Cahier des charges projet IA04 : Flotte de drones en 2D

Sommaire

1 Présentation globale de l'idée

1.1 Définition d'un drone

Un drone (de l'anglais drone) désigne un aéronef sans pilote à bord. Le drone peut avoir un usage civil ou militaire. [...] Sa taille et masse (de quelques grammes à plusieurs tonnes) dépendent des capacités recherchées. Le pilotage automatique ou à partir du sol permet des vols longs de plusieurs dizaines d'heures (à comparer aux deux heures typiques d'autonomie d'un chasseur).
- Wikipédia : https://fr.wikipedia.org/wiki/Drone

1.2 Modélisation

On suppose qu'un agent représente un drone et on se place dans un plan 2D (xOy). Chaque drone est assimilé à un point qui représente sa position (x, y) dans le plan. L'objectif est de faire en sorte que les agents coexistent dans un périmètre donné. Les coordonnées sont positives, on se place dans $\mathbb{R}_+^2$. L'espace de déplacement (l'environnement) n'est pas torique et est limité par des bornes en x et y. L'espace n'est pas forcément carré.

1.3 Comportement des drones

La coexistence des drones est basée sur plusieurs règles de fonctionnement :

1. Les drones ont tendance à s'organiser en flotte, le nombre de drones maximal dans une flotte est une constante à définir au début du programme.
2. Les drones sont démarrés seuls au départ et ne font pas partie d'une flotte.
3. Chaque drone a un unique identifiant (qui dans JADE je suppose correspondra en fait à son AID).
4. Chaque flotte a un un maître qui guide sa flotte.
5. Les flottes sont en anneau, le diamètre peut varier en fonction des objets ou autres drones sur leur passage, de manière donc à éviter les collisions.
6. La communication des drones se fait par ondes radio à courte portée, ce qui concrètement veut dire qu'un drone ne peut émettre qu'à un certain rayon autour de lui, et de même ne recevoir qu'à une distance inférieure ou égale à ce rayon.
7. Lorsqu'un drone (ou une flotte) rencontre un autre drone (ou une autre flotte), une procédure est activée de manière à ce que les deux ensembles fusionnent (ou non, en fonction du nombre de drones déjà présents dans la flotte).
8. Lorsqu'un drone est détruit (par exemple il s'écrase contre un bâtiment, représenté par un polygone sur le plan), il est retiré du graphique (le point qui le représentait est effacé).
9. Inversement lorsqu'un drone est créé, le point le représentant apparait sur le graphique.
10. La communication utilisée dans une flotte est une communication de proche en proche (topologie en anneau) : https://fr.wikipedia.org/wiki/Topologie_de_reseau#Le_r.C3.A9seau_en_anneau

1.4 Objectif

L'idée est donc de modéliser ce système et d'y incorporer une **interface graphique** (animation) afin de visualiser en temps réel les déplacements et les différents interactions des drones.

Voici une image de ce à quoi cela pourrait ressembler (ce serait donc une capture à un instant donné de la figure graphique) :

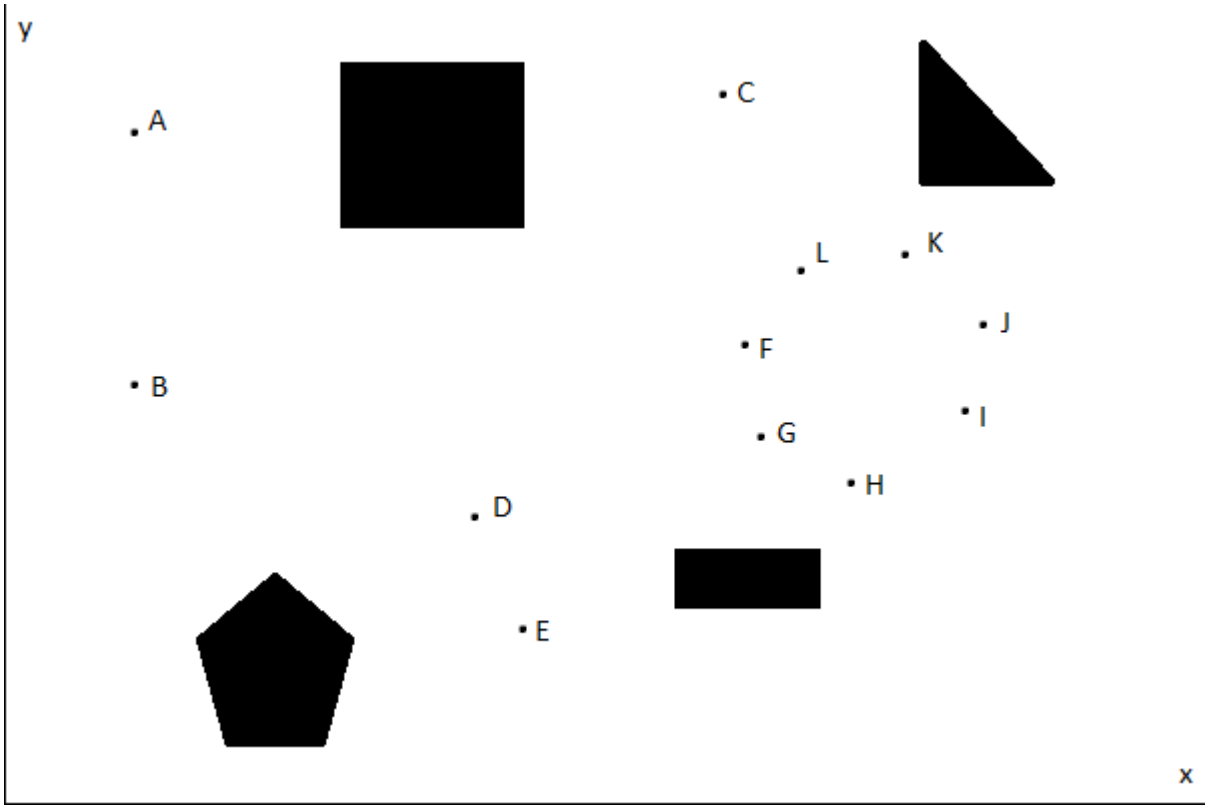

FIGURE 1 – Les points A, B, ... sont des drones et les polygones noirs sont des objets (facultatif)

2 Implémentation

2.1 Architecture JADE

L'architecture se divise en trois parties : les drones qui évoluent dans l'environnement 2D, les objets qui se trouvent sur le terrain, et l'interface graphique qui va représenter tout cela sur l'écran. Ces trois parties sont plus ou moins indépendantes.

2.2 La classe `Drone`

2.2.1 Description

La classe `Drone` est la principale classe à implémenter. En termes de cardinalité, le nombre d'instances de cette classe peut varier de 1 à n.

2.2.2 Attributs

La liste d'attributs de la classe `Drone` :

1. La position actuelle du drone dans le plan $p = (x, y)$ qui est représentée par une instance d'une classe à définir, qui définira un certain nombre de méthodes (par exemple renvoyer la distance entre deux points du plan).
2. Un identifiant unique qui peut éventuellement être l'AID du drone.
3. Un état à définir par la suite, notamment lors des fusion de flotte, il faudra imposer aux drones de ne plus se déplacer et d'ignorer les messages reçus de l'environnement, donc il leur faudra un état particulier qui les mettra en stand-by.
4. La prochaine position du drone (calculée en fonction de la position actuelle, de la position objectif, et de l'environnement immédiat).
5. La position objectif du drone.
6. La flotte à laquelle il appartient, s'il appartient à une flotte.
7. Son rang dans la flotte et le voisin auquel il doit envoyer un message (anneau unidirectionnel), s'il appartient à une flotte.

2.2.3 Behaviours

Le drone doit pouvoir réagir à son environnement de plusieurs manières, dépendamment des stimuli qui lui parviennent. Voici la liste des behaviours de la classe `Drone` :

1. Un behaviour qui renvoie la position de l'agent à l'agent `Display` quand il le lui demande.
2. Un behaviour qui envoie spontanément un message à l'agent `Display` lors de la mort de l'agent (l'agent `Display` devra dans ce cas supprimer le drone de la liste des drones et mettre à jour l'affichage).
3. Un behaviour qui réagit aux messages provenant de d'autres éléments de la classe `Drone`.
4. Un behaviour qui envoie périodiquement des messages dans l'environnement immédiat de l'agent (il se contente d'émettre aux autres agents, la réception du message se fera par une fonction de filtration sur l'attribut position).
5. Un behaviour qui réagit aux messages provenant des objets (dans cette modélisation, les objets sont des ensembles de points fixes dont les points périphériques envoient périodiquement des messages aux drones).

2.3 La classe `Display`

2.3.1 Description

La classe `Display` est une classe indépendante de la classe `Drone`. Son rôle est de rappatrier les positions des drones et de les afficher sur une interface graphique. L'implémentation de classe passe donc par l'utilisation d'une librairie graphique Java. En termes de cardinalité, le nombre d'instances de cette classe est exactement de 1. C'est la première classe créée lors du lancement du programme, et elle crée les autres objets dès son démarrage. Elle initialise les position des drones (qui peuvent éventuellement être passées en paramètres) et des objets.

2.3.2 Attributs

La liste d'attributs de la classe `Display` :

1. Un tableau dynamique (du fait de la suppression possible de drones au cours de l'évolution de la simulation) recensant les drones existants.
2. Le nombre de drones existants.
3. Un tableau dynamique recensant les objets sur le terrain.
4. Le nombre d'objets existants.
5. La dernière représentation graphique calculée (matrice dépendant de la librairie graphique utilisée, paramètre à déterminer).

2.3.3 Behaviours

Voici la liste des behaviours de la classe `Display` :

1. Un behaviour qui envoie périodiquement à tous les agents `Drone` une demande de leur position. Il met à jour le tableau de positions.
2. Un behaviour qui reçoit un signal de mort de la part d'une drone, dans ce cas il devra le supprimer de sa liste de drones.
3. Un behaviour qui répond un signal d'arrêt en provenance de la console, il supprimera tous les drones et objets et terminera l'exécution.
4. Un behaviour qui rafraîchit constamment l'interface graphique sur la base des positions actuelles des drones.

2.4 La classe `Object`

2.4.1 Description

La classe `Object` est facultative. Il faut prioriser le développement des deux classes précédentes. Si les deux classes précédentes sont terminées en avance et que la simulation fonctionne, on peut passer à cette classe. Elle représente les objets qui se trouvent sur le terrain. Les objets sont des polygones. Un objets est un ensemble de points dont les points frontaliers sont ceux qui émettent des messages.

2.4.2 Attributs

La liste d'attributs de la classe `Object` :

1. Un tableau de points représentant la totalité des points qui le composent.
2. Un identifiant unique.

2.4.3 Behaviours

La liste des behaviours de la classe `Object` :

1. Un behaviour qui émet constamment des messages (destinés aux agents) dans les environs de l'objet. Seuls les points frontaliers doivent émettre des messages. Il faudra donc une méthode de classe qui définit la frontière de l'objet.
2. Un behaviour qui renvoie l'ensemble des points le constituant à l'agent `Display` lorsqu'il le lui demande.

3 Communication des drones

Dans cette partie, on va définir les différents cas de figure qui peuvent se présenter dans la vie d'un drone et l'implémentation et les algorithmes qui en découlent. On définira également d'un point de vue technique la nature des messages échangés et l'état des drones au cours de ces situations.

3.1 Déplacement sans flotte

.

3.2 Crash contre un objet

.

3.3 Croisement d'un drone seul

.

3.4 Croisement d'un drone ayant une flotte

.

3.5 Intégration d'un drone dans une flotte

.

3.6 Fusion de deux flottes

.

3.7 Destruction d'un élément de la flotte

.

3.8 Déplacement en flotte

.

4 Organisation du code

4.1 Conventions de nommage et d'écriture

Pour que la rédaction du code source soit cohérente bien qu'elle soit faite par différents contributeurs, on va définir quelques règles d'écriture.

4.1.1 La langue

Toutes les noms de variables, de classes, d'attributs, de méthodes, etc. seront en anglais.

4.1.2 Les identificateurs

On n'utilisera des abréviations que lorsque c'est réellement utile (mot trop long). Par exemple, si on attrape une exception, on utilisera de préférence le mot entier `exception` plutôt que simplement `e`. Ceci afin d'éviter qu'on se retrouve avec des noms de variables qui n'ont aucune signification dans le code. Un nom de variable clair peut aider à comprendre le code.

4.1.3 Les séparateurs

On utilisera la convention **CamelCase** et non **underscore**. Par exemple, au lieu d'écrire `une_variable`, on écrira `uneVariable`.

4.1.4 Les attributs

Les attributs de classe doivent être distingués des autres variables afin de ne pas les confondre dans les méthodes. Ils seront tous préfixés de `m_` (pour *member*). Par exemple `m_position`.

4.1.5 Les classes

Les noms de classes démarrent par une majuscule, le reste en minuscule, sauf dans le cas des acronymes (sigles) qui ont plusieurs lettre en majuscules.

4.1.6 Les méthodes

Les méthodes (et fonctions, mais il n'y en a pas en Java) commenceront par une minuscule afin de ne pas les confondre avec une classe.

4.1.7 Les variables

Les variables démarreront par une minuscule.

4.1.8 Les accolades

Écrire de préférence :

```
public void method()
{
}
```

plutôt que :

```
public void method(){
}
```

La première version est plus claire, notamment quand plusieurs accolades et indentations s'imbriquent.

4.1.9 Les espaces

De préférence (toujours dans un souci de clarté pour que tout le monde en profite et que le code soit rapidement abordable) :

```
for(int i = 0 ; i < 10 ; i ++)
{
    x = x + ((y * 10) % 3)
}
```

plutôt que :

```
for(int i=0;i<10;i++)
{
    x=x+((y*10)%3)
}
```

4.1.10 Les sauts de lignes

Le nombre de saut de lignes et leur position est assez subjectif. Viser la clarté et des paquets d'instructions cohérents. Par exemple, il vaut mieux écrire :

```
public void method()
{
    m_x *= 5;
    m_y *= 3;

    for(int i = 0 ; i < 10 ; i ++)
    {
```

```
        m_x = (m_x * i) % 2;
        m_y = (m_y * i) % 3;

        System.out.println(m_x + m_y);

        if(i == 5)
            System.out.println("i = 5 !");
    }
}
```

que :

```
public void method()
{
    m_x *= 5;
    m_y *= 3;
    for(int i = 0 ; i < 10 ; i ++)
    {
        m_x = (m_x * i) % 2;
        m_y = (m_y * i) % 3;
        System.out.println(m_x + m_y);
        if(i == 5)
            System.out.println("i = 5 !");
    }
}
```

4.1.11 Les indentations

Il faut suivre la logique des `if`, des `for`, etc. Par exemple, il vaut mieux écrire :

```
for(int i = 0 ; i < 10 ; i ++)
{
    if(i == 5)
        for(j = 0 ; j < 5 ; i++)
            if(j == 3)
                System.out.println("i = 5 et j = 3");
}
```

plutôt que :

```
for(int i = 0 ; i < 10 ; i ++)
{
    if(i == 5)
    for(j = 0 ; j < 5 ; i++)
    if(j == 3)
    System.out.println("i = 5 et j = 3");
}
```

Il faut également éviter les instructions mono-ligne que certains utilisent parfois mais qui ne servent à rien sinon à rendre la lecture et compréhension du code plus difficile :

```
1 for(int i = 0 ; i < 10 ; i ++){if(i == 5){System.out.println("i = 5 et j = 3");}}
```

4.2 Commentaires

Il faut commenter chaque partie de code qui nécessite une explication. Les commentaires seront en français, parce que ce sera plus simple pour tout le monde, surtout si les explications sont complexes.

4.3 Les constantes

Il faudra mettre toutes les constantes (comme par exemple le nombre de drones quand on lance le programme, la taille de la fenêtre graphique, etc.) dans le fichier `Constants`.

4.4 Description des classes

4.4.1 `MainContainer`

C'est le conteneur principal, c'est lui qui contient les agents drones, affichage, etc. Ce conteneur initialise l'agent `Display`, ce dernier prendra la main et créera ensuite les drones (et éventuellement les objets si toute la partie simulation des drones est terminée en avance et fonctionne).

4.4.2 `Display`

C'est la classe qui crée les drones et met à jour l'interface graphique. Elle possède un seul behaviour, qui demande périodiquement aux drones leur position.

4.4.3 `Drone`

C'est la classe représentant le drone.

4.4.4 `Position`

C'est une classe simple représentant une position (x, y) sur le terrain. En pratique, l'implémentation est telle qu'une position est en fait une cellule d'une grille (le terrain est découpé en cellules, voir les commentaires du code).

4.4.5 `Constants`

Cette classe recense les constantes du programme et les méthodes appelées dans plusieurs classes.

4.4.6 `GUI`

GUI signifie *Graphical User Interface*, c'est la classe qui implémente l'interface graphique. Elle utilise la librairie **sdljava** (voir sur google). Tel que le code a été fait (et c'était le seul moyen de faire fonctionner la GUI avec les autres agents), elle hérite de la classe `Agent` et possède une référence sur `Display`. Elle se rafraichit constamment en fonction de l'état actuel du tableau de drones que possède `Display` **(elle ne communique donc pas par messages)**.