Simulation d'un drone multi-rotor dans son environnement

Lucie MATHÉ

6 novembre 2019

Table des matières

1 Bases du simulateur **2**

2 Messages **3**
2.1 Mise en forme 3
2.2 Fonctionnement 4

3 Version 1 **5**
3.1 Serveur 5
3.2 Environnement 6
3.2.1 Building 6
3.2.2 Drone 6
3.2.3 Environnement 6
3.2.4 Main 7
3.3 Station Sol 7
3.4 Drone 8

4 Extensions **9**
4.1 Un drone intelligent 9
4.1.1 Communication 10
4.1.2 Autopilote 10

5 Limites **13**
5.1 Problèmes connus 13
5.2 Conclusion 13

Chapitre 1

Bases du simulateur

L'objectif de ce TD est de développer un simulateur de drone. Pour cela, il faut créer 4 entités permettant de faire fonctionner ce simulateur. Ces caractéristiques ont été définies en groupe avec la classe.

- **Environnement** : un tableau de cases en 3 dimensions. L'environnement contient un drone et des bâtiments qu'il faudra éviter. Il reçoit comme information les déplacements du drone et renvoie les cases changées du tableau.
- **Drone** : un drone se déplace de case en case, chaque déplacement se fait sur une distance de 1. Il possède une 6-connexité. Il reçoit une position à atteindre et renvoie sa nouvelle position.
- **Station Sol** : La station sol représente l'interaction humaine dans ce simulateur. Dirigée par l'utilisateur, elle envoie des positions à atteindre au drone. À chaque tour, elle est informée de l'état de son drone (position actuelle, en fonctionnement ou mort).
- **Moteur du simulateur** : Autrement dit le serveur, il est chargé de transmettre les messages aux trois autres entités du simulateur. Il contient l'observateur permettant de gérer les collisions du drone. J'ai préféré inclure l'observateur dans le moteur du simulateur comme indiqué dans la feuille de TD, au lieu de créer une cinquième entité.

Chapitre 2

Messages

2.1 Mise en forme

Chaque entité et action possède un trigramme permettant de raccourcir la longueur des messages. Ainsi on retrouve :

- **Environnement** : ENV
- **Drone** : D + identifiant ("D01" pour le 1er drone)
- **Station Sol** : GCS (Ground Control Station)
- **Moteur du simulateur** : SRV (en référence au rôle de serveur)
- **Observateur** : OBS (même si celui-ci est dans le serveur, on note OBS si on s'adresse à lui)

Chaque élément du messages est séparé par **":"** et se termine par **"\n"**. Le premier élément du message est l'expéditeur suivi du destinataire puis de l'action et des éléments relatifs à l'action.

Exemple : *"DO1:SRV:action"*

2.2 Fonctionnement

Toutes les entités se connectent en envoyant un message : *"ENV:SRV:CONNECT"*.

Conformément à ce qu'on a discuté avec la classe, le simulateur fonctionne en tour par tour. Chaque tour fonctionne toujours de la même façon, les entités traitent leur message à tour de rôle.

GCS->D01->ENV->OBS->GCS->...

Le serveur transmet ces messages, en réalité nous avons donc :

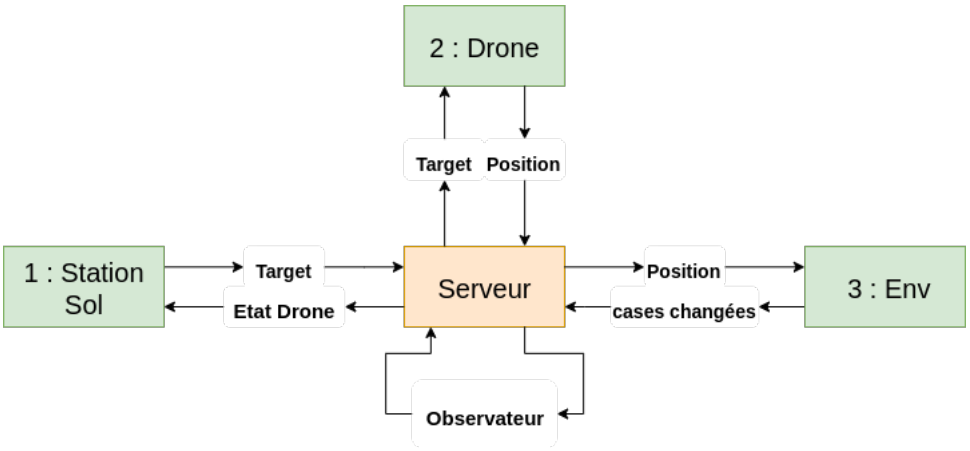

FIGURE 2.1 – Architecture du cycle

On peut différencier le serveur en orange et les différents clients. L'observateur étant dans le serveur il n'y pas de message envoyé pour qu'il traite les données de l'environnement. J'ai voulu quand même le montrer sur ce schéma.

Le simulateur commence lorsque 3 clients sont connectés. La première action effectuée est l'envoi de la target au drone par la station sol.

À la fin de son tour, l'entité indique au serveur qu'elle a fini : *"ENV:SRV:DONE"*

Chapitre 3

Version 1

La version 1 consiste à implémenter un simple drone se déplaçant dans l'environnement.

3.1 Serveur

Le serveur a été implémenté en Python (attention, je suis sous un environnement virtuel de travail Python3). J'ai fait ce choix car n'ayant pas fait de serveur depuis longtemps, j'ai choisi le langage qui m'a semblé le plus accessible pour faire du multi-threading permettant de gérer les différents clients.

Au lancement du serveur, celui-ci attend que les trois entités se connectent avant de commencer l'envoi des messages. Si deux entités de même type se connecte alors le serveur refuse la seconde. Lorsqu'une entité se connecte, il lance la thread associée au comportement attendu de l'entité. Ainsi, on va pouvoir différencier les comportements en fonction des messages. Il indique au client le rôle qu'il a en envoyant *"You are connected as* + role".

Pour envoyer des messages, on crée une liste de client, lorsqu'une entité arrive, on stocke la socket du client. J'ai créé une fonction BROADCAST prenant en argument le message et le destinataire. La fonction permet d'envoyer le message au destinataire choisi en récupérant la socket dans la clientList.

Si le drone meurt, le serveur ferme tous les clients grâce à la fonction GAMEOVERFUN. Le message est alors : *"SRV:entité:OVER"*. En revanche, le serveur ne se ferme pas de façon propre mais il s'arrête sur une erreur de lecture de message.

3.2 Environnement

L'environnement a été implémenté en C++. Pour moi, un langage objet semblait évident pour l'environnement car il permet de créer une classe pour chaque objet de l'environnement. J'ai choisi le C++ au Java car c'est le langage avec lequel je suis le plus à l'aise. Ainsi, on peut retrouver les classes `Drone`, `Building` et `Environnement`. Nous retrouvons également un `main` permetttant de gérer le client.

3.2.1 Building

La classe Building est relativement simple : un immeuble se définit par une position de base en x, y, une longueur (en x), une largeur(en y) et une hauteur (en z).

En dehors des getters, il contient seulement deux fonctions :

— INTERSECT : lors de la création de l'environnement, on crée des immeubles de façon aléatoire. Cette fonction permet de vérifier que deux Buildings ne s'intersecte pas.

— ISINSIDE : de façon plus simple, la fonction isInside permet de vérifier si un point(x, y, z) appartient à un immeuble.

3.2.2 Drone

Un drone se définit par une position (x, y, z) et une direction (non utilisée pour l'instant mais prévu au départ donc je l'ai laissée). Les deux fonctions implémentées sont la translation jusqu'à un nouveau point et la rotation.

3.2.3 Environnement

L'environnement est un espace 3D contenant une liste d'immeuble et un drone. Pour des questions de facilité, l'environnement est cubique de taille 10. À l'initialisation de l'environnement, celui-ci reçoit les éléments fixes : sa taille et les immeubles. Ensuite, il contient différentes fonctions :

— ADDDRONE : ajoute un drone à l'environnement

— WHATSINSIDE : permettant de récupérer le contenu d'une case

— DRAWENV : affiche l'état actuel de l'environnement sur le terminal, hauteur par hauteur.

— MOVEDRONE : déplace le drone dans l'environnement et renvoie les cases changées (qui sont ensuite transmises à l'observateur)

Dans le dessin de l'environnement, on commence en haut à gauche par le point (0,0). Ensuite, le décalage d'une case à droite nous fait augmenter de 1 en y et le décalage d'une case en bas nous fait augmenter la position de 1 en x. L'environnement est affiché étage par étage en 2D. Par exemple, sur un environnement de taille 2 nous avons pour l'étage 0 :

(0,0)	(0,1)
(1,0)	(1,1)

TABLE 3.1 – Positions en 2D dans l'affichage de l'environnement

3.2.4 Main

Au démarrage de l'environnement, le main crée la carte de façon aléatoire. On crée donc une liste d'immeuble (on définit une longueur\largeur maximum pour les immeubles).

Une fois l'environnement initialisé, on crée la socket qui se connecte au serveur. Nous créons le drone au premier message reçu de celui-ci. Ensuite, nous déplaçons le drone en fonction des positions qu'il nous envoie. Pendant le jeu, l'environnement ne reçoit des messages que du drone.

Si celui-ci meurt, le serveur envoie OVER et la socket se ferme.

L'environnement envoie des messages adressés à l'observateur lui indiquant les cases changées au cours du tour. Pour chaque case changée, il envoie donc : *"ENV:OBS:UPD:x:y:z:content"*. UPD signifie update, suivi de la position en 3 dimensions de la case et enfin son contenu sous forme d'un `std::string`.

3.3 Station Sol

La Station sol a été implémentée en Java. J'ai choisi le Java car la Station Sol me paraissait assez simple et le Java est le langage avec lequel je suis le moins à l'aise. L'intéraction avec l'utilisateur est également assez simple en Java.

La Station Sol affiche à l'utilisateur le message reçu de l'environnement. Au début du jeu, la station sol force l'utilisateur à donner une position à atteindre au drone qu'il contrôle. Ensuite, elle propose à chaque tour de changer cette target. Ce choix que doit faire l'utilisateur permet de rythmer le tour car on attend que l'utilisateur réponde.

J'ai créé une fonction CORRECTFORM permettant de vérifier que la position donnée à l'utilisateur correspond au critère du jeu : (x, y, z) qui appartient à l'environnement.

On garde en mémoire la dernière target envoyée pour la renvoyer au drone si celle-ci ne change pas. On renvoie le message au drone pour que celui-ci ait toujours un message à lire au début de son drone. Le message au drone est donc : *"GCS:D01:TGT:x:y:z"*.

3.4 Drone

Il me restait donc le C pour construire le drone. La V1 étant très simple, je n'ai vu au départ aucun problème à coder le drone en C.

Le drone de la V1 est donc peu intelligent, il se contente d'avancer dans la case qui le rapproche le plus de sa position à atteindre. Il peut totalement foncer dans un batiment. Pour des questions de facilité, un drone commencera toujours en (0,0,0). À chaque tour, on récupère la target et on avance depuis la dernière position envoyée.

On renvoie donc la position (x, y, z) à l'environnement avec le message : *"D01:ENV:NPS:x:y:z"* (NPS pour New PoSition).

La mort du drone représente la fin de la partie.

Chapitre 4

Extensions

4.1 Un drone intelligent

Un des objectifs de ce projet était de construire un autopilote permettant au drone de se déplacer dans l'environnement sans rentrer dans un bâtiment. Pour ne pas rentrer dans un bâtiment, ce drone possède un capteur lui permettant de voir dans les cases connexes.

Pour simuler ce capteur, le drone demande à l'environnement ce que contient la case qu'il souhaite visiter.

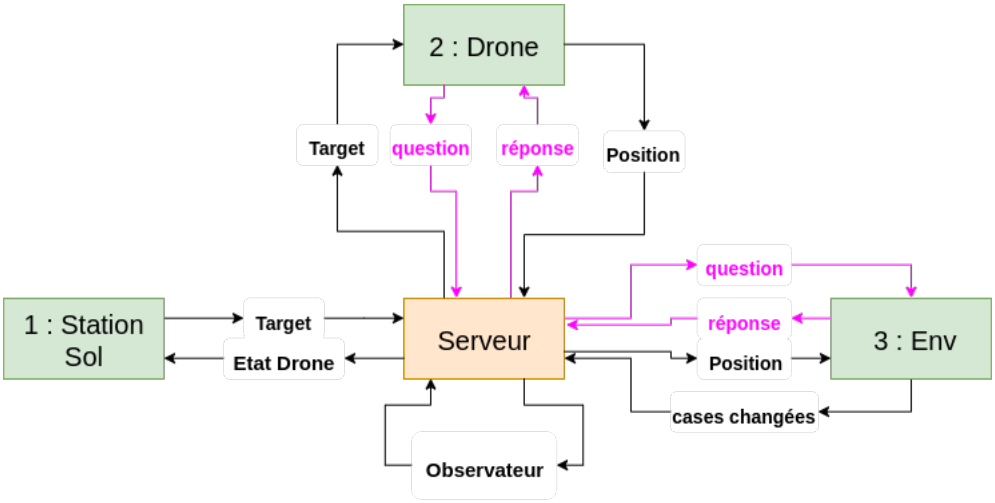

FIGURE 4.1 – Architecture du cycle avec autopilote

La partie question-réponse enrose peut-être effectuée ou non et itérée si besoin. Cette échange a lieu pendant le tour du drone, avant qu'il renvoi la position qu'il décide d'atteindre pendant ce tour.

4.1.1 Communication

Le serveur doit maintenant permettre à l'environnement de lire et écrire des messages alors que ce n'est pas sont tour de jouer (il ne doit pas s'actualiser tant que le moteur du simulateur en lui a pas indiqué, utile pour le multi-drone).

Pour cela, j'ai créé une variable globale DRONEISASKING qui passe à vrai si le pose une question à l'environnement. Dans ce cas, le drone fait une pause dans son tour, l'environnement traite le message qu'il a reçu et le drone fini son tour.

Il va traiter deux nouveaux types de messages :

— La demande du drone à l'environnement SRV : *"D01:ENV:ASK:x:y:z"*, il demande donc pour une case du simulateur.

— La réponse de l'environnement au drone : *"ENV:D01:ANS:content"*, le contenu est donc une chaine de caractère avec les différents éléments. On utilise donc la fonction WHATSINSIDE pour répondre.

4.1.2 Autopilote

Pour avoir un drone qui ne meurt pas, il suffit potentiellement de juste éviter les cases qui contiennent un bâtiment. En revanche, pour que le drone atteigne la position souhaitée, il est nécessaire de lui implémenter un système un peu plus complexe. L'autopilote est implémenté dans l'exécutable smart_drone.

Pour simplifier le problème dans une résolution 2 dimensions, le drone va d'abord s'aligner en `Z` avec sa target.

Il nous reste donc 4 directions pour le robot. On caclcule la distance entre la position souhaitée et les 4 positions possibles pour le robot. On va donc trier les cases voisines dans l'ordre de celle qui nous rapproche le plus de la destination à la plus éloignée. En cas d'égalité de distance entre deux cases, on a prédéfini un ordre pour les destinations : NORTH, SOUTH, EAST, WEST. Il ira donc dans la première destination nommée dans cer ordre. Par exemple, dans le tableau 4.1, on ira au sud en position (1,0).

D01	EPT
EPT	BLK

TABLE 4.1 – Exemple : calcul de distance

Dès qu'un robot reçoit des informations sur une case, il stocke cette case dans un tableau grâce à une structure comprenant les coordonées de la case et un booléen qui symbolise si la case est vide ou pas.

Lorsqu'un robot cherche à se déplacer, il demande des informations sur les cases d'après l'ordre prédéfini. Si une case n'a jamais été visitée, le robot demande à l'environnement s'il peut y aller. Il choisira toujours une case qu'il n'a pas encore visitée. En revanche, si toutes les cases autour de lui ont été visitées, il choisira toujours celle qui apparaîtra en premier dans son tableau de case. Ainsi, si un robot se retrouve dans un cul de sac, il peut revenir jusqu'à qu'il croise une case non visitée.

Prenons en exemple l'environnement réduit du tableau 4.2).

D01	EPT	EPT	BLK	EPT
EPT	BLK	BLK	BLK	EPT
EPT	BLK	BLK	TGT	EPT
EPT	EPT	EPT	EPT	EPT
EPT	EPT	EPT	EPT	EPT

TABLE 4.2 – Exemple en cul-de-sac, initialisation

Ici, la distance la plus courte pousse le drone à aller en (0,1). Ainsi, le drone va avancer jusqu'à être on fond de la voie sans issue. On va noter *"V + numéro de visite "* l'ordre des cases visitées.

V00	V01	V02	D01	BLK
EPT	EPT	BLK	BLK	BLK
EPT	EPT	BLK	EPT	TGT
EPT	EPT	EPT	EPT	EPT
EPT	EPT	EPT	EPT	EPT

TABLE 4.3 – Exemple en cul-de-sac, problème

Ensuite, le drone va donc reculer jusqu'à retrouver une case vide. Ensuite, il pourra continuer sa progression vers la cible.

V00	V01	V02	D01	BLK
EPT	EPT	BLK	BLK	BLK
EPT	EPT	BLK	EPT	TGT
EPT	EPT	EPT	EPT	EPT
EPT	EPT	EPT	EPT	EPT

(a) Exemple en cul-de-sac, chemin

V00	D01	V02	V03	BLK
EPT	EPT	BLK	BLK	BLK
EPT	EPT	BLK	EPT	TGT
EPT	EPT	EPT	EPT	EPT
EPT	EPT	EPT	EPT	EPT

(b) Exemple en cul-de-sac, retour

V00	V01	V02	V03	BLK
EPT	V04	BLK	BLK	BLK
EPT	V05	BLK	V09	D01
EPT	V06	V07	V08	EPT
EPT	EPT	EPT	EPT	EPT

(c) Exemple en cul-de-sac, fin

En résumé, l'autopilote du drone fonctionne comme ceci :

Algorithm 1 autopilote algorithm

```
1: function EmptyCase(nextPoint, cases)
                                        ▷ visited if in array, empty is the boolean
2:     if not visited then :
3:         return -1
4:     if visited and empty then :
5:         return cluster
6:     if visited and not empty then :
7:         return MAX_INDEX

8: function smartMove(currentPos, directionOrder, cases)
                                        ▷ cases : informations we have about cases
9:     closestPoint ← MAX_INDEX
10:    for  dir in directionOrder do
11:        nextPoint ← getNextCase(currentPos, dir)
                                                ▷ get the case in direction dir
12:        index ← EmptyCasenextPoint, cases
13:        if index == -1 then :
14:            return nextPoint
15:        if index < closestPoint then :
16:            closestPoint ← index
17:    return cases[closestPoint].point
```

À la fin de l'algorithme, nous récupérons donc la case dont nous demandons le contenu à l'environnement.

Il est important de préciser qu'il y aura toujours 4 directions à tester, hors, il est possible que des directions ne soient pas possible à effectuer. Par exemple, à la création du drone celui-ci est en (0,0,0). Il ne peut effectuer que deux actions : SOUTH ou EAST. Dans ce cas, admettons que SOUTH soit plus avantageux que EAST, alors le drone aura pour directionOrder : SOUTH, EAST, SOUTH, SOUTH.

En revanche, on admet que le drone n'est pas enfermé à son démarrage, il pourra forcément effectuer une action.

Chapitre 5

Limites

5.1 Problèmes connus

Parmis les problèmes connus de mon simulateur on peut reconnaître une erreur récurrente au lancement de l'environnement qui effectue une `SegFault`. L'erreur est causée par un Point3i qui s'initialise mal. En effet, ces valeurs étant initialisées de manière aléatoire, il est possible que des fois l'aléatoire ne soit pas en ma faveur. Étant donné qu'il suffit de relancer l'exécutable pour que ça marche je n'ai pas cherché à résoudre ce problème.

Également, j'ai mis des pauses entre chaque envoi des messages de l'environnement. Il est donc possible qu'il mette un certain temps à s'actualiser. Il est important d'attendre que la question soit posée avant de répondre, sinon, la station sol n'arrive pas à finir son tour.

5.2 Conclusion

J'ai voulu faire le mutli-drone ayant fini l'autopilote mardi soir mais, après avoir fait un état de l'art de ce qu'il fallait implémenté, j'ai préféré écrire ce rapport et améliorer mon code afin de rendre quelque chose de propre et expliqué plutôt qu'une fonctionnalité en plus que je n'aurais sûrement pas pu finir d'implémenter.

Du point de vue du choix des langages je pense que le choix au départ était plutôt correct mais cela a eu des limites lorsque j'ai voulu créer un autopilote performant pour le drone. En effet, le langage C étant assez limité (pour les tableaux dynamiques entre autre), cela m'a beaucoup compliqué la tâche. Je pense à posteriori que j'aurais du coder la station sol en C car elle est maintenant plus simpliste que le drone.