensimag 2A MMXVII

Résumé

L'objectif de ce TP est de développer en `Java` une application permettant de simuler de manière graphique des *systèmes multiagents*. Dans un premier temps, nous nous intéresserons à trois systèmes de type automate cellulaire : le *jeu de la vie de Conway*, un *jeu de l'immigration*, et le *modèle de ségrégation de Schelling*. Dans un second temps, nous nous intéresserons à la simulation d'un système de mouvement d'essaims auto-organisés : le *modèle de Boids*.

Ce projet vous permettra d'aborder les aspects fondamentaux de la programmation orientée objet : encapsulation, délégation, héritage, abstraction et utilisation des collections `Java`. Un effort particulier sera porté sur la conception orientée objet des classes, pour généraliser au maximum le code commun entre les différentes systèmes multiagents étudiés.

1 Introduction

Nous allons nous intéresser dans ce TP à une branche de l'intelligence artificielle dédiée à l'étude des propriétés des systèmes collectifs : les *systèmes multiagents*. De manière basique, un système multiagent est un système constitué d'un ensemble d'*agents autonomes* capables d'interagir entre eux et avec leur environnement. Un agent peut être par exemple un robot, un processus informatique, ou encore une entité légale ou un être humain [1]. Les chercheurs de ce domaine s'intéressent principalement au développement de modèles de raisonnement et d'interaction réalistes (comme ceux que l'on voit dans les simulations de foules utilisées par exemple au cinéma), et à l'analyse de leurs propriétés.

Même si une partie des travaux sur les systèmes multiagents sont purement formels, tout un pan de la communauté fonde son travail sur la *simulation* de modèles de systèmes multiagents. Précisément, il s'agit de créer une communauté d'agents artificiels régis par des règles de comportement (en général très simples), et d'observer la dynamique de cette communauté par le biais de simulations informatiques. C'est ce que nous allons faire dans ce travail.

Le sujet semble long à première vue, mais certaines parties sont très guidées et n'exigent pas énormément de code à écrire. Les parties suivantes demandent un peu plus de réflexion, mais vous avez toutes les bases pour les mener à bien. Le travail est prévu pour des groupes de trois étudiants. Vous avez la possibilité de former des binômes, mais attention à bien gérer la charge de travail le cas échéant.

Bien entendu, vos enseignants sont là pour vous aider et répondre à vos questions (mais cela exige que vous vous y preniez assez tôt...).

La date limite de rendu est fixée au **lundi 13 novembre 2017, 19h**.

Bon travail à tous !

2 Un premier simulateur : jouons à la balle

Une interface graphique de simulateur, sommaire, vous est fournie. Elle permet :

1. de contrôler une simulation, via différents boutons – `Début`, `Lecture`, `Suivant`, `Quitter`.
2. de créer et d'afficher une fenêtre graphique d'une taille donnée, et d'une couleur de fond donnée, sur laquelle vous pourrez dessiner différentes formes géométriques (rectangles, ovales, texte, image...) ;

1. Dans l'absolu, la frontière entre un agent et un simple programme informatique capable de communiquer est mince. Même si en général, un agent a des capacités de raisonnement et d'interaction évoluées, ce n'est pas toujours le cas, et la question de savoir si l'on est face à un système distribué classique ou à un système multiagents est une question de point de vue.

Cette interface graphique est disponible sous forme d'une archive `gui.jar` contenant le *bytecode* Java des classes disponibles (mais vous n'avez pas accès au code source). La documentation (API, *Application Programming Interface*) de ces classes, principalement `GUISimulator` et `Simulable`, sera disponible sur la page `Chamilo` du cours. Un fichier `TestGUI.java` est aussi fourni, avec un exemple de dessin.

Pour commencer, on s'intéresse à simuler et afficher des « balles » qui se déplacent à l'écran, en rebondissant sur les bords. Cette partie est simple et très guidée, l'objectif est uniquement de prendre en main la programmation d'un simulateur avec les éléments distribués.

2.1 Les balles

Une balle est représentée par ses coordonnées (x, y) dans le plan... et rien d'autre. Il faut pouvoir fixer sa position, la translater et afficher ses coordonnées. Plutôt que de créer une classe spécifique (par exemple `Ball`), il est ici possible d'utiliser une classe Java existante qui fournit exactement ce qui est attendu : la classe `Point` du package `java.awt`. Documentez-vous sur cette classe, en regardant son API disponible ici : `http://docs.oracle.com/javase/7/docs/api/index.html?java/awt/Point.html`. De manière générale, toutes les classes des librairies Java existantes sont (très) bien documentées, et vous devez avoir le réflexe de regarder leur API !

Question 1 Créez une classe `Balls` composée de quelques balles placées sur le plan à des coordonnées *ad hoc*. Ajoutez les méthodes suivantes (plus si nécessaire) :

- `void translate(int dx, int dy)` : translate toutes les balles ;
- `void reInit()` : remet toutes les balles à leur position initiale ;
- redéfinissez la méthode `public String toString()` pour qu'elle retourne une chaine de caractères avec les positions de toutes les balles.

Créez ensuite une classe `TestBalls`, munie d'une méthode `main`, qui crée une instance de `Balls` et utilise les méthodes ci-dessus pour modifier/afficher les coordonnées des balles.

2.2 Animons un peu tout cela

Vous allez maintenant animer les balles à l'aide de l'interface de simulation fournie.

À sa création, la classe `GUISimulator` est associée à un objet de type `Simulable`, qui déclare deux méthodes `next()` et `restart()`. Ces méthodes sont automatiquement exécutées en réponse aux actions de l'utilisateur sur les boutons :

- `void next()` est invoquée suite à un clic sur le bouton `Suivant`, ou bien à intervalles réguliers si la lecture a été démarrée (le pas de temps entre deux évènements `next()` est paramétrable).
- `void restart()` est invoquée suite à un clic sur le bouton `Début`. La lecture est alors arrêté, et le simulateur doit revenir dans l'état initial.

Pour utiliser cette interface, vous devez créer une classe (par exemple `BallsSimulator`) qui *réalise* l'interface `Simulable`, c'est-à-dire qui définit concrètement les méthodes de l'interface pour traiter les évènements de manière adéquate en fonction des données et de l'état de la simulation. Il est alors possible d'associer une instance de `BallsSimulator` (qui par héritage EST de type `Simulable`) à un objet `GUISimulator`.

Les relations entres les classes sont représentées sur le *diagramme de classes* de la figure 1. Une courte notice sur la notation UML (*Unified Modeling Langage*) utilisée est disponible sur la page `Chamilo` du cours.

☞ *Notez bien la séparation de la partie « calcul » de la simulation (ici dans la classe `Balls`) et de la partie « graphique », ici gérée via la classe `BallsSimulator`). Ceci est généralement conseillé, notamment pour la clarté du code et la réutilisation.*

Question 2 Ecrivez une classe `BallsSimulator` qui réalise l'interface `Simulable`. Ici cette classe peut simplement posséder un attribut de type `Balls`. Les méthodes `next()` et `restart()` ne font que déléguer le traitement à cet attribut pour mettre à jour puis réafficher l'état des balles.

La figure 3 fournit un programme de test qui ouvre une fenêtre de simulation, sans rien dessiner pour l'instant. Par contre vous pouvez utiliser les boutons de l'interface pour modifier l'état des balles. Un exemple de trace est fourni, avec trois balles et une translation de $(10, 10)$ à chaque appel à `next()`.

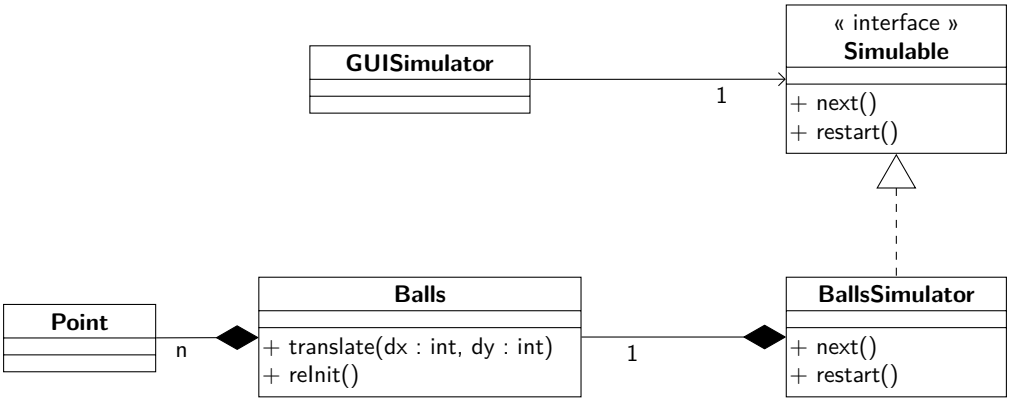

FIGURE 1 – Diagramme des classes permettant de contrôler le simulateur `Balls` à l'aide de l'interface graphique fournie. La classe `BallsSimulator` doit *réaliser* l'interface `Simulable`, et donc (re)définir les deux méthodes `next()` et `restart()`.

```java
public class BallsSimulator implements Simulable {
    // ...

    @Override
    public void next() {
        // ...
    }

    @Override
    public void restart() {
        // ...
    }
}
```

FIGURE 2 – La *réalisation* de l'interface `Simulable` est faite grâce au mot-clé `implements`. La classe s'engage donc à définir toutes les méthodes déclarées dans l'interface, ici `next()` et `restart()`. Par héritage une instance de `BallsSimulator` EST un objet `Simulable`, et peut donc être utilisée comme telle par le `GUISimulator`.

```java
import gui.GUISimulator;
import java.awt.Color;

public class TestBallsSimulator {
    public static void main(String[] args) {
        GUISimulator gui = new GUISimulator(500, 500, Color.BLACK);
        gui.setSimulable(new BallsSimulator());
    }
}
```

```
[java.awt.Point[x=10,y=110], java.awt.Point[x=110,y=110], java.awt.Point[x=210,y=110]]
[java.awt.Point[x=20,y=120], java.awt.Point[x=120,y=120], java.awt.Point[x=220,y=120]]
[java.awt.Point[x=30,y=130], java.awt.Point[x=130,y=130], java.awt.Point[x=230,y=130]]
[java.awt.Point[x=40,y=140], java.awt.Point[x=140,y=140], java.awt.Point[x=240,y=140]]
[java.awt.Point[x=0,y=100], java.awt.Point[x=100,y=100], java.awt.Point[x=200,y=100]]
[java.awt.Point[x=10,y=110], java.awt.Point[x=110,y=110], java.awt.Point[x=210,y=110]]
```

FIGURE 3 – Programme de test et trace d'exécution de la simulation des balles, après 4 clics sur `Suivant`, puis `Début`, puis `Lecture`.

2.3 C'est plus joli en affichant

La dernière étape consiste à afficher les balles dans la fenêtre graphique.

Le fonctionnement est simple, il suffit d'utiliser la méthode `addGraphicalElement()` de la classe `GUISimulator`. Cette méthode prend en paramètre une instance d'une sous-classe de `GraphicalElement`, chaque sous-classe représentant une forme particulière (ovale, rectangle, text, ...). Il est aussi possible de rajouter un type d'affichage en écrivant une nouvelle classe fille de `GraphicalElement`. Pour effacer toutes les formes de la fenêtre, on utilise la méthode `reset()`.

Le code de la figure 4 est un exemple d'utilisation basique de l'interface graphique, pour afficher quatre carrés bleus sur fond noir.

```
1  import java.awt.Color;
2  import gui.GUISimulator;
3  import gui.Rectangle;
4
5  public class TestGUI1 {
6      public static void main(String[] args) {
7          GUISimulator window = new GUISimulator(500, 500, Color.BLACK);
8
9          for (int i = 100; i < 500; i += 100) {
10             window.addGraphicalElement(
11                 new Rectangle(i, 250,
12                         Color.decode("#1f77b4"), Color.decode("#1f77b4"), 10));
13         }
14     }
15 }
```

FIGURE 4 – Un exemple de code illustrant le fonctionnement de l'interface graphique, qui dessine 4 carrés bleus sur fond noir.

Question 3 Reprenez votre simulateur pour afficher les balles dans la fenêtre graphique, par exemple comme des cercles. A chaque appel de `next()`, les cercles doivent être réaffichés selon leur nouvelle position.

☞ *Dans quelle classe sont invoquées les méthodes de dessin ? Réfléchissez en particulier au lien(s) nécessaire(s) entre les classes `BallsSimulator` et `GUISimulator`.*

Question 4 Si vous êtes joueur, animez les balles pour qu'elles rebondissent sur les bords de la fenêtre pour toujours rester visibles.

Et voilà, vous avez terminé votre premier simulateur !

3 Des automates cellulaires

Tous les éléments sont désormais en place pour construire des simulations graphiques de systèmes multiagents. Il ne manque plus qu'à coder les modèles des systèmes, puis les lier à un simulateur (et à l'interface graphique qui va avec).

Les premiers systèmes étudiés sont des types de systèmes multiagents particuliers : les *automates cellulaires*. Pour citer Wikipédia :

> Un automate cellulaire consiste en une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini et qui peut évoluer au cours du temps. L'état d'une cellule au temps $t+1$ est fonction de l'état au temps t d'un nombre fini de cellules appelé son « voisinage ». À chaque nouvelle unité de temps, les mêmes règles sont appliquées simultanément à toutes les cellules de la grille, produisant une nouvelle « génération » de cellules dépendant entièrement de la génération précédente.

Les trois systèmes multiagents auxquels nous allons nous intéresser dans cette partie sont des exemples d'automates cellulaires.

3.1 Le jeu de la vie de Conway

Vous avez probablement entendu parler du jeu de la vie de Conway. Il s'agit d'un automate cellulaire dont les règles sont extrêmement simples.

— Chaque cellule de la grille peut prendre deux états : *vivant* ou *mort*.
— Une cellule morte possédant exactement trois voisines (sur huit) vivantes devient vivante (elle naît).
— Une cellule vivante possédant deux ou trois voisines (sur huit) vivantes le reste, sinon elle meurt.

Un exemple est donné figure 5.

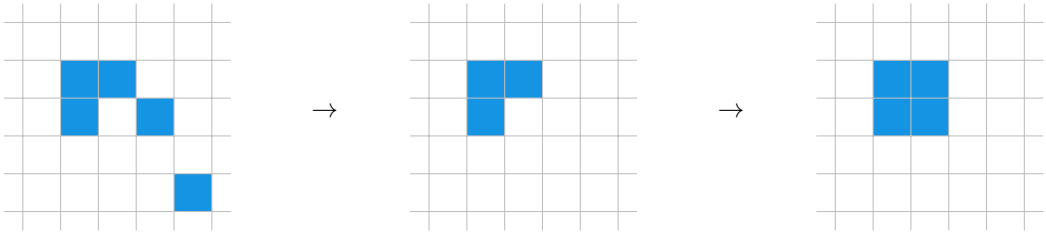

FIGURE 5 – Un exemple d'évolution sur trois étapes d'un extrait de grille dans le jeu de la vie. Les cellules mortes sont en blanc, les cellules vivantes en bleu.

En ce qui concerne les cellules situées sur les bords de la grille, on suppose pour simplifier que la grille est circulaire. Ainsi, par exemple, si la grille est de taille $n \times m$, la cellule (k, m) aura pour voisins les cellules $(k-1, m-1)$, $(k, m-1)$, $(k+1, m-1)$, $(k-1, m)$, $(k+1, m-1)$, $(k-1, 1)$, $(k, 1)$ et $(k+1, 1)$ (si $k < n$).

Question 5 Développez un simulateur graphique du jeu de la vie de Conway, en vous appuyant sur les éléments déjà développés aux parties précédentes.

☞ *Attention, l'état des cellules à l'étape $t+1$ dépend de l'état des cellules à l'étape t. Faites attention si vous modifiez « à la volée » l'état des cellules, car lorsque vous calculez l'état suivant d'une cellule donnée, il est possible que ses voisines aient déjà été modifiées, ce qui conduit à un résultat incorrect.*

3.2 Le jeu de l'immigration

Le deuxième automate cellulaire considéré est une légère variante du jeu de la vie, appelée « jeu de l'immigration » (pour une raison que j'ignore). Le principe est simple :

— chaque cellule peut être dans n états (au lieu de deux pour le jeu de la vie) ;
— une cellule dans un état k passe à l'état $k+1 \pmod n$ si et seulement si elle a trois voisines ou plus dans l'état $k+1$.

La figure 6 montre un exemple d'évolution d'un extrait de grille du jeu de l'immigration, utilisant le cycle à quatre états apparaissant sur la figure 7.

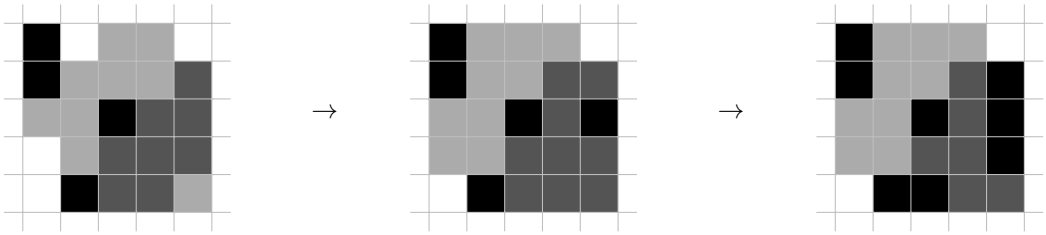

FIGURE 6 – Un exemple d'évolution sur trois étapes d'un extrait de grille dans le jeu de l'immigration (les cases en dehors de l'extrait de la grille ne sont pas montrées, mais vous pouvez en deviner certaines au vu de l'évolution des cases du bord de l'extrait). Les couleurs associées aux états apparaissent dans la figure 7.

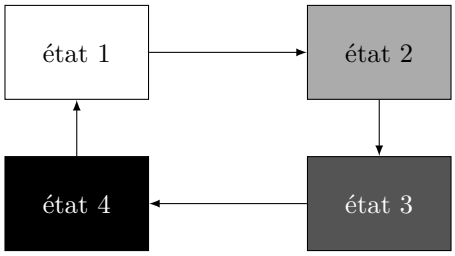

FIGURE 7 – Le cycle d'états utilisé dans l'exemple de la figure 6.

Question 6 Développez un simulateur graphique du jeu de l'immigration, en vous appuyant sur les éléments déjà développés aux parties précédentes.

☞ *Vous allez sans doute remarquer qu'une grande partie du code est commun entre ce jeu de l'immigration et le jeu de la vie. Avez-vous un moyen d'exploiter correctement ces éléments communs et de ne pas récrire tout le code ?*

3.3 Le modèle de Schelling

Nous allons maintenant nous pencher sur un modèle d'automate cellulaire légèrement différent, mais se voulant plus réaliste : le modèle de ségrégation de Schelling.

En 1971, l'économiste Thomas Schelling, qui étudiait la dynamique ségrégationniste, montra, grâce à un modèle extrêmement simple, que même une tendance très légère de chaque individu d'une population à préférer des voisins similaires pouvait conduire à une ségrégation totale (Schelling, 1971). Pour le démontrer, il s'est appuyé sur le modèle d'automate cellulaire suivant.

— Chaque cellule de la grille représente une habitation.
— Une habitation peut être soit vacante, soit habitée par une famille de couleur c (on suppose qu'il y a un nombre fini de couleurs).
— Si une famille de couleur c a plus de K voisins (sur huit) de couleur différente (K est un seuil qui est un paramètre de la simulation), alors la famille déménage, c'est-à-dire qu'elle va chercher une habitation vacante et s'y installer. L'habitation qu'elle occupait jusqu'ici devient alors vacante.

Question 7 Développez un simulateur graphique du modèle de Schelling. Faites varier le nombre de couleurs et le seuil K, et observez le résultat au bout de quelques itérations. À partir de quel seuil obtenez-vous une ségrégation ?

☞ *N'oubliez pas, lors de l'initialisation de vos cellules, de prévoir suffisamment de logements vacants. Assurez-vous en outre que deux familles ne puissent pas emménager dans le même logement à la même étape.*

☞ *Pour stocker l'ensemble des logements vacants, il est fortement conseillé d'utiliser une* collection Java. *Ces classes fournissent des implémentations efficaces de nombreuses structures de données :* `List`, `HashSet`, `PriorityQueue`, etc. *Un document de présentation des principales collections Java et de leur utilisation est disponible sur la page* `Chamilo` *du cours.*

☞ *Encore une fois, une partie du code précédent est probablement réutilisable. À vous d'exploiter correctement cette caractéristique.*

4 Un modèle d'essaims : les *boids*

Un autre système va maintenant être étudié pour simuler le déplacement d'agents en essaims. Ce modèle a été proposé par Craig Reynolds pour l'animation graphique de groupes d'animaux auto-organisés, tels des essaims d'oiseaux ou de poissons (Reynolds (1987)). Ces agents sont appelés des *boids*[2].

Contrairement aux automates cellulaires, les *boids* ne sont pas répartis sur une grille régulière mais se déplacent librement dans l'espace, 2D dans notre cas. Le comportement d'un agent dépend de la position

2. abrégé de *bird-oid objects*, en référence aux agents se comportant comme des oiseaux.

et de l'orientation des agents l'environnant, c'est-à-dire qui sont suffisamment près de lui et dans son champ de vision. Le modèle de base repose sur l'application de trois règles :

Cohésion un agent se dirige vers la position moyenne (centre de masse) des ses voisins ;

Alignement un agent tend à se déplacer dans la même direction que ses voisins ;

Séparation les agents trop proches se repoussent, pour éviter les collisions.

Ces trois règles sont suffisantes pour observer la formation spontanée de groupes, à partir d'agents initialement répartis de manière aléatoire. D'autres règles peuvent ensuite être ajoutées : points d'attraction ou de répulsion, suivi de trajectoires, contournement d'obstacles, vents et courants, etc.

La notion de voisinage dépend de la distance et éventuellement de l'orientation. Un *boid* n'est par exemple influencé que par des *boids* suffisamment proches et dans son champs de vision (pas derrière lui).

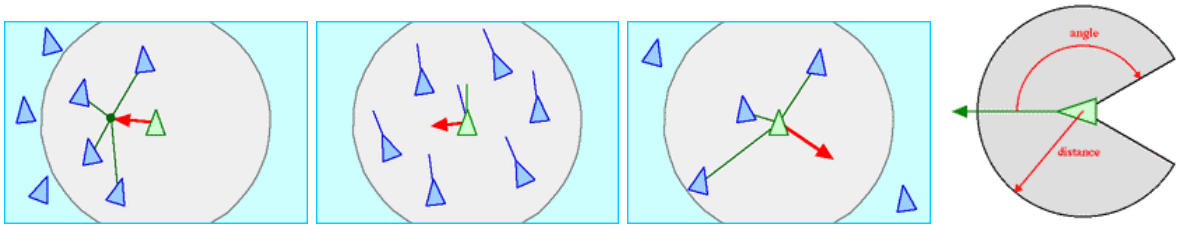

FIGURE 8 – Illustration des principales règles régissant les mouvements de *boids*. De gauche à droite : cohésion, alignement et séparation. A droite : notion de voisinage, en fonction d'une distance et de l'orientation. Figures issues de la page de Craig Reynolds `http://www.red3d.com/cwr/boids/`

4.1 Programmation d'un système de *boids*

La programmation d'un système de *boids* est simple. Elle peut être vue comme la résolution d'un système dynamique basé sur les lois de Newton, avec des hypothèses très simplificatrices (schéma explicite, pas de temps égal à 1, masses des boids égales à 1, etc.). En résumé :

— A chaque pas de temps n, un agent est caractérisé par deux vecteurs : sa position x_n et sa vitesse x'_n (qui définit de fait son orientation).

— Chacune des règles régissant l'essaim génère une « force » f_i sur chaque agent. Le calcul peut dépendre des propriétés de l'agent x_n et x'_n et de celles des agents dans son voisinage Vx_n et Vx'_n. Au final, la force $f = \sum f_i$ est appliquée sur l'agent pour le diriger.

— La position et la vitesse d'un *boid* sont alors directement calculés par :

$$x''_{n+1} = \frac{1}{m} \sum_i f_i(x_n, x'_n, Vx_n, Vx'_n) \qquad \text{accéleration (règles)}$$

$$x'_{n+1} = x'_n + x''_{n+1} \qquad \text{vitesse}$$

$$x_{n+1} = x_n + x'_{n+1} \qquad \text{position}$$

La partie intéressante est donc dans la définition des différentes règles de groupe f_i appliquées aux agents. Plutôt que de les décrire ici, nous vous indiquons les références suivantes :

— le papier original de Reynolds (1987) ;

— le site *Boids Pseudocode* de Parker (2007), qui décrit très simplement toutes les règles principales ;

— l'excellent ouvrage *The Nature of Code* de Shiffman *et al.* (2012), dont un chapitre est entièrement consacré aux agents autonomes de type *boids* (et bien plus, en fait) [3]. Il est accessible en ligne : `http://natureofcode.com/book/chapter-6-autonomous-agents/`.

— d'autres ressources, très nombreuses sur le web.

Vous pouvez utiliser directement les règles de comportement d'un essaim décrites dans ces sites, ou bien sûr les modifier ou en créer de nouvelles. Soyez inventifs !

3. plus que du pseudo-code, vous y trouverez presque une classe `Boid` de base ! Mais rassurez-vous, il reste du travail...

Question 8 Implantez un simulateur de *boids*, en utilisant le cadre de simulation précédent.

☞ *Commencez avec des règles simples. En particulier vous pouvez au moins au début limiter la notion de voisinage à un critère de distance, sans prendre en compte l'orientation des agents.*

☞ *Même si les* boids *sont très différents des agents cellulaires, des similarités existent tout de même. Comme précédemment, essayez de* généraliser *les parties communes entre les différentes classes pour réutiliser le maximum de code. Si c'est trop difficile, la seule partie commune sera au minimum de rentrer dans le contexte de notre simulateur ; et c'est déjà ça !*

4.2 La cohabitation, c'est plus difficile...

L'objectif va maintenant être de faire cohabiter plusieurs groupes de *boids*, aux comportements différents, dans la simulation. Les règles peuvent varier, mais aussi la fréquence de mise à jour des positions et vitesses, ou les interactions entre agents d'essaims différents. Par exemple, on peut imaginer des bancs de petits *boids*, type sardine, qui vivent en groupe de manière autonome. Mais il peut exister des *boids* plus gros, prédateurs, potentiellement plus lents mais très friands de ces agacants petits poissons... La chasse est ouverte !

4.2.1 Un gestionnaire à évènements discrets

Jusque ici, la gestion du temps dans le simulateur était très simple : chaque appel à `next()` consistait à passer « au pas de temps suivant ». Le problème avec plusieurs groupes de *boids* est que la fréquence avec laquelle les règles sont recalculées n'est pas forcément identique pour tout le monde. On peut imaginer des agents vifs (*e.g.* des étudiants) qui réagissent beaucoup plus vite que d'autres (*e.g.* leurs enseignants, qui ne se mettent à jour par exemple qu'un pas de temps sur cinq).

Même si d'autres solutions seraient possibles (par exemple avec des *threads* Java, chacun gérant les actions d'un groupe d'agents), il est ici proposé de centraliser le problème en utilisant un *gestionnaire à événements discrets*. Ce gestionnaire possède une séquence ordonnée d'événements datés (par un entier par exemple, ou une classe `Date`). À chaque évènement est associée une action à réaliser. Les évènements sont ajoutés dans un ordre quelconque, mais bien traités à la date adéquate.

En pratique, le gestionnaire maintient une « date courante » de simulation. Lorsque la méthode `next()` est invoquée, cette méthode incrémente alors la date courante puis exécute dans l'ordre tous les évènements non encore exécutés jusqu'à cette date. `isFinished()` retourne `true` si plus aucun évènement n'est en attente d'exécution.

Un diagramme de classes est proposé figure 9. La classe `Event` est abstraite, elle devra être héritée par des sous-classes qui représenteront des évènements réels avec leurs propres propriétés et définiront la méthode `execute()` de manière adéquate.

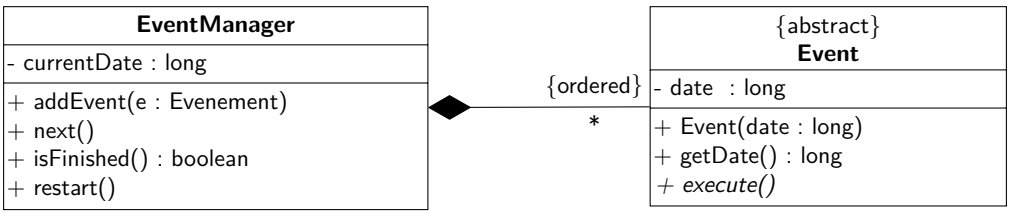

FIGURE 9 – Diagramme de classes d'un gestionnaire à événements discrets

Un exemple d'utilisation d'un tel gestionnaire d'évènements discrets est présenté en annexe A.

Question 9 Implantez les classes `Event` et `EventManager` selon les spécifications décrites ci-dessus. Elles peuvent être testées avec les codes donnés figures 10 et 11.

☞ *Pour la gestion ordonnée des évènements, pensez à regarder du côté des* Collections *Java afin de vous simplifier la tâche.*

4.2.2 Modification de votre simulateur

Il s'agit maintenant d'intégrer le gestionnaire d'évènements dans votre simulateur.

Si vous avez correctement séparé les systèmes de calcul (cellulaires, *boids*) de leur interface avec le simulateur, il n'y a que très peu de modifications à effectuer. Sur l'exemple de la figure 1, seule la classe `BallsSimulator` est à modifier pour intégrer un gestionnaire d'évènements. Au lieu d'invoquer directement les méthodes de mise à jour des balles, la méthode `next()` doit simplement indiquer au gestionnaire d'évènements d'incrémenter sa date et d'exécuter les évènements adéquats. Les appels aux méthodes des systèmes d'agents sont en fait réalisés dans les méthodes `execute()` des évènements.

Vous l'avez compris, un événement est par nature ponctuel et non répété. Or, si l'on veut une animation qui « dure » (typiquement des cercles qui ne s'arrêtent jamais de bouger, sauf lorsque l'on clique sur le bouton d'arrêt) il faut donc un mécanisme permettant de créer de nouveaux événements à la volée, lors de la simulation. La seule possibilité pour cela est que ce soient les événements eux-mêmes qui créent les nouveaux événements et les postent dans le simulateur : un événement de translation à l'instant t va d'abord effectuer la translation, puis créer un nouvel événement à la date $t + 1$ et le poster dans le simulateur afin qu'il soit exécuté au pas suivant. Ainsi, l'animation est perpétuelle.

Question 10 Intégrez un gestionnaire d'évènements à votre simulateur. Tous les systèmes multiagents précédents doivent continuer à fonctionner de manière identique.

☞ *Attention à garder un code fonctionnel de ce que vous avez fait jusqu'à présent, au cas où...*

4.2.3 Simulation de plusieurs groupes de *boids*

Tout est prêt maintenant pour faire cohabiter différent groupes de *boids*. Chaque groupe sera géré par ses propres évènements, avec un pas de temps spécifique entre l'exécution de deux évènements.

Question 11 Créez quelques groupes de *boids* aux comportements différents. Ils peuvent ne pas suivre les mêmes règles d'organisation, ou bien avoir des règles spécifiques.

☞ *Bien entendu, il est demandé de proposer une hiérarchie de classes permettant de généraliser au maximum ce qui est commun entre tous les groupes.*

Question 12 Pour aller plus loin, vous pouvez ensuite faire interagir des groupes entre eux (par exemple des proies/prédateurs). Il s'agit alors de correctement gérer les ensembles de *boids* qui doivent être connus pour la mise à jour de chaque agent particulier.

5 Livrable attendu

Le travail rendu fera l'objet d'une évaluation par les pairs, on vous demande de ne pas mettre vos noms dans les différents fichiers, mais votre numéro d'équipe teide. Les critères pour l'évaluation par les pairs correspondront à ce qui est décrit dans ce document ainsi que ce que vous avez vu pendant les séances encadrées.

L'application rendue devra répondre aux spécifications des différentes parties ci-dessus. Si toutes les contraintes ne sont pas prises en compte, bien le spécifier dans le rapport. En plus de ceci, quelques exigences non fonctionnelles sont attendues :

- le code rendu devra être propre, bien structuré, et respecter le *coding style* Java (voir le lien sur la page `Chamilo` ... et tout ce qui est fait en cours !) ;
- en plus de noms de variables explicites, les aspects « techniques » de votre code devront être commentés, pour en faciliter la compréhension ;
- le principe d'encapsulation devra être respecté : masquage des attributs, garantie de l'intégrité des états des objets, principe de délégation ;
- utilisez l'héritage pour factoriser tout code nécessaire à plusieurs objets, et spécifiez des méthodes abstraites dans les classes de haut niveau [4] ;

4. si vous avez à utiliser `instanceof` en dehors de la redéfinition d'une méthode `equals(Object o)`, il y a généralement un problème de conception objet...

— même s'il n'est pas demandé de tests exhaustifs, votre application devra être le plus robuste possible ;
— il n'est pas demandé de renseigner l'API des classes de manière exhaustive. Néanmoins, une documentation minimale des classes et méthodes principales peut être rédigée. Les tags `/** ... */` seront utilisés, pour générer la documentation en `html` ou `pdf` à l'aide de l'application `javadoc`.

Le livrable final sera déposé sous `teide` sous la forme d'une archive `tar.gz` contenant :
— le code source de votre application ;
— un document au format `pdf` de 4 pages maximum expliquant et justifiant vos choix de conception, l'utilisation à bon escient des classes et des méthodes les plus adaptées. Vous décrirez également les principaux test effectués et résultats obtenus.

Références

Conrad Parker : Boids pseudocode, 2007. URL http://www.vergenet.net/~conrad/boids/pseudocode.html.

Craig W Reynolds : Flocks, herds and schools : A distributed behavioral model. *In ACM Siggraph Computer Graphics*, volume 21, pages 25–34. ACM, 1987. URL http://www.red3d.com/cwr.

Thomas Schelling : Dynamic models of segregation. *Journal of Mathematical Sociology*, 1 :143–186, 1971.

Daniel Shiffman, Shannon Fry et Zannah Marsh : *The nature of code*. D. Shiffman, 2012. URL http://natureofcode.com.

Annexes

A Gestionnaire d'évènements discrets

Les figures suivantes présentent un exemple de classe d'évènement, son utilisation dans un gestionnaire à évènements discrets et la trace résultante.

```java
public class MessageEvent extends Event {
    private String message;

    public MessageEvent(int date, String message) {
        super(date);
        this.message = message;
    }

    public void execute() {
        System.out.println(this.getDate() + this.message);
    }
}
```

FIGURE 10 – Exemple de classe représentant un événement héritant le modèle `Event` présenté figure 9. Ici il ne s'agit que d'afficher un message dans la console.

```java
public class TestEventManager {
    public static void main(String[] args) throws InterruptedException {
        // On crée un simulateur
        EventManager manager = new EventManager();

        // On poste un événement [PING] tous les deux pas de temps
        for (int i = 2; i <= 10; i += 2) {
            manager.addEvent(new MessageEvent(i, " [PING]"));
        }
        // On poste un événement [PONG] tous les trois pas de temps
        for (int i = 3; i <= 9; i += 3) {
            manager.addEvent(new MessageEvent(i, " [PONG]"));
        }

        while (!manager.isFinished()) {
            manager.next();
            Thread.sleep(1000);
        }
    }
}
```

FIGURE 11 – Un exemple de code illustrant le fonctionnement d'un gestionnaire d'évènements, à l'aide d'un scénario fixé à l'avance (ici l'ajout d'événement « `[PING]` » tous les deux pas de temps, et d'un « `[PONG]` » tous les trois pas de temps)

```
Next... Current date : 1
Next... Current date : 2
2 [PING]
Next... Current date : 3
3 [PONG]
Next... Current date : 4
4 [PING]
Next... Current date : 5
Next... Current date : 6
6 [PING]
6 [PONG]
Next... Current date : 7
Next... Current date : 8
8 [PING]
Next... Current date : 9
9 [PONG]
Next... Current date : 10
10 [PING]
```

FIGURE 12 – Trace d'exécution de la simulation spécifiée dans le code de la figure 11.