

# INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE ROUEN NORMANDIE

## PROJET C++

---

Déplacement tridimensionnel de drones indépendants

---



Louis BAGOT  
Timothé BERNARD  
Morgan BUISSON  
Théau COUSIN  
Simon DELECOURT  
Quentin LOISEAU  
Margot SIRDEY

*À l'attention de : M. KOTOWICZ*

Mai 2018

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Présentation du produit . . . . .	2
1.2	Manuel d'utilisation . . . . .	2
<b>2</b>	<b>Spécifications</b>	<b>4</b>
2.1	Fonctionnalités à satisfaire et contraintes . . . . .	4
2.2	Besoins détaillés . . . . .	4
2.2.1	Description des fonctions à satisfaire . . . . .	4
2.2.2	Ordre de priorité des fonctionnalités . . . . .	5
2.2.3	Cas d'utilisation . . . . .	5
<b>3</b>	<b>Conception</b>	<b>7</b>
3.1	Conception préliminaire . . . . .	7
3.1.1	Scénarios et tests d'intégration . . . . .	7
3.2	Conception détaillée . . . . .	9
3.2.1	Diagramme de classes . . . . .	9
3.2.2	Détail du package Essaim . . . . .	9
3.2.3	Détail du package Environnement . . . . .	10
3.2.4	Détail du package Comportement . . . . .	11
<b>4</b>	<b>Implémentation et tests unitaires</b>	<b>12</b>
4.1	Implémentation . . . . .	12
4.1.1	Environnement . . . . .	12
4.1.2	Essaim : Capteurs . . . . .	12
4.1.3	Problèmes non implémentés ou non résolus . . . . .	12
4.2	Tests unitaires . . . . .	13
<b>5</b>	<b>Conclusion et perspectives</b>	<b>14</b>

# Chapitre 1

## Introduction

### 1.1 Présentation du produit

L'objectif est de réussir à faire déplacer un ensemble d'objets indépendants dans l'espace : des drones, avec comme but majeur d'éviter les collisions avec l'environnement et les autres drones.

Le produit se final se présente sous la forme d'un programme sur PC. Le programme permet de visualiser dans l'espace les drones, des colis et des obstacles. Il permet aussi de commander les drones, afin qu'ils réalisent les mouvements demandés. Ces commandes sont données par l'utilisateur par l'intermédiaire d'une interface basique en mode console. Pour se déplacer correctement, les drones disposent de plusieurs capteurs afin de détecter les obstacles autour de lui. L'utilisateur peut demander le retrait d'un colis. Notre projet requiert donc un affichage en 3D. Cet environnement physique sera défini à l'avance et modélisé dans le programme.

L'idée est d'implémenter différents algorithmes, afin de donner un comportement intelligent/autonome à nos drones. Il y aura plusieurs drones qui évolueront alors indépendamment les uns des autres.

### 1.2 Manuel d'utilisation

Il faut d'abord télécharger SDL1.2 ainsi que OpenGL pour pouvoir faire tourner les simulations. Sur Ubuntu ces bibliothèques sont présentes lors de l'installation de CodeBlocks, il faut alors ajouter dans les options de compilation les bibliothèques GLU et GL. Nous avons trouvé les informations nécessaires sur le site d'openclassrooms (<https://openclassrooms.com/courses/creez-des-programmes-en-3d-avec-opengl/introduction-a-opengl>). Pour l'installation sous MacOSX cela semble plus compliqué... Nous avons trouvé ce lien qui explique comment procéder pour l'installation de la SDL : <https://www.kth.se/social/files/56fd013ff276544e9cfdccd9/SDL-Setup-Mac%2810.11.1%29-Xcode%287.0.1%29.pdf>

Il faut aussi avoir CPPUNIT pour pouvoir faire tourner les tests. Nous utilisons CodeBlocks pour avoir un environnement de compilation généré automatiquement, ce qui est bienvenu lorsque l'on manipule tant de bibliothèques.

Une fois le projet lancé et fonctionnel, l'utilisateur dispose de plusieurs possibilités :

- Lancer les tests ou le code. Pour lancer les tests, l'utilisateur doit simplement changer la valeur de la variable booléenne de préprocesseur "TEST" dans le fichier main.cpp.
- Si les tests sont lancés, on peut simplement observer la fenêtre de CPPUNIT afficher que tous les tests ressortent 'OK'.
- Si la simulation est lancée, l'utilisateur pourra tourner la caméra intuitivement à la souris pour observer la simulation évoluer ; et zoomer/dézoomer à la molette de souris. Comme le temps n'a pas permis d'interface, il faudra toucher directement au code pour modifier le contexte de simulation. Les changements les plus intéressants sont :

- L'ajout d'Obstacles ou de Colis. Les obstacles sont les grands pavés droits visibles à l'affichage ; les colis sont visualisés via un point vert de retrait et un point violet de dépôt. Il suffit de copier les lignes déjà présentes au niveau du commentaire '//Test retrait colis" pour en créer un nouveau à de nouvelles positions.
- La modification de l'Environnement, directement dans le constructeur : il est possible d'accélérer la simulation en touchant au paramètre  $dt$  ; ou de jouer avec le paramètre de gravité.

Nous avons joint des vidéos avec ce rapport ; nous pouvons y voir deux cas :

- Un cas d'école de retrait de colis, où les drones les mieux placés vont bien chercher les colis et les déposent au bons endroits. Nous voyons les Capteurs des Drones devenir rouges lorsqu'ils détectent un Obstacle ; ce qui provoque une réaction chez le Drone dû au Comportement Naïf.
- un cas de visualisation de l'environnement physique où les Drones sont des objets inertes soumis à une vitesse initiale et la gravité, ce qui permet de visualiser les collisions.

## Chapitre 2

# Spécifications

### 2.1 Fonctionnalités à satisfaire et contraintes

Nous avons dans un premier temps prévu trois fonctionnalités pour notre projet.

- **Livraison de colis** : Plusieurs points de retrait et point de livraison sont identifiés par rapport à la demande. Le drone le plus proche du point de retrait prend l'initiative de retirer le colis et va le livrer au point de livraison. Ceci en évitant les obstacles et les autres drones sur sa trajectoire.
- **Spectacles** (type feu d'artifice – vu au J.O 2018) : L'objectif est que les drones se déplacent de manière synchrone suivant des instructions prédéfinies. Par exemple passage d'une formation cubique à une formation pyramidale.
- **Compétitions entre drones**(*Obsolète*) Le client veut pouvoir assister à des compétitions de drones. Le but de la compétition est de relier un point A à un point B en évitant les obstacles sur sa trajectoire et boucler le parcours le plus rapidement possible. Les compétitions peuvent se passer en solo (seul contre tous et le premier remporte la compétition) ou en équipe (les équipes s'affrontent entre elles). Dans l'environnement des objets qui, une fois récupérés, donnent des bonus ou des malus de temps.

Nous les présentons ici pour comprendre au mieux le déroulement de notre travail et pour aborder en détails des perspectives d'évolution de notre implémentation. La compétition entre drones était un objectif très ambitieux et nous l'avons très vite compris. Cependant, le retrait du colis et les formations de drones restaient les deux fonctionnalités à implémenter. Le rendu ne satisfait finalement que la première fonctionnalité.

#### Contraintes

Voici les principales contraintes que nous nous sommes fixés pour ce projet :

- Lorsqu'un drone entre en collision avec un obstacle il s'écrase au sol.
- Les collisions doivent être évitées un maximum.
- Les adresses de livraison et de retrait sont entrées par le client.

### 2.2 Besoins détaillés

#### 2.2.1 Description des fonctions à satisfaire

Voici les différentes étapes de chacune des fonctionnalités que nous avons prévu à l'origine.

##### Pour la livraison de colis :

- Identifier le drone le plus proche du point de retrait et lui donner l'ordre de s'y rendre,

- Identifier le plus court trajet entre le point de retrait et le point de livraison en évitant les obstacles dans l'environnement,
- Atterrir délicatement sur le lieu de livraison.

**Spectacle** : L'essaim de drone doit pouvoir réaliser les figures suivantes :

- Formation cubique,
- Formation pyramidale,
- Formation cercle.

**Compétition (Obsolète)** :

- Récupérer les bonus – éviter les malus,
- Se rendre sur la ligne d'arrivée le plus rapidement possible,
- Coopérer avec son équipe et bloquer l'équipe adverse.

### 2.2.2 Ordre de priorité des fonctionnalités

Comme notre projet s'annonçait ambitieux, et sous les conseils du professeur, nous avons défini un ordre de priorité des fonctionnalités à satisfaire. Le premier objectif a été respecté dans son intégralité et seul le premier point du deuxième objectif a vu le jour. Nous expliquons plus en détails par la suite les raisons pour lesquelles nous n'avons pas pu répondre à tous nos objectifs.

**Premier objectif** (à réaliser avec un seul drone)

- **Affichage** : Gérer correctement l'affichage des drones, l'environnement, positionner les sphères(=drones) et les obstacles au bon endroit. Gérer la caméra.
- **Environnement physique** : Il doit gérer la gravité, les accélérations et traiter les collisions à partir de toutes les positions l'environnement. Il doit aussi gérer la position au temps  $t+1$ .
- **Comportement d'un drone** : Un drone doit prendre l'initiative d'éviter une collision et doit gérer son accélération. Nous avons pour but d'implémenter plusieurs algorithmes de comportement pour les drones : un algorithme naïf (celui implémenté dans le rendu) qui consiste à prendre de l'altitude pour éviter chacun des obstacles, et un algorithme de recherche de chemin appelé  $A^*$ .
- **Implémentation de la méthode "AllerPoint"** : `AllerPoint(Point B)` est une méthode retournant un vecteur accélération qui sera interprété par l'environnement physique.
- **Gestion des capteurs**

**Deuxième objectif**

- Gérer la livraison d'un colis par un drone.
- Rendre le programme applicable à un ensemble de drones ; c'est à dire, gérer les formations, améliorer la gestion de l'essaim.

**Troisième objectif**

- Machine learning et obstacles dynamiques.

### 2.2.3 Cas d'utilisation

#### Diagramme des cas d'utilisation

Le diagramme des cas d'utilisation rassemble l'ensemble des fonctionnalités que nous avons prévu au départ.

Deux acteurs auront la possibilité d'agir sur notre Gestionnaire de Drones : l'utilisateur et le drone. Comme spécifié précédemment, l'utilisateur peut demander une formation de drones ou livrer/retirer un colis. Le drone, quant à lui, est capable d'aller à un point de destination donné. Notons que les demandes de l'utilisateur ne peuvent être satisfaites sans cette capacité du drone à se déplacer d'un point  $A$  à un point  $B$  (notion symbolisée par "include" dans le diagramme).

Les cas d'utilisation "Demander une formation" et "Livrer un colis" déclenchent des comportements semblables (déplacements de drones dans l'espace), mais avec des modalités différentes. C'est pourquoi ils sont regroupés au sein de même module, nommé Essaim. Il représente les comportements théoriques

(“les directives”) donnés au drone. Le module nommé “Comportement” correspond au déplacement physique/réel du drone. C’est pourquoi le cas d’utilisation “Aller au point de destination” est placé dans ce module.

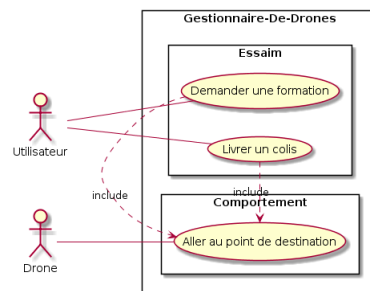


FIGURE 2.1 – Diagramme des cas d’utilisation

### Fonctionnalités par cas d’utilisation

Le premier cas d’utilisation : “demande de livraison de colis” se déroule selon les étapes suivantes :

- Choix du type de colis à transporter et du point de livraison par l’utilisateur.
- Affectation automatique du point de retrait en fonction du colis précédemment choisi.
- Le drone se rend au point de retrait et prend le colis.
- Le drone livre le colis, c’est à dire qu’il se déplace jusqu’au point de livraison en tenant compte des obstacles sur son chemin (et en empruntant le chemin le plus court).

Le deuxième cas d’utilisation : “demande de formation” se déroule selon les étapes suivantes :

- L’utilisateur choisit une forme et un point B qui en sera le centre.
- Les drones se rendent chacun à un point donné pour constituer la formation.

# Chapitre 3

## Conception

### 3.1 Conception préliminaire

#### 3.1.1 Scénarios et tests d'intégration

Les tests d'intégration consistent en la vérification du bon fonctionnement de l'interaction entre les différents modules de notre programme. Il s'agit donc en réalité de vérifier le bon déroulement des deux scénarios suivants.

##### Premier scénario envisagé

Ce scénario représente la fonctionnalité "livraison d'un colis". **Scénario 1** : Le client demande la livraison d'un colis. Il choisit alors le colis et un point B (point d'arrivée).

Par soucis de clarté nous avons pris la décision de séparer ce scénario en deux diagrammes. Le premier représente l'enchaînement des méthodes à partir de la demande de l'utilisateur. Le second représente une boucle réalisée par les drones. La boucle interne symbolise la lecture de la liste des objectifs de chacun des drones. Ils se déplaceront et la boucle sur l'environnement permettra alors de faire la mise à jour de la position de tous les drones.

Ce second diagramme se place à la suite de la méthode "ajouterObjectif" (visible dans le premier schéma).

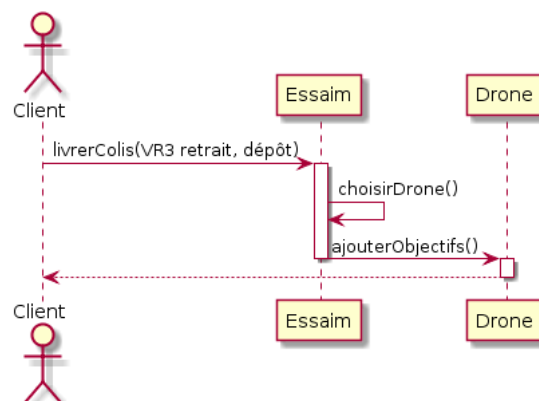


FIGURE 3.1 – Diagramme de séquence du scénario 1 (partie 1)



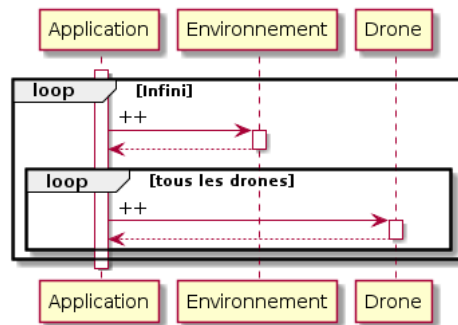


FIGURE 3.2 – Diagramme de séquence du scénario 1 (partie 2)

### Second scénario envisagé

Ce scénario représente la fonctionnalité "formation de drones". Puisque cette dernière faisait partie de nos objectifs, nous avons choisi d'en présenter le diagramme de séquence.

**Scénario 2 :** Le client demande une formation de drones. Il entre la forme souhaitée et le point B (point central de la formation).

Pour chacun des drones qui va être utile à la formation demandée nous lui donnons un nouveau point de destination (afin de créer correctement la formation) qui s'ajoute à sa liste d'objectifs. Après cette boucle, nous retrouvons à nouveau le diagramme de séquence permettant la mise à jour de l'environnement. (vu ci dessus figure 3.2).

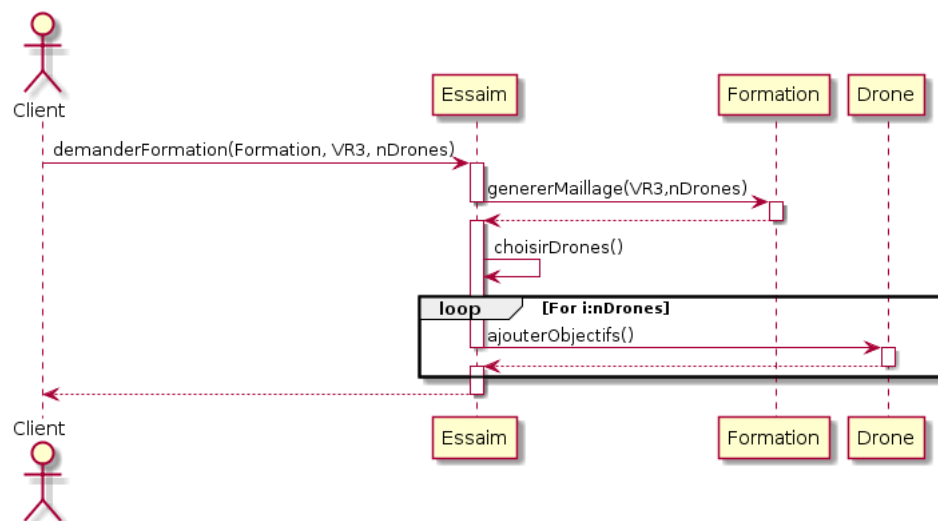


FIGURE 3.3 – Diagramme de séquence du scénario 2

## 3.2 Conception détaillée

### 3.2.1 Diagramme de classes

Voici le diagramme de classes de notre projet. Les associations avec la classe *VecteurR3* ne sont pas représentées sur le schéma par soucis de clarté.

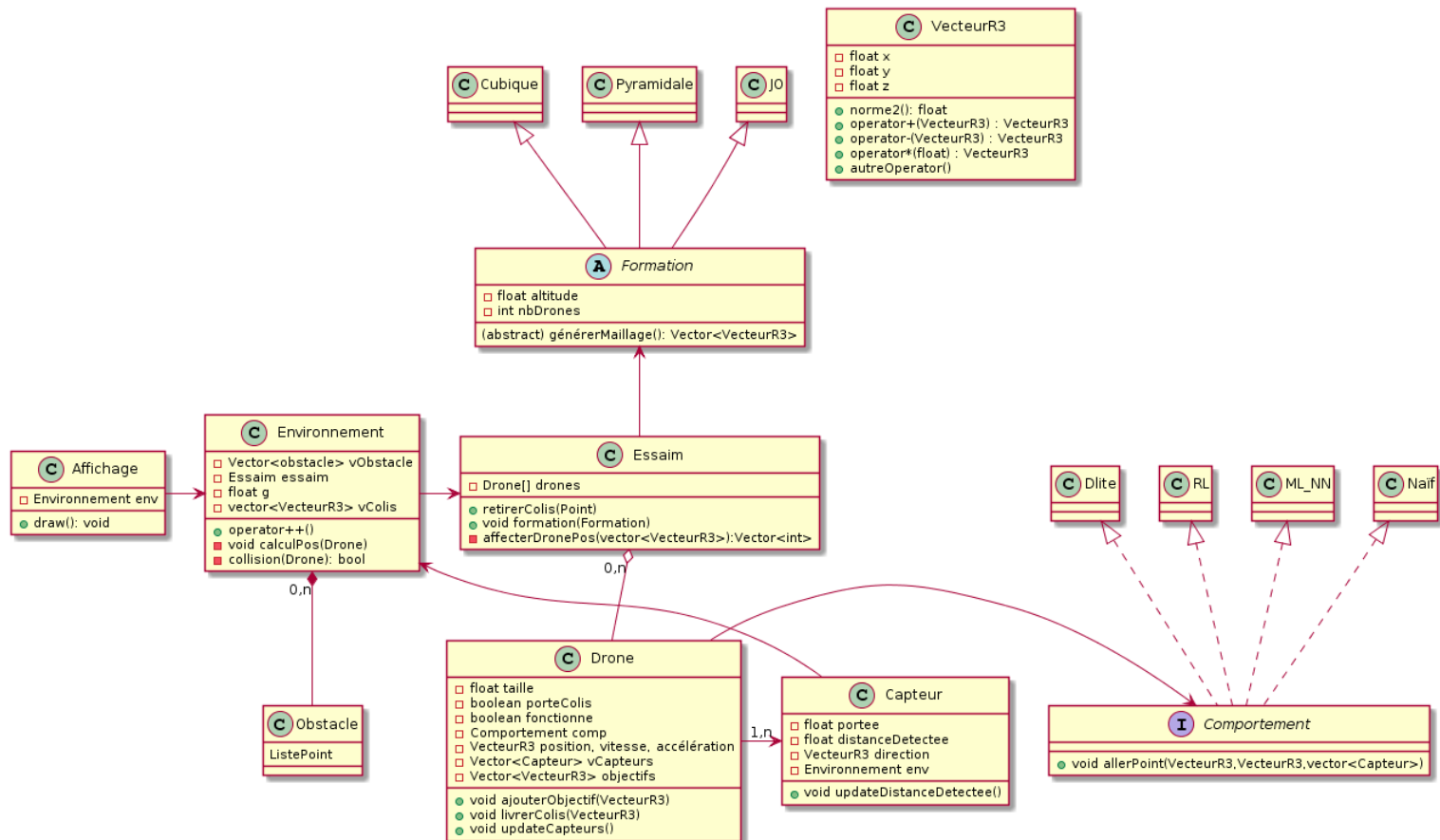


FIGURE 3.4 – Diagramme de classe

### 3.2.2 Détail du package Essaim

Le package essaim regroupe toutes les classes relatives à l'implémentation et à la représentation en Objet de nos drones. Logiquement, elle contient en son sein la classe Drone et Essaim. La classe Capteur est une classe qui sera uniquement utilisé "physiquement" par Drone. Elle permet à un drone de détecter les objets à une certaine distance dans une certaine direction. Cependant, dans notre diagramme de classe, il y a un lien entre Environnement et Capteur, car c'est l'environnement qui va envoyer les détails des objets aux capteurs.

La classe **Capteur** se décompose comme suit :

- Un *float* portée : c'est la distance maximale à laquelle peut "voir" un Capteur.
- Un *float* distanceDetectee : c'est la distance actuellement détectée.
- Un *Vecteur* de  $\mathbb{R}^3$  qui pointe dans la direction dans laquelle le Capteur regarde, relativement au Drone.
- Un pointeur sur l'environnement dans lequel évolue le Drone auquel est associé le Capteur ; qui lui permet de connaître le monde physique dans lequel il vit à chaque frame.

- Un Vecteur de  $\mathbb{R}^3$  qui est la position du Drone associé à ce Capteur ; ainsi qu'un *float* qui correspond la taille de l'engin.

La classe **Drone** comporte donc tous les paramètres utiles à sa description :

- Le rayon (*float*) car nous avons modélisé nos Drones comme des sphères, pour simplifier le modèle.
- 3 Vecteurs de  $\mathbb{R}^3$  pour définir : la position, la vitesse, et l'accélération. Ces éléments sont cruciaux pour la gestion physique de notre projet
- Un vecteur de  $\mathbb{R}^3$  pour la gravité : c'est le seul élément connu de l'Environnement ; cela permet au Drone de pouvoir contrer la gravité par défaut s'il n'a pas d'objectif. (l'hypothèse est que le constructeur peut entrer en brut la valeur de  $g$  pour aider le Drone)
- Une liste de Capteurs qui agiront comme les sens du Drone.
- Une queue d'objectifs à accomplir, c'est à dire les colis à aller chercher ou déposer
- Deux booléens : Un pour savoir si le drone est fonctionnel ou non (Le drone n'est plus fonctionnel si il touche un mur, un obstacle ou encore un autre drone)

Dans ce package, on retrouve dans l'Object Essaim, qui est un vecteur de Drone. Il comporte toutes les fonctions qui donneront les ordres et les déplacements à effectuer par les drones, comme par exemple aller chercher un colis, faire une formation, etc...

On a dans le package Essaim, la classe formation, qui est le détail des formations possibles par l'essaim de drones. Ainsi, on a à l'intérieur de cette classe, des fonctions qui généreront le maillage (le découpage de l'espace) et qui attribueront les positions à rejoindre pour faire la formation...

### 3.2.3 Détail du package Environnement

Dans le package Environnement, on va retrouver toutes les classes relatives à l'environnement que nous imaginions pour nos drones. Ainsi, nous avons la classe Environnement et la classe **Obstacle**. La classe des obstacles regroupe les objets inertes qui seront dans l'environnement. On considère que tous les obstacles sont des pavés droits, dont les arêtes suivent les vecteurs de la base orthonormée. Cette classe est donc en fait une simple liste de points représentant les sommets de l'obstacle. Elle ne comporte pas de méthodes particulières, uniquement des getter sur ses points et sur ses faces, ainsi qu'un constructeur qui génère les sommets du pavé en fonction des tailles de côté entrées.

Les getters sur les faces vont servir à la fois à l'affichage (OpenGL affiche face par face), et à la mise à jour des valeurs des Capteurs du Drone.

La classe contenant les éléments de l'environnement et leurs positions (i.e Essaim, Obstacles, colis) est donc **Environnement**. Elle gère la détection des collisions et le calcul de la position des drones. C'est le moteur physique du projet. Les Capteurs se servent de cette classe pour mettre à jour leur valeur. Ses attributs sont :

- Vecteur  $\mathbb{R}^3$  origineEnv et *float* cote, ce sont le point le plus négatif de notre cube d'Environnement, et la taille de son côté.
- Le *float* absorb est un général coefficient d'absorption des surfaces, pour les rebonds.
- L'environnement possède une liste vObstacles dispersés en son sein.
- On associe un Essaim de Drones à l'Environnement ; qui sera dans le code un pointeur sur Essaim, pour éviter certaines dépendances circulaires.
- Nous définissons une constante de delta temps, *float* dt, dans la formule  $f(t+dt) = f(t) + dt * f'(t)$  utilisé pour le calcul de la frame suivante
- Une liste de retraits vRetraits et de depots vDepots qui correspondent aux points d'un colis (associés à un même indice). Cela permet surtout à l'Affichage de les récupérer et donc à l'utilisateur de visualiser la situation.

L'environnement, comme dit plus haut, est le moteur physique de notre projet. Elle implémente donc plusieurs fonction incontournables, notamment celle qui gère les collisions entre les différents objets :

- la fonction qui calcule la position d'un drone en prenant en compte le vecteur accélération du drone et la gravité de l'environnement, en utilisant la formule vue plus haut.

- `void collisionsInterDrones()` vérifie de manière optimisée si les drones entrent en collision, et affecte les valeurs de vitesse correspondantes le cas échéant.
- `void collisionBords(Drone)` vérifie si un drone entre en collision avec bord de l'Environnement, et affecte directement la vitesse de l'objet le cas échéant.
- `void collisionObstacle(Drone)` vérifie si un drone entre en collision avec un obstacle, et affecte directement la vitesse de l'objet le cas échéant.
- l'opérateur `++` utilise toutes ces fonctions pour que la frame d'état suivante soit visuellement réaliste.

### 3.2.4 Détail du package Comportement

Nous définissons d'abord l'interface (implémentée comme une classe abstraite, sans attributs) `Comportement`. Celle-ci est composée d'une seule fonction qui résume ce que nous entendons par ce terme : c'est une fonction qui prend en entrée l'état du Drone (capteurs, position, vitesse) ainsi que l'objectif qu'il cherche à atteindre ; et qui ressort le vecteur accélération qui répond au problème.

Comme explicité précédemment, beaucoup d'approches sont possibles pour cette fonction. La seule que nous avons finalement pu prendre le temps d'implémenter est un algorithme **Naïf**. Ce dernier porte bien son nom : il va simplement se diriger vers l'objectif. S'il rencontre un obstacle, il va freiner et faire l'hypothèse que celui-ci n'a pas une hauteur infinie, et donc qu'il peut le survoler. Il va donc simplement monter, tant qu'il détecte des obstacles. Il est important de préciser qu'il ne s'intéresse qu'aux Capteurs qui vont dans la direction de sa destination. Il n'est donc absolument pas adaptable ! Un problème plutôt inattendu à résoudre fut le choix du vecteur accélération pour simplement aller d'un point A à un point B. Il est important de commencer à freiner (accélération négative) avant d'arriver à destination.

Pour le résoudre, nous avons décidé qu'une courbe appréciable pour la vitesse du drone serait une parabole, qui passe par 0 à la destination. Elle passe aussi par la vitesse du Drone au point de départ. Nous avons donc résolu un système portant sur une équation du second ordre, afin de trouver la bonne accélération à donner en fonction de la position du Drone. Nous n'avons étrangement pas réussi à faire fonctionner notre interpolation pour n'importe quelle vitesse initiale, ce qui donne parfois des imprécisions dans le déplacement ; mais le résultat est assez satisfaisant.

Une autre approche aurait été d'utiliser un algorithme de recherche de chemin comme `Dlite`, qui est une version améliorée de `A*` pour la découverte dynamique de nouveaux Obstacles. Cependant, bien que nous ayons obtenu une version générique fonctionnelle de `A*` en 3D, nous n'avons pas eu le temps de l'appliquer à notre problème, car cela posait de nombreux problèmes complexes, comme notamment un maillage intelligent de l'Environnement. De plus, nous retombions sur le problème du choix de l'accélération, une fois la direction de déplacement choisie.

La dernière approche, qui était l'idée qui a donné naissance au projet, était le `Machine Learning`, et en particulier dans ce cas le `Reinforced Learning`. Le principe aurait été d'apprendre aux Drones à répondre aux ordres, plutôt que de coder directement le comportement. Cette tâche se prêtait particulièrement bien au modèle des Capteurs.

## Chapitre 4

# Implémentation et tests unitaires

### 4.1 Implémentation

Dans cette partie nous allons discuter des problèmes d'implémentation rencontrés et des solutions trouvées durant ce projet.

#### 4.1.1 Environnement

Ce package a nécessité une approche particulièrement méticuleuse, pour un déroulement à la fois réaliste et optimisé du monde physique. En particulier, les collisions se sont avérées être très complexes. En effet, le problème d'intersection d'ensembles convexes n'est pas mince. Pour simplifier, nous avons déjà choisi de modéliser nos Drones par des sphères et nos Obstacles par des pavés droits; cependant il fallu ajouter à ces derniers la propriété qu'ils suivaient les axes du repère. Cela permit beaucoup de simplifications dans les calculs et un code, bien que lourd, assez simple.

un autre problème qui suivait fut de trouver le vecteur vitesse résultant d'un impact. Finalement, nous nous sommes souvenus de la matrice de Householder vue en ADG qui répondait presque parfaitement au problème (réflexion par rapport à un plan). Cela nous a fait gagner beaucoup de temps.

#### 4.1.2 Essaim : Capteurs

Dans le package Essaim, la fonction la plus complexe à implémenter fut sans conteste la fonction d'actualisation des valeurs des capteurs à chaque frame.

En effet nous retrouvons un problème de collision très complexe; et pour le résoudre nous avons à nouveau dû simplifier notre modèle. Nous avons décidé que les Capteurs suivraient, eux aussi les axes du repère. Un Drone aurait alors simplement 6 Capteurs selon  $(x, -x, y, -y, z, -z)$ . Encore une fois, cette simplification mène à un code assez lourd mais bien plus simple que pour le problème original.

il est important de noter que, bien qu'implémenté et fonctionnel, nous n'avons pas ajouté la détection des bords de l'environnement par les Capteurs. Cela devenait finalement trop délicat de trouver un point que les Drones pourraient atteindre sans détecter d'Obstacle; rendant les tests compliqués.

#### 4.1.3 Problèmes non implémentés ou non résolus

Nous avons parlé de  $A^*$  et des problèmes d'accélération; ceux-ci ont particulièrement freiné notre avancée car c'était une approche nouvelle pour nous et nous avons du mal à l'attaquer.

Parmi les éléments que nous n'avons pas pu implémenter, deux problèmes complexes auxquels nous avons

tout de même réfléchi sortent du lot. Ceux-ci n'en forment finalement qu'un : le problème de formation. Celui-ci se décomposait en deux parties :

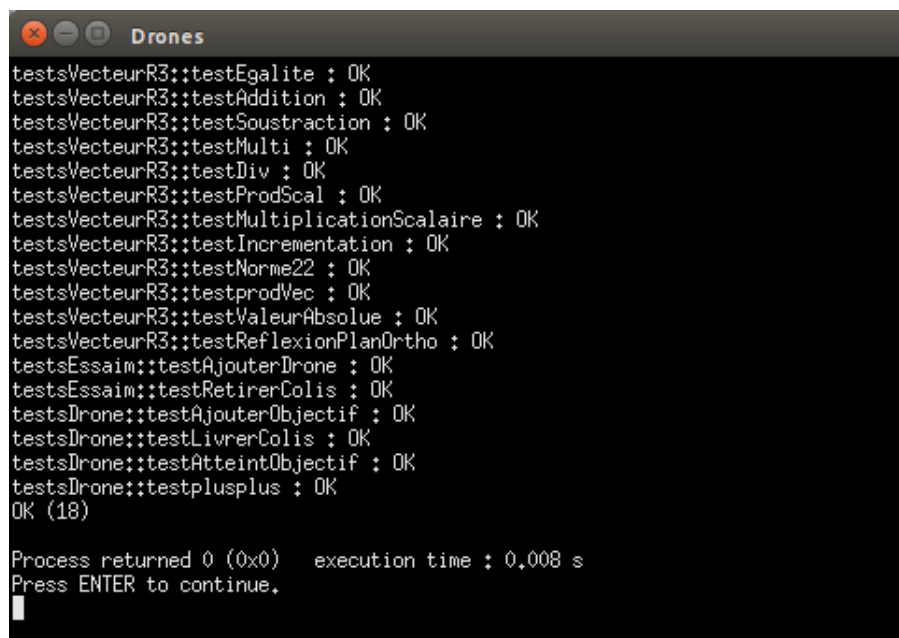
- le maillage d'un objet 3D selon des contraintes de position, de taille et de nombre de points
- l'affectation d'un ensemble drone (dans ce cas, seule la position du drone est intéressante) à ensemble un point. Nous avons décidé comme heuristique de minimiser la somme des distances parcourues, ce qui permettait de minimiser aussi les croisements de trajectoires. Nous avons vu un algorithme d'affectation en Théorie des Graphes qui répondait précisément à ce problème; nous aurions aimé l'implémenter pour l'appliquer et le voir en action dans un problème concret.

## 4.2 Tests unitaires

L'implémentation des tests unitaires s'est révélée bien pour lourde et consommatrice en temps que nous avions anticipé. En plus de problèmes de compatibilité et de gestion de github qui nous ont fait perdre beaucoup de temps, nous avons longtemps réfléchi à l'approche de certains tests qui étaient très complexes à tester dû à la complexité du problème qu'ils résolvaient. Ainsi, nous avons considéré que l'algorithme Naïf ainsi que l'Affichage seraient simplement testés par l'expérience plus que via CPPUNIT. Les tests sur le VecteurR3, bien que longs car nombreux, furent très simples à coder. Cependant, les tests sur l'Essaim et le Drone se révélèrent bien plus complexes car ils demandaient beaucoup de préparation, et surtout une compréhension du contexte de test à appliquer très poussée qui nous a demandé plus de réflexion que prévu.

Nous n'avons finalement pas pris le temps d'implémenter les tests sur l'Environnement; pour deux raisons. La première était que certains tests étaient particulièrement difficiles à mettre en oeuvre alors que la vérification pouvait se faire simplement grâce à l'affichage (notamment le calcul des positions ou des collisions, qui demandait une précision monstrueuse sur les calculs machine pour la vérification). La seconde raison a été évoquée plus haut, nous avons eu beaucoup de mal à faire fonctionner CPPUNIT sur les différents PC, et nous avons pris un peu trop de temps pour l'implémentation du reste du code.

Les tests effectués sur Essaim, Drone et VecteurR3 ont cependant bien aidé à notre compréhension de ce principe et étaient finalement fonctionnels :



```
testsVecteurR3::testEgalite : OK
testsVecteurR3::testAddition : OK
testsVecteurR3::testSoustraction : OK
testsVecteurR3::testMulti : OK
testsVecteurR3::testDiv : OK
testsVecteurR3::testProdScal : OK
testsVecteurR3::testMultiplicationScalaire : OK
testsVecteurR3::testIncrementation : OK
testsVecteurR3::testNorme22 : OK
testsVecteurR3::testprodVec : OK
testsVecteurR3::testValeurAbsolue : OK
testsVecteurR3::testReflexionPlanOrtho : OK
testsEssaim::testAjouterDrone : OK
testsEssaim::testRetirerColis : OK
testsDrone::testAjouterObjectif : OK
testsDrone::testLivrerColis : OK
testsDrone::testAtteintObjectif : OK
testsDrone::testplusplus : OK
OK (18)

Process returned 0 (0x0)   execution time : 0.008 s
Press ENTER to continue.
```

FIGURE 4.1 – Tests Unitaires

## Chapitre 5

# Conclusion et perspectives

Ce projet a été pour nous haut en nouveautés. D'une part, il nous a permis d'appliquer pour la première fois nos connaissances en C++, et qui plus est, sur un travail d'une durée relativement longue. D'autre part, nous sommes 7 personnes à avoir travaillé sur ce projet. Aucun d'entre nous n'avait eu l'occasion de participer à un travail en collaboration avec 7 personnes. Malgré cela, les échanges ont été actifs entre tous les membres du groupes. Nous sommes très heureux d'avoir été encouragés à modifier nos habitudes (travail en binôme, en trinôme...).

Un tel projet nous a appris à savoir se "recentrer" sur l'essentiel de nos objectifs. En effet, le début de notre projet était, avec du recul, beaucoup trop ambitieux. C'est pourquoi d'ailleurs nous n'avons pu remplir que partiellement nos objectifs à la vue du temps imparti.

La réalisation de ce projet a combiné informatique et mathématiques. Nous en avons appris d'avantage sur le langage C++ et avons pu appliquer des connaissances de mathématiques que nous n'aurions jamais pensé nécessaires pour un tel projet (pour obtenir les distances détectées par les drones par exemple).

En résumé, cette toute nouvelle expérience s'est avérée enrichissante tant d'un point de vue des échanges que d'un point de l'apprentissage technique.

Nous pouvons bien évidemment envisager des améliorations pour ce projet puisque la totalité des objectifs n'ont malheureusement pas été satisfaits. Ainsi, un algorithme de comportement optimisé et une gestion de formation sont les premières perspectives d'évolution de notre programme. A terme, la possibilité de compétitions inter-drones serait elle aussi envisageable et d'autant plus "amusante" pour l'utilisateur.

# Table des figures

2.1	Diagramme des cas d'utilisation . . . . .	6
3.1	Diagramme de séquence du scénario 1 (partie 1) . . . . .	7
3.2	Diagramme de séquence du scénario 1 (partie 2) . . . . .	8
3.3	Diagramme de séquence du scénario 2 . . . . .	8
3.4	Diagramme de classe . . . . .	9
4.1	Tests Unitaires . . . . .	13