

UTBM

Algorithmes de détections d'obstacles pour drones

TX52

Loïc Souverain
06/01/2019

Table des matières

| | | |
|----------|--|----|
| I | Introduction..... | 5 |
| I-1 | Origine militaire des drones | 5 |
| I-2 | Utilisation civile des drones..... | 5 |
| II | Etat de l'art..... | 8 |
| II-1 | Algorithmes | 8 |
| II-1-1 | Algorithmes de réactions | 8 |
| II-1-1-1 | Stop Avoidance..... | 8 |
| II-1-1-2 | Shift Avoidance..... | 9 |
| II-1-1-3 | Algorithme d'évitement de drone..... | 11 |
| II-1-2 | Génération de carte | 13 |
| II-1-2-1 | Virtual Force Field Avoidance | 14 |
| II-1-2-2 | Vector Field Histogram Avoidance | 16 |
| II-1-2-3 | Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles | 18 |
| II-1-3 | Récapitulatif | 19 |
| II-2 | Capteurs | 20 |
| II-2-1 | Capteurs unidirectionnels | 20 |
| II-2-1-1 | Capteurs Infrarouges..... | 20 |
| II-2-1-2 | Capteurs Ultrasons | 20 |
| II-2-1-3 | Référence de capteurs compatible Arduino..... | 20 |
| II-2-2 | Capteurs nuage de points..... | 20 |
| II-2-2-1 | LIDAR | 20 |
| II-2-2-2 | Stéréoscopie | 20 |
| II-2-2-3 | Caméra temps de vol..... | 21 |
| II-2-2-4 | Référence de capteurs compatible Arduino..... | 21 |
| II-2-3 | Conclusions sur les capteurs | 21 |
| III | Simulation..... | 23 |
| III-1 | Présentation d'Unity | 23 |
| III-2 | Modélisation du drone | 24 |
| III-2-1 | Remarques générales | 24 |
| III-2-2 | Déplacements..... | 25 |
| III-3 | Modélisation des capteurs | 26 |
| III-3-1 | Capteur mono-point..... | 27 |
| III-3-2 | Capteur mono-point (cône de détection) | 28 |
| III-3-3 | Lidar | 29 |

| | | |
|---------|--|----|
| III-4 | Algorithmes simulés | 30 |
| III-4-1 | Variantes de Shift Avoidance..... | 30 |
| III-4-2 | Implémentation de Virtual Force Field sans carte | 33 |
| III-5 | Autres simulations..... | 36 |
| IV | Bibliographie..... | 38 |

Table des Illustrations

| | |
|---|----|
| Figure 1 Prototype de drone torpille en 1918 (États Unis) | 5 |
| Figure 2 Drone MQ-1 Predator, en service de 1995 à 2018..... | 5 |
| Figure 3 Le Xiaomi mi drone 4K prend en charge l'enregistrement vidéo de résolution 3840 x 2160P à 30fps..... | 5 |
| Figure 4 Un pilote de drone avec ses lunettes de réalité virtuelle | 6 |
| Figure 5 Ligne de départ d'une course de drone | 6 |
| Figure 6 Drone agricole diffusant du pesticide | 6 |
| Figure 7 Schéma de fonctionnement d'un drone de surveillance | 6 |
| Figure 8 Un drone inspectant un mur | 7 |
| Figure 9 Le résultat de l'inspection | 7 |
| Figure 11 Schéma du comportement d'un drone avec algorithme Stop Avoidance | 9 |
| Figure 12 Schéma du comportement d'un drone avec algorithme Shift Avoidance | 10 |
| Figure 13 Shi-Tomashi | 12 |
| Figure 14 Lucas-Kanade..... | 12 |
| Figure 15 Détection de Blob | 12 |
| Figure 16 meanshift..... | 12 |
| Figure 17 camshift | 12 |
| Figure 18 Détection par les couleurs..... | 13 |
| Figure 19 Détection par la trajectoire | 13 |
| Figure 20 Classifieur en cascade..... | 13 |
| Figure 21 Virtual Force Field et un obstacle | 15 |
| Figure 22 Virtual Force Field et deux obstacles..... | 15 |
| Figure 23 Virtual Force Field dans un couloir étroit..... | 16 |
| Figure 24 Création de la grille d'histogramme dans le VFH | 17 |
| Figure 25 Création de l'histogramme polaire..... | 18 |
| Figure 26 Algorithme d'évitement d'obstacle générant une carte de son environnement | 19 |
| Figure 10 Comparaison entre un système à une caméra, et un système stéréoscopique | 21 |
| Figure 27 Capture d'écran Unity | 23 |
| Figure 28 Liste des composants d'un drone dans la hiérarchie | 24 |
| Figure 29 Visualisation du drone dans la scène | 24 |
| Figure 30 Le drone vu par l'inspecteur..... | 24 |
| Figure 31 La caméra fixé au drone | 24 |
| Figure 32 Lorsqu'un drone se déplace vers l'avant, il est incliné sur l'avant | 25 |
| Figure 33 Extraction du code correspondant au décollage..... | 26 |
| Figure 34 Visualisation d'un raycast..... | 26 |
| Figure 35 Représentation des raycast | 27 |
| Figure 36 Représentation du cône de détection..... | 28 |
| Figure 37 Illustration de la situation..... | 32 |
| Figure 38 Phase 0 : Décollage..... | 32 |
| Figure 39 Phase 1 : déplacement vers l'objectif..... | 32 |
| Figure 40 Phase 2 : Manoeuvre d'évitement | 32 |
| Figure 41 Phase 4 : Rotation..... | 32 |
| Figure 42 Retour à la phase 1 | 32 |
| Figure 43 Retour à la phase 2 | 32 |
| Figure 44 : retour à la phase 1..... | 32 |
| Figure 45 Phase 5 : atterrissage | 32 |

| | |
|--------------------------------------|----|
| Figure 46 Fin de la simulation | 32 |
|--------------------------------------|----|

I Introduction

Drone, qui signifie faux bourdon en anglais, désigne un véhicule aérien sans pilote ni humain à bord. En anglais, il est d'ailleurs souvent appelé « UAV » pour « Unmanned Aerial Vehicle ».

I-1 Origine militaire des drones

Le premier drone a été élaboré en 1916 au Royaume Unis, en pleine Première Guerre Mondiale. L'enjeu était de créer des torpilles aériennes télécommandées. Depuis, il a été utilisé dans de nombreuses missions militaires, tel que la surveillance, l'espionnage ou la distribution de flyers de propagande.



Figure 1 Prototype de drone torpille en 1918 (États Unis)



Figure 2 Drone MQ-1 Predator, en service de 1995 à 2018

I-2 Utilisation civile des drones

Aujourd'hui, les drones ont également de nombreux usages dans le civil. Le plus courant est la prise de vue. Souvent équipé d'une caméra, le drone est un moyen peu coûteux d'effectuer des prises de vues en hauteur de bonne qualité. Certains drones évolués ont également des fonctions supplémentaires, comme la possibilité de suivre un sujet mouvant, faire des photos en tournant autour d'un point fixe, ou bien d'effectuer un scan 3D du terrain via photogrammétrie.



Figure 3 Le Xiaomi mi drone 4K prend en charge l'enregistrement vidéo de résolution 3840 x 2160P à 30fps.

Il existe également des drones de courses dit FPV (« First Person View ») Racing. Le principe est de piloter un drone via des casques de réalité virtuelle type cardboard.



Figure 4 Un pilote de drone avec ses lunettes de réalité virtuelle



Figure 5 Ligne de départ d'une course de drone

On trouve également une utilisation professionnelle des drones. Dans l'agriculture, les drones sont utilisés pour surveiller les plantations. Ils sont capables de monitorer la croissance des plantes, et permettent également une distribution plus fine des pesticides et de l'eau.



Figure 6 Drone agricole diffusant du pesticide

On trouve également, comme dans le domaine militaire, des drones dédiés à la surveillance.

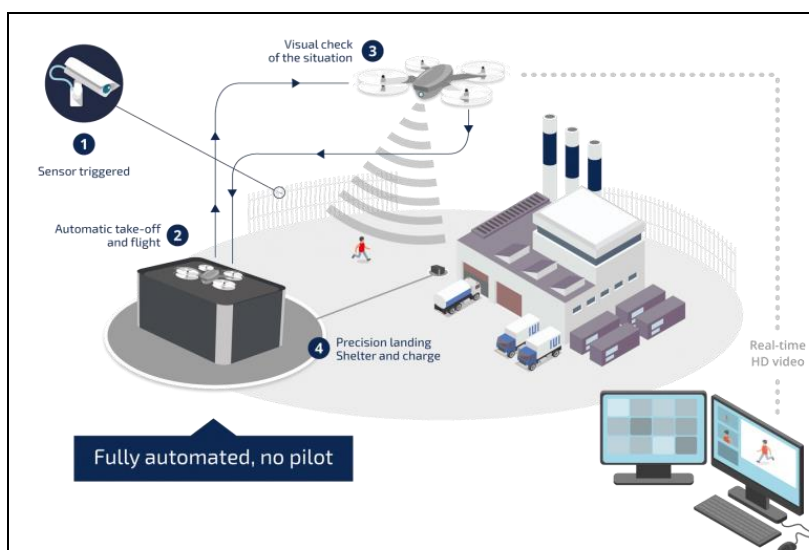


Figure 7 Schéma de fonctionnement d'un drone de surveillance

Dans ce schéma, on peut voir que si une caméra détecte un intru, on peut automatiquement envoyer un drone sur la zone suspecte pour vérifier la situation. Une fois la situation filmée et transmise, le drone revient à sa zone de décollage.

Enfin, il existe des drones d'inspections, capable d'inspecter des endroits difficiles d'accès. Il peut s'agir de ponts ou de bâtiments en hauteur.

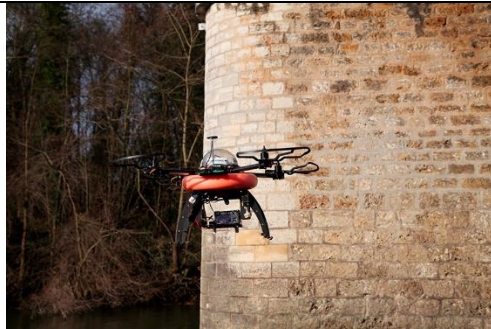


Figure 8 Un drone inspectant un mur

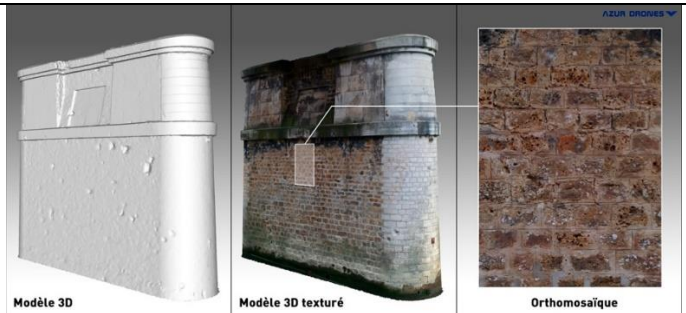


Figure 9 Le résultat de l'inspection

En France, il y avait 7510 pilotes de drones en Octobre 2018, gérant un parc de 13 300 appareils.

II Etat de l'art

Cet état de l'art est divisé en deux parties, la partie Algorithmes, et la partie Capteurs.

II-1 Algorithmes

D'après les travaux de Helen Oleynikova dans son article *Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles*^[7], On peut distinguer les algorithmes de détections de collisions en deux catégories. Les algorithmes de réactions, qui réagissent aux signaux reçus, et les algorithmes de génération de carte, qui sauvegarde les résultats des capteurs pour générer une carte. L'auteur mentionne également une troisième catégorie, les algorithmes d'exploration. Mais cette catégorie d'algorithmes n'a pas la même finalité que les deux précédents. Les algorithmes d'exploration ont pour but de minimiser les zones inconnues, et non pas pour but principal la détection de collisions. Pour cette raison, nous n'allons pas les étudier ici.

II-1-1 Algorithmes de réactions

Tout d'abord il y a les algorithmes de réactions. On donne au drone un but, et il essaye de s'y rendre en ligne droite. Si un capteur détecte un obstacle, une manœuvre d'évitement sera effectuée. Et c'est en cela qu'ils sont « de réactions » : ils ne font que réagir à un obstacle imminent. Ces algorithmes sont plus utilisés dans les zones dégagées et on trouve dans cette catégorie les algorithmes permettant l'esquive d'objets mobiles. On trouve aussi la majorité des algorithmes exploitant les capteurs infrarouges et ultraviolets.

II-1-1-1 Stop Avoidance

Cet algorithme est proposé dans la bibliothèque *Collision Avoidance Library* développé par Intel^[9].

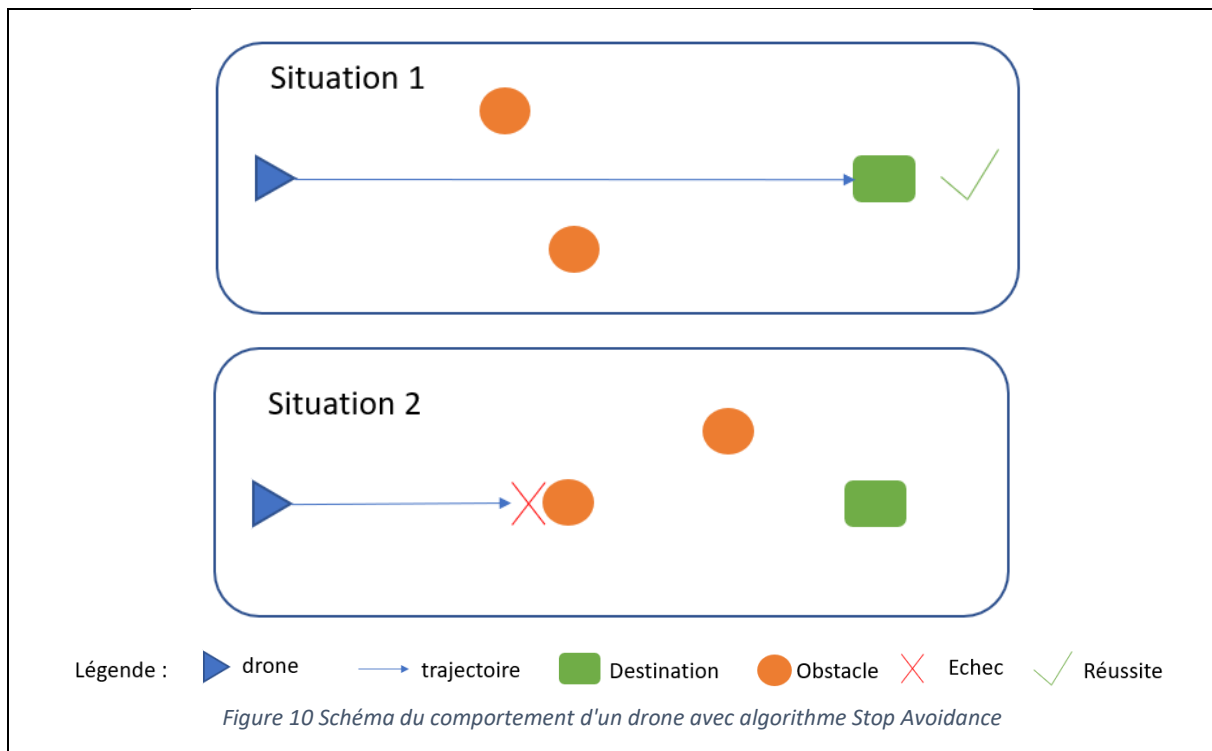
Il s'agit de l'algorithme d'évitement d'obstacle le plus simple. Lorsqu'un obstacle est détecté, il est considéré comme dangereux, le drone se stoppe, puis se pose. L'obstacle est évité.

Voici le pseudo code de cet algorithme :

```
// On exclu ici les phases d'atterrissage et de décollage, elle ne sont pas spécifique au problème
Tant que vrai : // boucle principale du programme
    Si !(obstacle_dangereux()):
        Avancer vers la cible
    Sinon :
        Arrêter d'avancer
        Atterrissage()
        Fin du programme
```

Obstacle_dangereux est une fonction qui renvoie vrai lorsqu'il y a un obstacle « dangereux ». Cette fonction est propre au capteur, au drone, et au cas d'utilisation. Dans le cas d'un capteur mono point (voir partie Capteur), cela correspond à un obstacle dont la distance au drone serait inférieure à un seuil.

Voici une vue schématique du comportement du drone (vue de dessus) :



Dans la première situation, il n'y a pas d'obstacle dangereux, le drone continue donc sa route. Dans la seconde situation, le drone rencontre un obstacle dangereux, et est donc contraint de s'arrêter. Il s'agit du défaut de cet algorithme : il y a de forte chance que le drone n'arrive pas à destination. Cependant, il évite bien la collision.

II-1-1-2 Shift Avoidance

Cet algorithme vient également de la bibliothèque *Collision Avoidance Library* d'Intel^[9].

Contrairement au précédent algorithme, celui-ci propose une stratégie d'évitement d'obstacle. Si un obstacle (dangereux) est détecté par le drone, celui-ci va se déporter sur la droite, en effectuant une rotation vers la droite, avec comme centre la destination du drone. Une fois la rotation effectuée, il ré-évaluera l'obstacle. S'il a disparu, le drone continuera sa course.

Voici le pseudo code :

```

etat = 1
delta = 0
// on exclu ici les phases d'atterrissage et de décollage, elle ne sont pas spécifique au problème

Tant que vrai:
    Si etat == 1: // état normal
        Si !obstacle():
            avancer vers la cible
        Sinon:
            etat = 2
    Si etat == 2: // manoeuvre d'évitement
        Rotation vers la droite avec pour centre la cible
        Si obstacle:
            delta = 0
            si delta < SEUIL: // il faut rajouter une marge, sinon il y aura contact entre
l'obstacle et les hélices

```

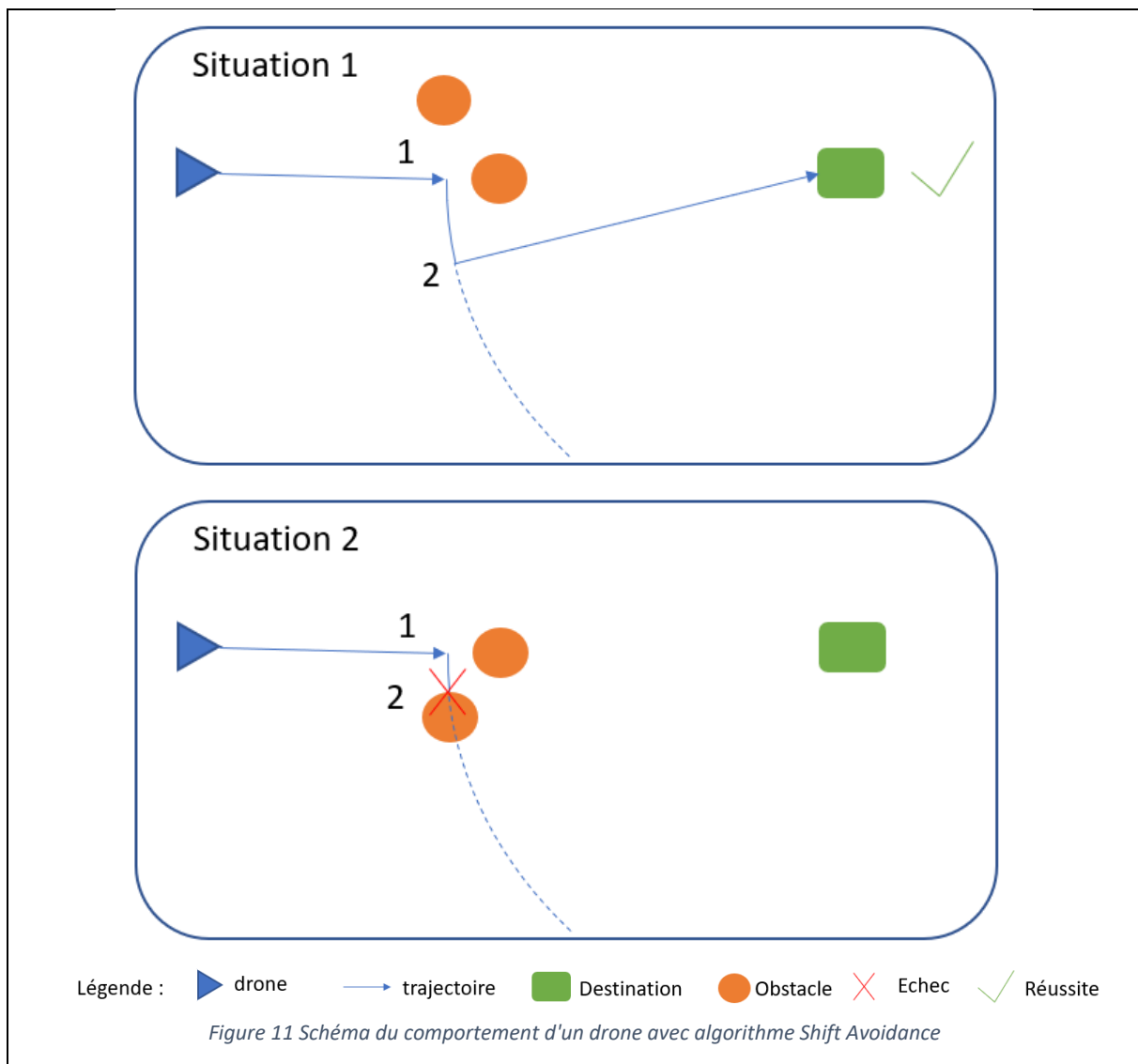
```

delta += déplacement
sinon:
    etat = 1

```

Obstacle() est une fonction qui détecte un obstacle dangereux. « Etat » est un entier qui indique l'état du drone. L'état 1 correspond à l'état normal : avancer vers la cible, et l'état 2 à la manœuvre d'évitement. On pourrait imaginer un état 0 pour le décollage, et un état 3 pour l'atterrissage. Delta est un flottant permettant de mesurer la distance parcouru depuis le dernier obstacle détecté. Lorsque Delta dépasse la constante SEUIL, le drone repasse à l'état 1.

Voici une vue schématique du comportement du drone (vue de dessus) :



Dans la situation 1, le drone va en ligne droite en direction de son objectif, jusqu'à détecter, au point 1 un obstacle. Il va alors effectuer une rotation autour de son objectif, jusqu'au point 2, puis, comme il n'y a plus d'obstacle, continuer en ligne droite vers sa destination.

Dans la situation 2, le drone va également en ligne droite jusqu'au point 1, puis détecte l'obstacle, et entame sa rotation. Le drone ne parvient pas à détecter le second obstacle, et il y a collision.

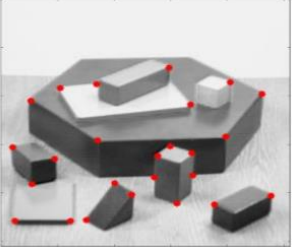
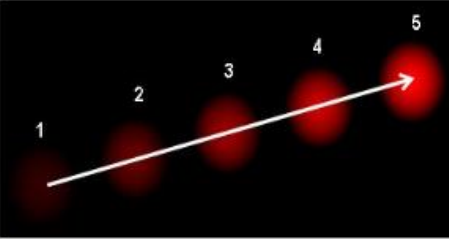
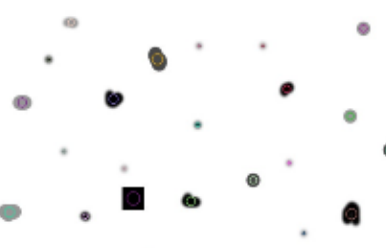
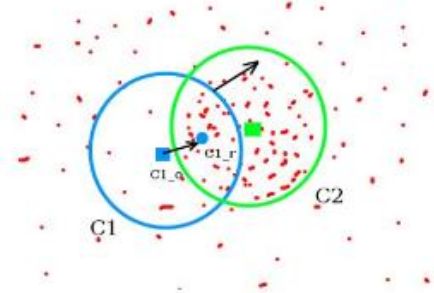

II-1-1-3 Algorithme d'évitement de drone

Cet algorithme, élaboré dans l'université de Berkley par Hani Sedaghat-Pisheh dans l'article *Collision Avoidance Algorithms For Unmanned Aerial Vehicles Using Computer Vision*^[8], permet à un drone d'éviter un autre drone. Il est basé sur la vision par ordinateur, par conséquent, il a été conçu pour fonctionner avec de la stéréoscopie. Il peut également être utilisé par n'importe quel système permettant d'obtenir une image rgb.

Le problème est séparé en deux parties. La première partie est la détection de drone via le système de vision, puis le second est de suivre leur trajectoire.

II-1-1-3-1 Suivi d'objet

Pour la détection de drone, 5 algorithmes ont été évalués : Shi-Tomasi, Lucas-Kanade, la détection de Blob, Meanshift, et Camshift.

| | |
|---|---|
|  <p>Figure 12 Shi-Tomashi</p> | <p>Une détection de caractéristique est utilisée pour trouver les coins les plus importants de l'image. Cette méthode requiert trop de calcul pour être embarquée dans un drone.</p> |
|  <p>Figure 13 Lucas-Kanade</p> | <p>Lucas-Kanade utilise le flot optique pour traquer les objets sur l'image courante, ainsi que les successives. La figure montre le suivi d'une balle à travers 5 frames successives. Cette méthode requiert également trop de calcul.</p> |
|  <p>Figure 14 Détection de Blob</p> | <p>Un blob est un groupe connexe de pixels qui partage des caractéristiques communes. Cette méthode n'est pas précise pour trouver des objets spécifiques, elle est plus adaptée à la détection de formes géométriques au contours clairs, comme sur la figure</p> |
|  <p>Figure 15 meanshift</p> | <p>Meanshift se base sur la couleur pour traquer les objets. Grâce à un histogramme des couleurs et à la densité des gradients, l'algorithme trouve le maximum de densité de pixels dans une image. La figure montre comment l'algorithme passe de la zone de recherche C1(peu dense) à C2 (très dense). Cette méthode, bien qu'excellente, est en difficulté lorsqu'un objet se rapproche très fortement de la caméra. Cette méthode a donc été écartée.</p> |
|  <p>Figure 16 camshift</p> | <p>Camshift veut dire « Continuously Adaptive Meanshift », Il s'agit donc d'une amélioration de Meanshift, qui change la taille de la fenêtre de recherche à chaque itération. Cela lui permet de pouvoir traquer un objet qui se rapproche de la caméra, et qui grossi. C'est donc la méthode qui a été choisie.</p> |

II-1-1-3-2 Détection de drone

Afin de pas devoir suivre tous les points mouvants, il est nécessaire de détecter le drone avant de pouvoir utiliser meanshift. Là encore, plusieurs alternatives ont été essayées : La détection de couleur, la détection de mouvement, et le classifieur en cascade.

| | |
|---|---|
|  <p><i>Figure 17 Détection par les couleurs</i></p> | <p>Avec une connaissance à priori des couleurs du drone, il est possible de segmenter l'image afin de retrouver la position du drone dans l'image. Dans la figure, les images binaires de la segmentation sont visibles. Cette méthode n'est pas assez robuste. La couleur des drones ne peut pas être connue à l'avance.</p> |
|  <p><i>Figure 18 Détection par la trajectoire</i></p> | <p>Si on fait la différence entre deux frames, il est possible de repérer les objets mouvants, et donc les drones. Cette méthode n'est pas utilisable sur un drone, puisque tout son environnement est en mouvement.</p> |
|  <p><i>Figure 19 Classifieur en cascade</i></p> | <p>Le classifieur en cascade de Haar est classique pour identifier des objets spécifiques. Une base de données de 2000 images d'apprentissage a été utilisée. C'est cette méthode qui a été choisie pour détecter le drone.</p> |

II-1-1-3-3 Récapitulation

Pour récapituler, la solution retenue pour détecter des objets en mouvements (ici des drones), a été la suivante :

- 1) Utilisation d'un classifieur en cascade pour reconnaître des drones à esquiver sur l'image.
- 2) Suivre la trajectoire du drone grâce à camshift
- 3) Lorsque la trajectoire devient dangereuse (utilisation stéréoscopie pour la distance), procéder à une manœuvre d'évitement.

II-1-2 Génération de carte

Il y a ensuite les algorithmes qui construisent une carte de leur environnement au fur et à mesure de leur parcours. La construction de telles cartes, bien que coûteuse en ressources, permet

au drone de trouver son chemin en cas de situation complexe, où il y aurait un risque que les précédents algorithmes laissent le drone coincé dans un minimum local.

II-1-2-1 *Virtual Force Field Avoidance*

Ce dernier algorithme proposé dans la librairie Intel repose sur le concept que la cible applique « une force attractive » sur le drone, et qu'au contraire, les obstacles appliquent « une force répulsive » sur le drone, qui va dépendre de la distance entre le drone et l'obstacle. La direction du drone sera déterminée par la somme de ces vecteurs.

Cette méthode nécessite donc que le drone mémorise la position des obstacles détectés, et donc construise une carte des obstacles.

Voici le pseudo code :

```

Coef = ... // coefficient de force
seuil = ... // Si un vecteur obstacle est colinéaire au vecteur destination, seuil est utilisé pour
              modifier le vecteur obstacle.
Tant que Vrai :

    Si Obstacle :
        carte.ajout(Obstacle) // mise à jour de la carte
        vecteur_destination = destination - position // application de la force de la destination
        Si vecteur_destination < epsilon :
            Atterrissage
            fin
        vecteur_somme = vecteur_destination
        Pour tout carte.obstacles() :
            vecteur_obstacle = position - obstacle
            Si sont_colineaires(vecteur_obstacle, vecteur_destination):
                vecteur_obstacle += seuil*vecteur_orthogonal_normé(vecteur_obstacle)
            vecteur_somme += coef*vecteur_obstacle/(norme(vecteur_obstacle)**2)

    deplacement(vecteur_somme)

```

Coef est un flottant à déterminer en fonction du gabarit du drone et de l'application. Ce coefficient est analogue à «G» dans la formule de la gravité.

La façon dont est détecté l'obstacle n'est pas détaillé ici, puisque dépendante du capteur.

Carte correspond à la carte des obstacles détecté par le drone, elle est mise à jour à chaque instant.

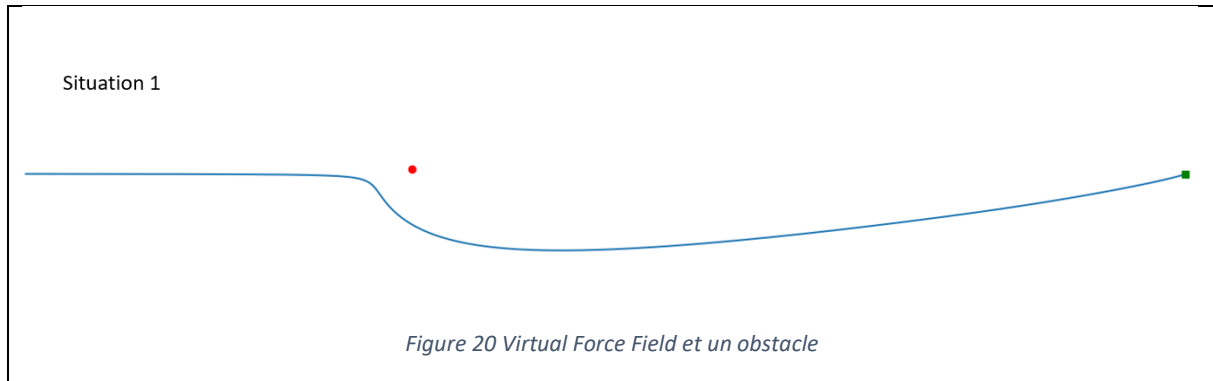
Si le vecteur destination est trop petit, cela signifie que le drone est arrivé, et donc l'atterrissage est déclenché.

Ensuite, on stocke dans vecteur_somme le vecteur correspondant à la force attractive de la destination, ajouté à tout les vecteurs_obstacle (force répulsive que les obstacles applique sur le drone). Il faut faire attention, si un vecteur obstacle est colinéaire au vecteur destination, il y a un risque d'annulation du vecteur somme, et donc d'arrêt du drone. Pour cette raison, un tel vecteur somme est légèrement dévié.

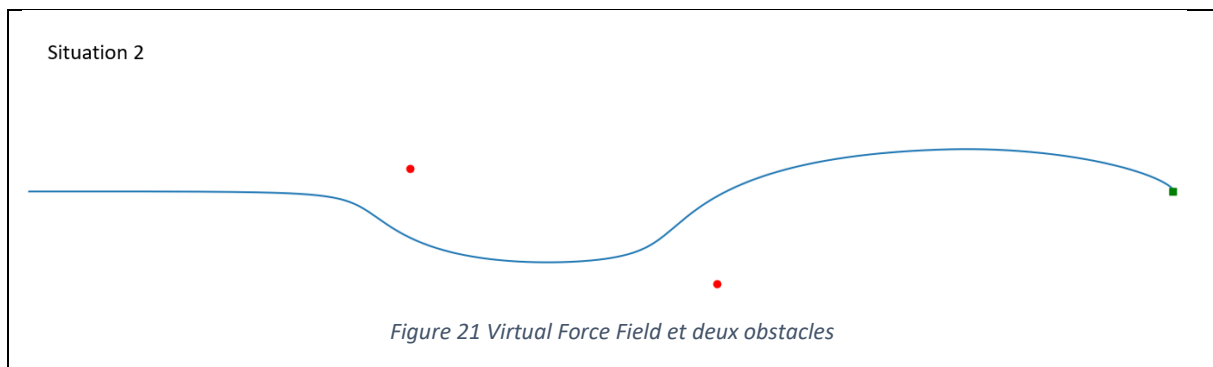
Enfin, le drone se déplace dans la direction du vecteur somme.

Voici plusieurs vues schématiques du comportement du drone avec un tel algorithme (effectué en python). Pour chaque situation, le drone commence à gauche, et doit se rendre à son objectif, le point vert à droite. Les obstacles sont en rouges.

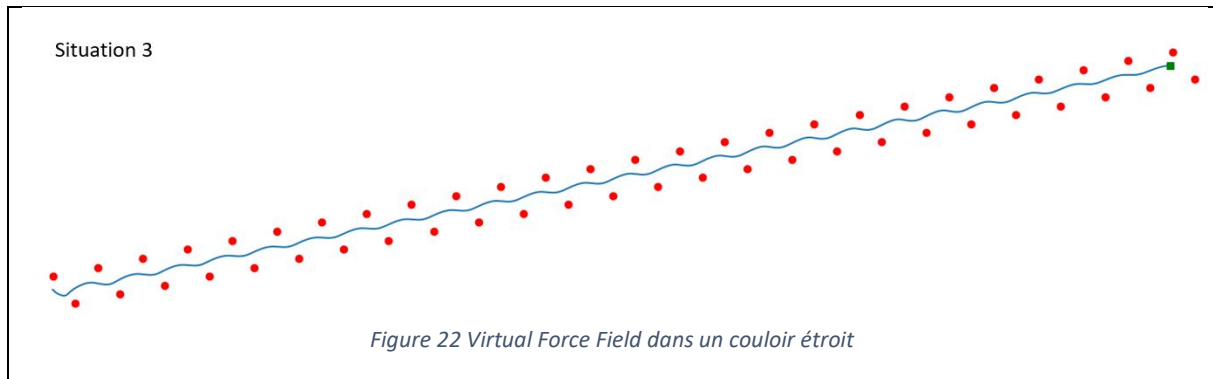
Dans toutes ces situations, la liste des obstacles est connue du drone à l'avance.



Dans cette première situation, un obstacle est placé sur la trajectoire du drone. Au début, le vecteur de l'obstacle est trop faible par rapport au vecteur objectif, la trajectoire du drone est donc rectiligne. Lorsque le drone s'approche de l'obstacle le vecteur obstacle, dont la norme est inversement proportionnelle au carré de la distance, devient très vite important. Cela se traduit par un brusque changement de direction. Une fois l'obstacle évité, le drone se dirige vers son objectif.



On peut voir ici que le drone a un comportement en plusieurs phase. Au début, les forces exercées par les deux obstacles sont négligeable. Le premier obstacle étant moins sur la trajectoire du drone que dans la première situation, le changement de direction est moins brutal. Le dernier changement de direction intervient lorsque le drone arrive à proximité du deuxième obstacle. On remarque ici le caractère non optimal de la trajectoire : entre le deuxième obstacle et l'arrivée, le drone effectue une trajectoire courbe, alors qu'une trajectoire en ligne droite aurait été plus efficace. Cela peut s'expliquer par le fait que le plus le drone se rapproche de son objectif, moins le vecteur est élevé, jusqu'à être quasiment nul à l'arrivée.



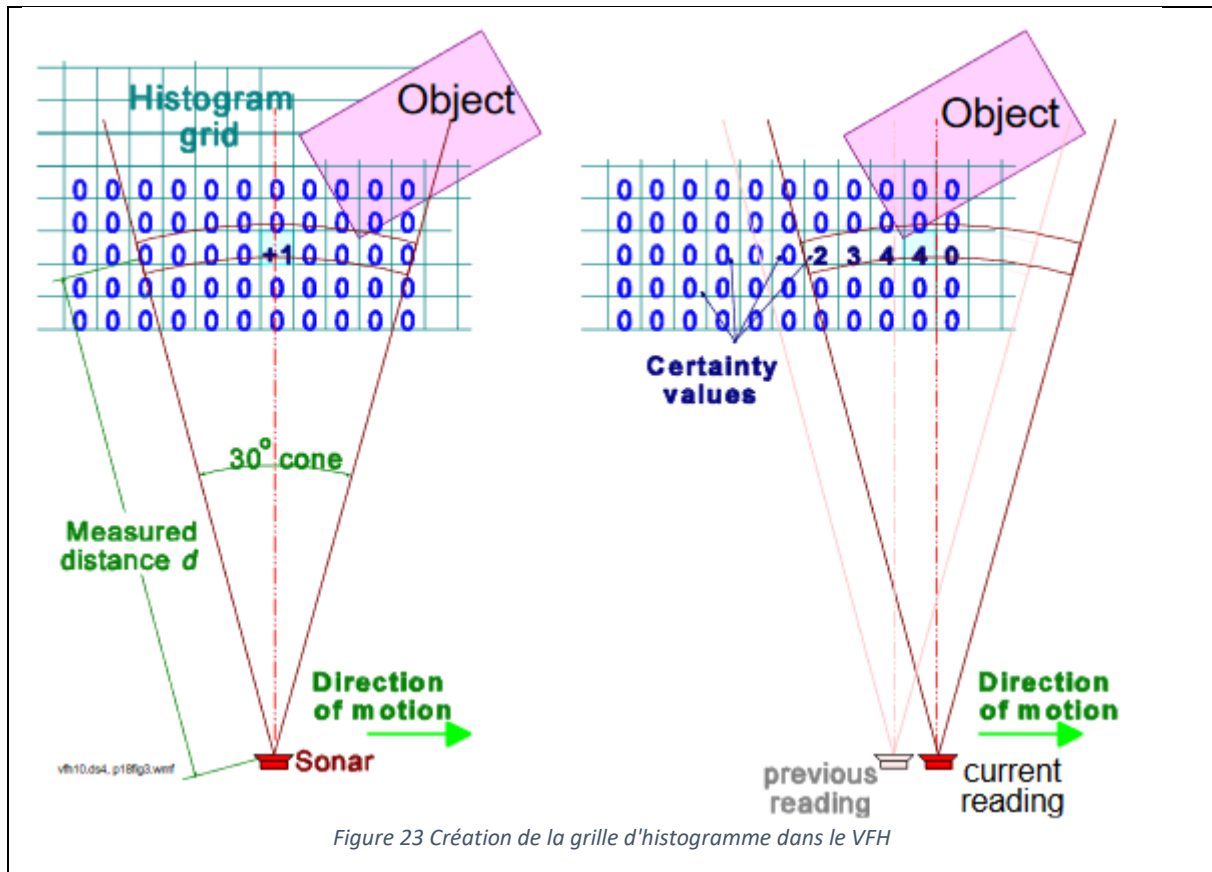
Dans cette dernière situation, nous remarquons le caractère oscillatoire de Virtual Force Field dans un couloir : à la place de faire le trajet en ligne droite, l'algorithme va s'éloigner d'un côté du mur afin de réduire sa distance à celui-ci, mais cela va avoir pour effet de le rapprocher de l'autre mur, et donc, il va s'en éloigner à nouveau, etc.

II-1-2-2 *Vector Field Histogram Avoidance*

Créé comme une amélioration au Virtual Force Field, le Vector Field Histogram (VFH), a d'abord été créé pour les robots roulants. Cet algorithme est détaillé dans l'article THE VECTOR FIELD HISTOGRAM - FAST OBSTACLE AVOIDANCE FOR MOBILE ROBOTS, écrit par J. Borenstein et Y. Koren^[10]. Il contient 3 étapes :

II-1-2-2-1 *La grille d'histogramme*

La création d'une grille d'histogramme en coordonnée cartésienne, centré sur le robot. Chaque case de cette grille contient la probabilité que cette case contienne un obstacle. Cette grille est mise à jour en temps réelle.

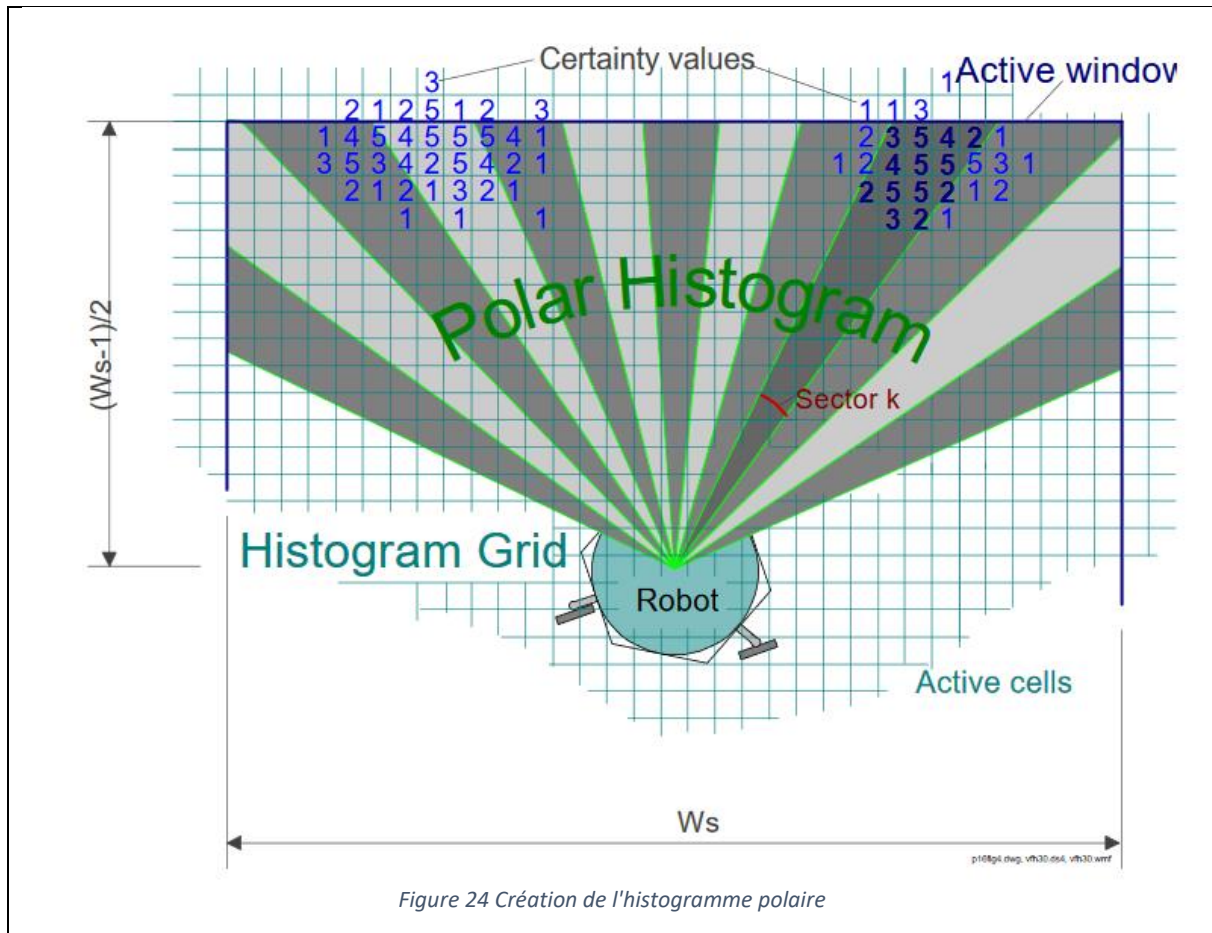


On peut voir à travers ce schéma comment cette grille est remplie dans le cas d'un capteur mono-point. Lorsqu'un obstacle est détecté par un tel capteur, on sait qu'il se trouve dans un cône de détection, à une certaine distance, mais on ne connaît pas sa position exacte. Cette grille permet donc de déterminer avec plus de précision où est l'obstacle.

Lorsqu'un obstacle est détecté par le capteur, seulement une case est incrémentée. Dans ce cas, il s'agit de la case située au centre du cône de détection, à la distance d . Après plusieurs itérations (image 2), on obtient une probabilité de distribution d'histogramme (« histogrammic probability distribution »).

II-1-2-2-2 Création de l'histogramme en coordonnée polaire

Une fois La grille d'histogramme créée, on la convertie en un histogramme a une dimension en coordonnée polaire :



Le schéma ci-dessus illustre la création de l'histogramme polaire. Chaque section contient la somme des cases correspondante.

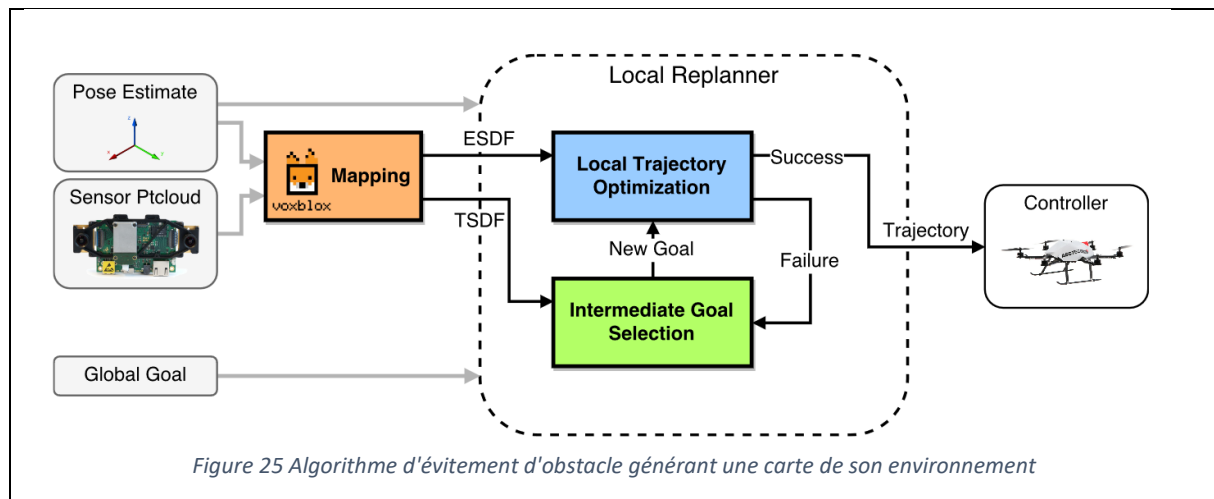
II-1-2-2-3 Choix de la vallée

Enfin, avec l'histogramme obtenu, on identifie une liste de vallée, qui correspondent à une absence d'obstacle. La vallée la plus proche de la direction de l'objectif sera choisie.

II-1-2-3 Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles

Nous allons ici décrire l'algorithme détaillé par les travaux de Helen Oleynikova dans l'article *Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles*^[7]. Son but est de permettre à un drone d'aller à son objectif dans un environnement dense en obstacle (type forêt), en évitant tout obstacle. Cet algorithme nécessite un capteur nuage de points.

Voici le schéma qui récapitule l'algorithme utilisé :



Nous allons décrire les différents blocs présentés sur ce schéma. Le premier, « Pose Estimate » est l'estimation de la position du drone. Cette estimation peut se faire de deux façons différentes : soit par un asservissement en position du drone, soit à l'aide d'un GPS. Ces deux solutions peuvent être combinées. Le bloc suivant, Sensor PtCloud, correspond à un capteur nuage de point, comme décrit dans la partie capteurs. Enfin, Le « global Goal » est la destination du drone.

A l'aide de l'estimation de la position et du capteur nuage de point, on peut créer une carte de voxel grâce à la librairie « Voxblox ». Une fois cette carte faite, on peut entrer dans la partie « Local Replanner » de l'algorithme. Le but de ce bloc est de calculer une trajectoire, qui sera envoyé au Controller du drone.

La première étape du local replanner, est de trouver une trajectoire grâce au « Local Trajectory Optimization ». Il prend en entrée le ESDF (Euclidean Signed Distance Field), qui lui sert à calculer les coûts de collisions. S'il ne parvient pas à générer une trajectoire, on va alors chercher un but intermédiaire, à l'aide du TSDF (Truncated Signed Distance Fields), qui va permettre de calculer le gain de l'exploration. Le but de ce bloc est donc de trouver un but intermédiaire intéressant à explorer, pour plus tard être en mesure d'accéder au but global. Le Local trajectory Optimization va alors essayer d'établir une trajectoire vers ce nouveau but. Quand il réussit, la trajectoire est envoyée au Controller du drone.

II-1-3 Récapitulatif

Voici un tableau récapitulatif des différents types d'algorithmes étudié :

| Type d'algorithmes | Réaction | Génération de carte | Exploration |
|--------------------|---|---|---|
| But | Aller d'un endroit à un autre, en réagissant aux signaux des capteurs | Aller d'un endroit à un autre, en construisant une carte de l'environnement | Minimiser l'espace non cartographié sur une zone spécifique |
| Avantage | Facile à mettre en place, possibilité de détecter les objets mobiles | Trouve toujours un chemin | |
| Inconvénient | Limité dans des situations denses en obstacles (labyrinthes, forêts) | Nécessite une plus grande capacité de calcul | Pas adapté à la solution |

II-2 Capteurs

II-2-1 Capteurs unidirectionnels

Les capteurs unidirectionnels sont pointés dans une direction, si un objet entre dans leur cône de détection, ils sont capables de déterminer la distance de cet objet.


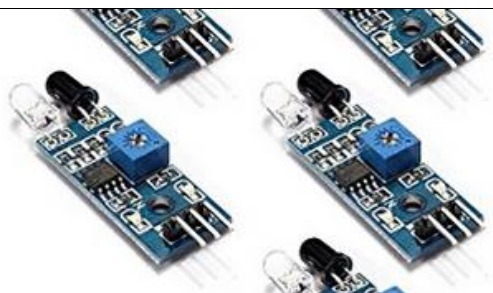
II-2-1-1 Capteurs Infrarouges

Les capteurs infrarouges sont des capteurs unidirectionnels utilisant les infrarouges. Un émetteur envoie un signal infrarouge, si le récepteur reçoit ce même signal après un certain temps, cela signifie que le signal a été réfléchi. En mesurant l'intervalle de temps entre l'émission du signal et sa réception, on peut déterminer la distance de l'obstacle. Si aucun signal n'est capté après un certain temps d'attente, on considère qu'il n'y a pas d'obstacle.

II-2-1-2 Capteurs Ultrasons

Les capteurs ultrasons utilise le même principe que les capteurs infrarouges, mais avec des ultrasons.

II-2-1-3 Référence de capteurs compatible Arduino

| Type de capteur | Ultrasons | Infrarouge |
|-----------------|--|---|
| Photo |  |  |
| Amazon | lien | lien |
| Prix | 9€99 (5pc) | 10€29 (10pc) |
| Portée | 2cm -> 3m | 2cm -> 30cm |
| angle | Moins de 15° | 35° |

II-2-2 Capteurs nuage de points

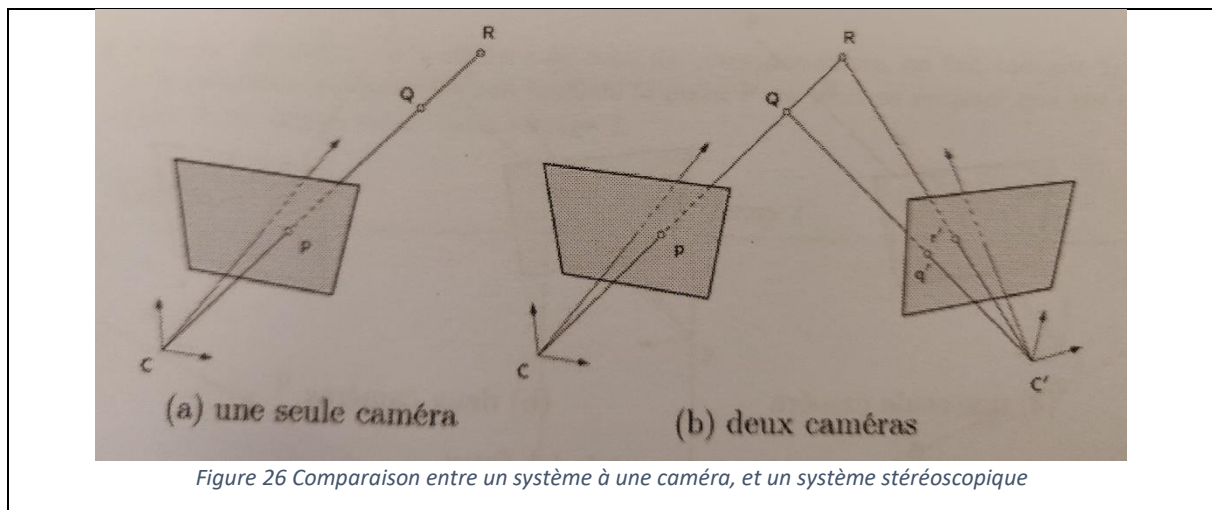
L'autre catégorie de capteurs composée de lidars, de la stéréoscopie et de cameras temps de vol, permet de récupérer des informations de distances avec une plus grande résolution, parfois reliée à une image couleur (image RGBD « Red Green Blue Distance »).

II-2-2-1 LIDAR

La technologie LIDAR, pour « Light Detection And Ranging », ou « Laser Imaging Detection And Ranging », se base sur l'analyse des propriétés d'un faisceau de lumière renvoyé à son émetteur. Il a le même fonctionnement qu'un radar, mais utilise un laser à la place des ondes radios.

II-2-2-2 Stéréoscopie

Un système de caméra stéréoscopique est composé de deux caméras, en imitant le système de vision humain avec deux yeux. Après une calibration, il est possible de calculer la distance des points visible par les deux caméras.


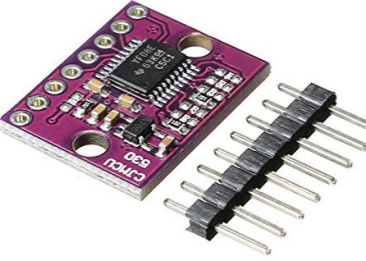



On peut voir ici qu'avec une seule caméra, on perd l'information de profondeur. La deuxième caméra permet de retrouver cette dimension.

II-2-2-3 Caméra temps de vol

Enfin, les caméras temps de vol fonctionnent sur le même principe que les capteurs infrarouges, mais l'information de temps entre l'émission et la réception du signal est effectuée pour chaque pixel de la caméra.

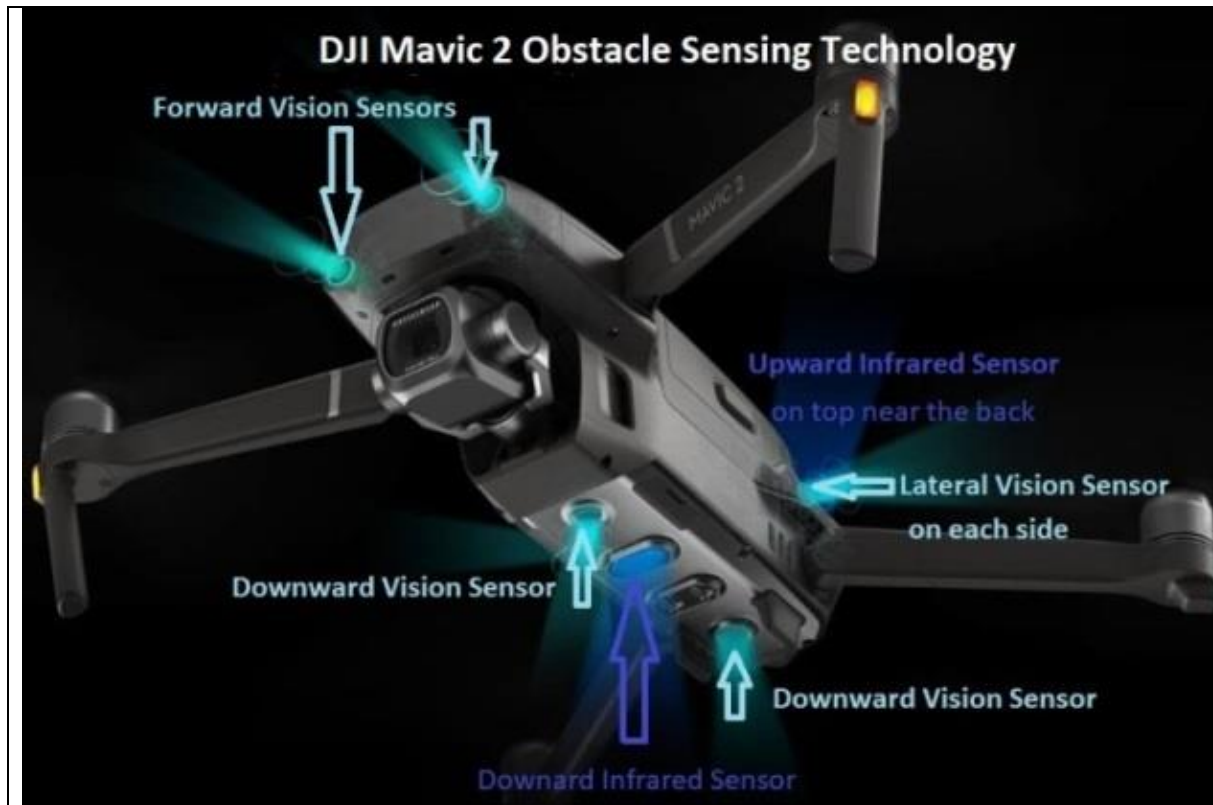
II-2-2-4 Référence de capteurs compatible Arduino

| Type de capteur | Lidar | Capteur temps de vol | Stéréoscopie |
|-----------------|---|--|---|
| Photo |  |  |  |
| Amazon | lien | lien | camera adaptateur |
| Prix | 39€10 | 20€ | 2*25€ + 50€ => 100€ |
| Portée | 30cm->12m (7m en extérieur) | Jusqu'à deux mètres | |
| Résolution | MONOPOINT | MONOPOINT | Chaque camera : 8Mpx |

II-2-3 Conclusions sur les capteurs

On remarque que les capteurs multipoints ne sont pas disponibles pour Arduino, puisqu'ils nécessitent une puissance de calcul et de stockage supérieur à la capacité de la puce utilisée pour le drone disponible. Des capteurs de ce type existent, mais seulement en mono-point (avec un cône de détection bien plus petit). Ils sont également plus onéreux que les capteurs d'ultrasons et d'infrarouges. La stéréoscopie semble être possible, mais c'est l'option la plus chère.

Les drones du commerce disposent en général de plusieurs capteurs différents. On peut notamment citer le drone DJI Mavic 2, qui est connu pour avoir un nombre important de capteurs, et une détection poussée des obstacles dans toutes les directions.



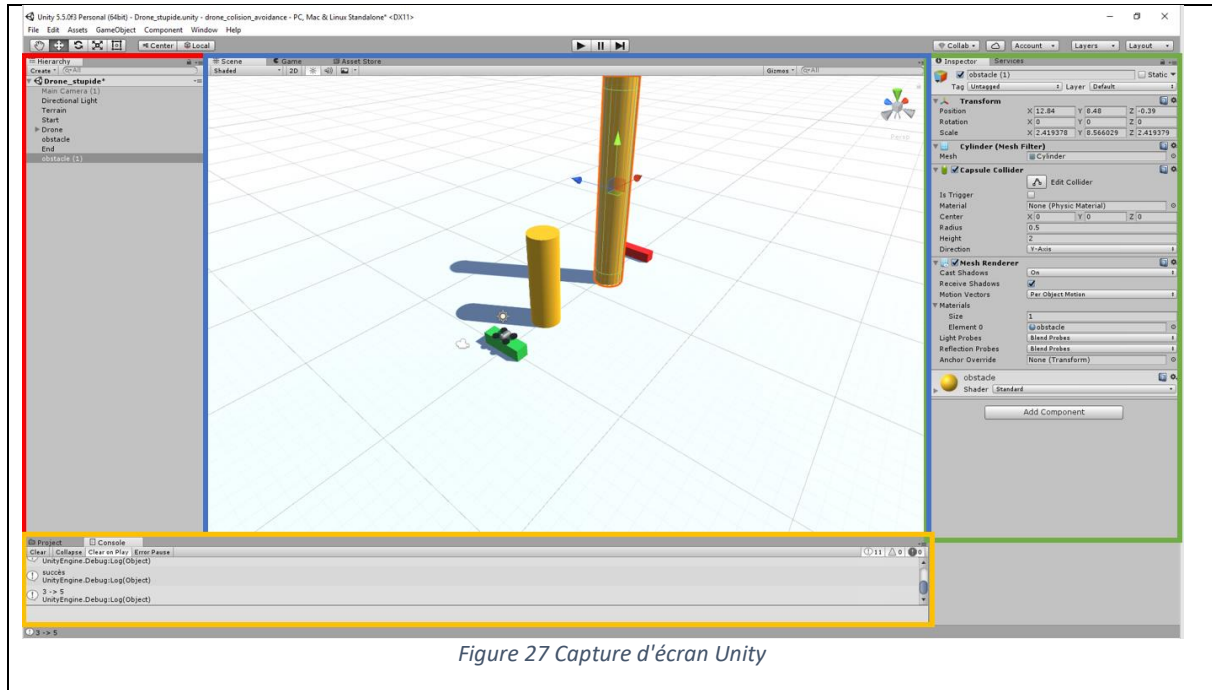
On peut voir ici que ce drone se sert de stéréovision à l'avant et sous le drone, ainsi que des capteurs infrarouges au-dessus et sous le drone, et un capteur de vision sur les flancs du drone.

III Simulation

Ne disposant pas de drone, j'ai utilisé des simulations pour tester des algorithmes. J'ai pour cela utilisé le moteur de jeu Unity 3D, et coder le comportement des drones en C#.

Le code est disponible ici : https://github.com/supermouette/drone_collision_avoidance/

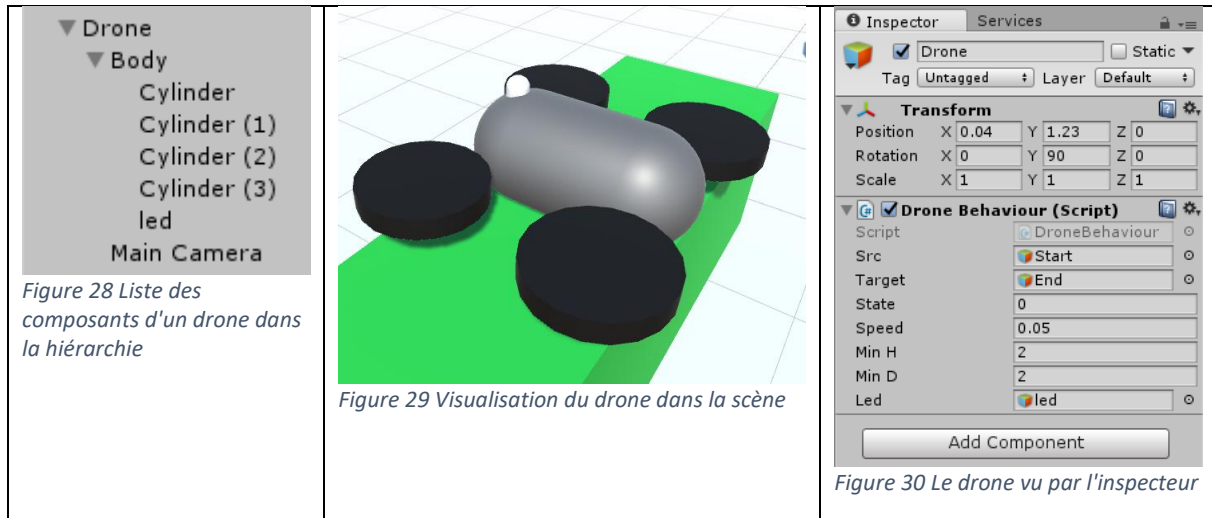
III-1 Présentation d'Unity



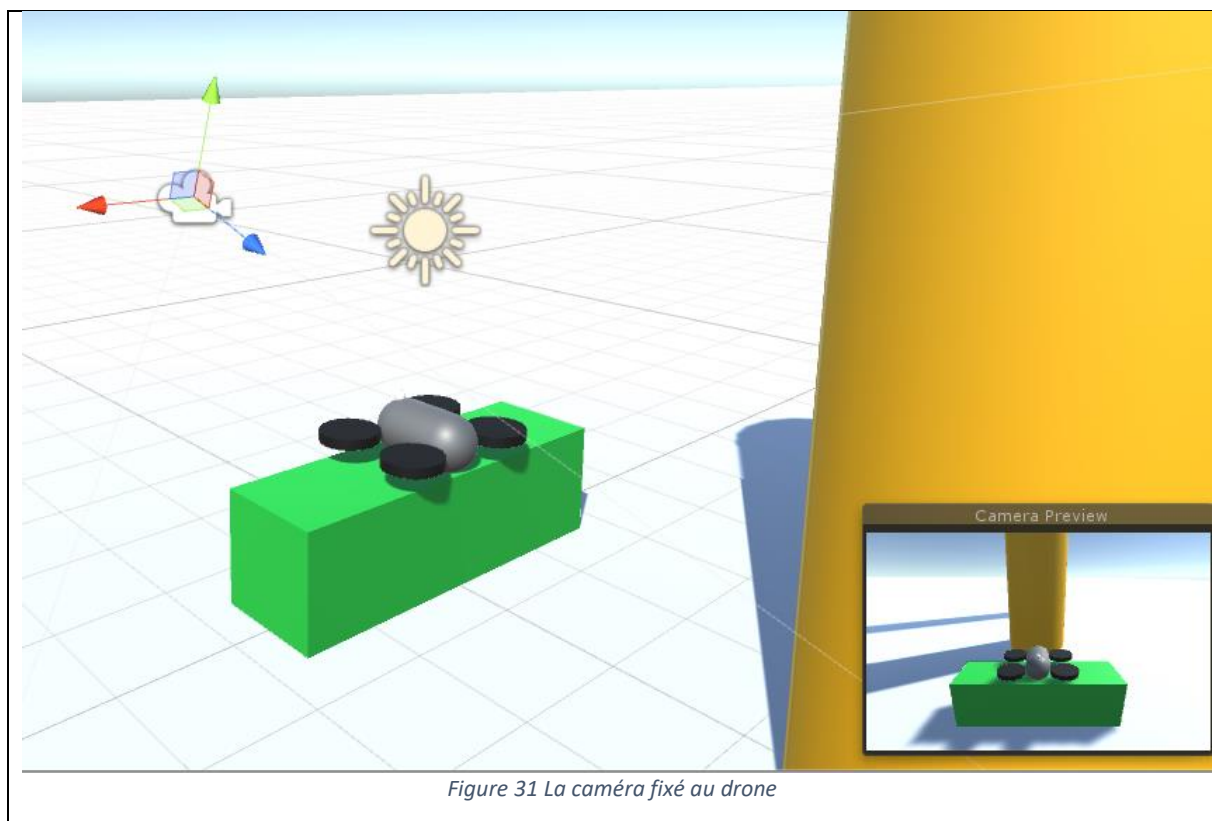
Unity se présente ainsi : La fenêtre principale, encadrée en bleu, correspond à la scène, où on vient placer les différents éléments qui composent la simulation. La liste de ces éléments est présente dans la hiérarchie, en rouge. En vert se trouve l'inspecteur qui regroupe toutes les informations de l'objet sélectionné dans la scène et la hiérarchie. On peut par exemple modifier à la main ses coordonnées, ou bien paramétrer ces attributs. Pour le drone, c'est par exemple via l'inspecteur qu'on pourra changer sa vitesse maximum. Enfin, la console est en jaune. Elle est utilisée pour le debugage. Par exemple, on l'utilise pour voir les différents changements d'état du drone.

III-2 Modélisation du drone

III-2-1 Remarques générales



Bien que le modèle 3D du drone est simpliste, il suffit à se représenter un drone, et surtout son gabarit. Le corps du drone est représenté par un objet de type capsule, et ses ailes par des cylindres. Une autre capsule plus petite se situe à l'arrière du drone. Elle est utilisée comme une LED, qui change de couleur en fonction de l'état du drone. Elle permet également de distinguer l'arrière de l'avant du drone. Enfin, une caméra est attachée derrière le drone, pour pouvoir suivre son évolution lors de la simulation



L'inspecteur permet de paramétrer le drone. On peut définir sa vitesse maximum, lui indiquer son point de départ (Src), ainsi que son objectif (Target). Il faut également indiquer quel objet correspond à la LED, sa hauteur après le décollage (minH). MinD peut être considéré comme spécifique à un algorithme, cela correspond à la distance minimale que le drone peut avoir avec un obstacle.

III-2-2 Déplacements

Les déplacements du drone sont codés comme de simples translations/rotations. Une rotation sera codée comme une rotation, et un déplacement vers l'avant a été traduit par une translation vers l'avant. Ainsi, la simulation diffère d'une situation réelle où, lorsqu'un drone avance vite, il se penche vers l'avant. En l'état actuelle, la simulation correspond donc à un cas où le drone avance à faible vitesse.



Figure 32 Lorsqu'un drone se déplace vers l'avant, il est incliné sur l'avant

```

51 // Update is called once per frame
52 void Update () {
53     if (state == 0) { // décollage
54         if (src.transform.position.y + minH > this.transform.position.y)
55         {
56             this.transform.Translate(this.transform.up.normalized * speed);
57         }
58         else
59         {
60             setState(1);
61         }
62     }
63 }

```

Figure 33 Extraction du code correspondant au décollage

Voici un exemple de déplacement : la phase de décollage. Le code se décompose ainsi : lors du décollage (phase 0), on applique une translation vers le haut sur le drone à la vitesse constante speed, spécifié par l'utilisateur dans l'inspecteur. On stoppe cette translation lorsque le drone dépasse la hauteur minimum minH, puis on passe à la phase suivante.

III-3 Modélisation des capteurs

La modélisation des capteurs se basent sur les raycasts (lancer de rayons). Ils n'ont pas de représentation physique (comme un boîtier visible). On peut tout de fois les représenter, comme sur la figure ci-dessous.

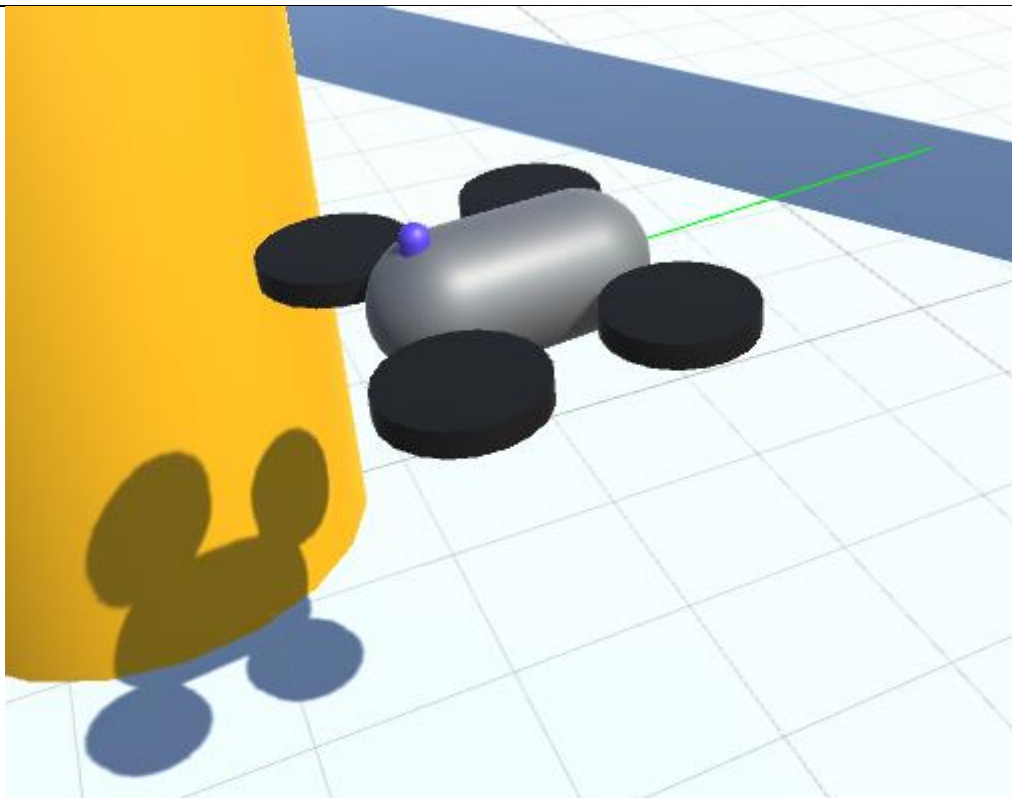


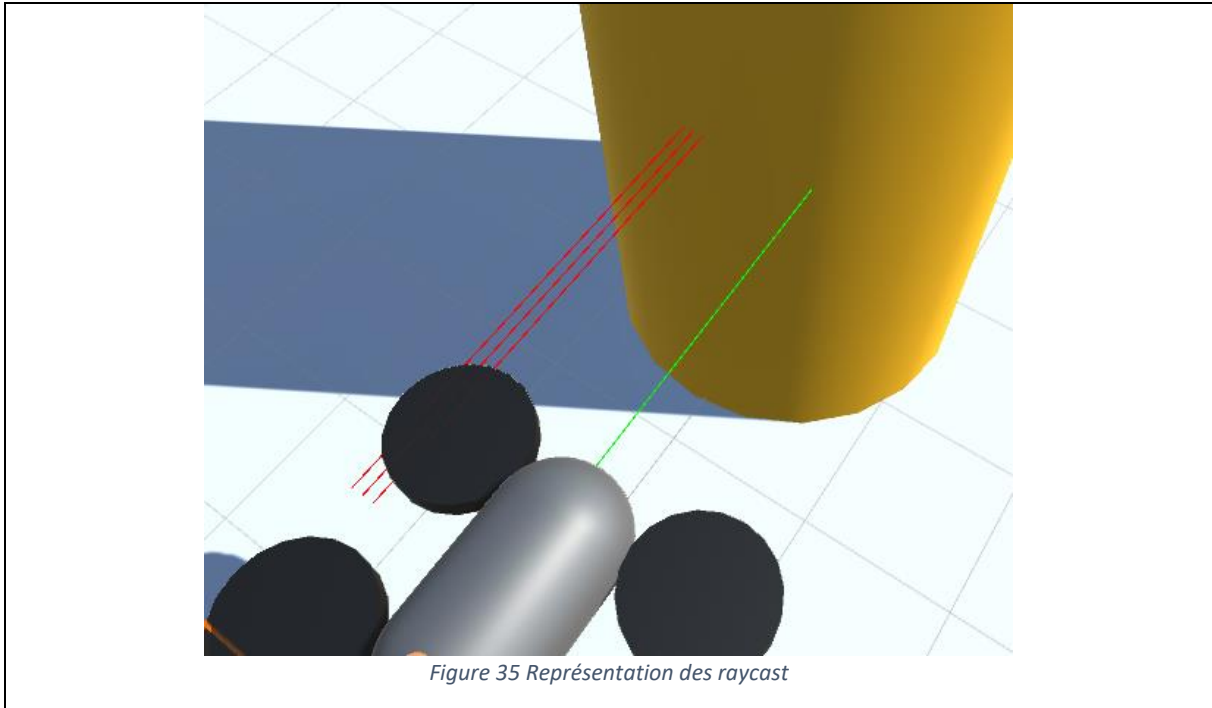
Figure 34 Visualisation d'un raycast

III-3-1 Capteur mono-point

Les capteurs mono-points sont représentés par un raycast (projection de rayons) dans la direction voulue. Si le rayon entre en contact avec un objet, le drone recevra l'information de distance entre lui et cet objet.

A chaque instant, un rayon vert est affiché pour visualiser le raycast. Si une collision entre le rayon et un obstacle, un rayon rouge apparaîtra.

Il faut noter que le point de départ des raycast est le centre du drone, il faut donc avoir ceci en tête lors de la définition de minD.



Voici le code de la fonction qui effectue le test de distance.

```
float monopointCaptor(Vector3 position, Vector3 vec, float range)
{
    RaycastHit hit;
    Ray downRay = new Ray(position, vec.normalized*range);
    if (Physics.Raycast(downRay, out hit))
    {
        Debug.DrawRay(position, vec.normalized * range, Color.red, 20, true);
    }
    else
    {
        Debug.DrawRay(position, vec.normalized * range, Color.green, -1, true);
    }

    return hit.distance;
}
```

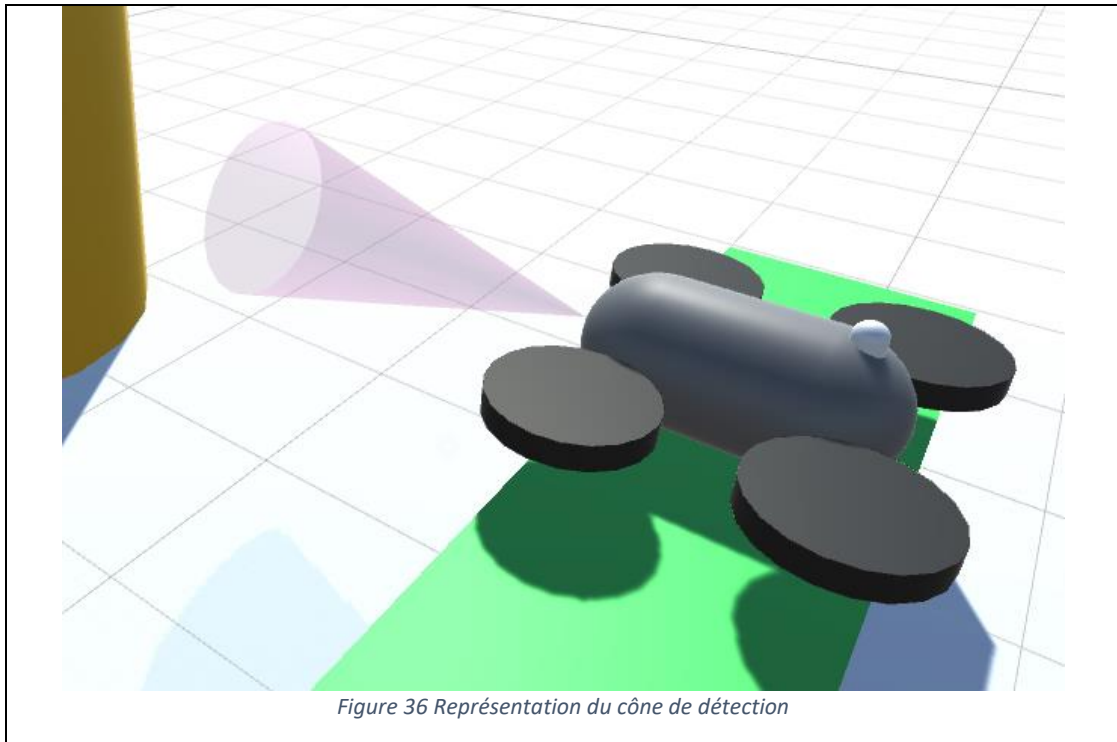
Cette fonction prend 3 paramètres, la position de départ du capteur, sa direction, et la distance maximale de détection. La fonction renvoie la distance à l'obstacle

III-3-2 Capteur mono-point (cône de détection)

Une représentation plus fidèle à la réalité consiste à considérer un cône de détection, comme les capteurs réels.

Il faut donc modéliser un cône, lui affecter un matériau transparent. Il faut également lui affecter un « Mesh Collider », avec les options « Convex » et « IsTrigger ». Il faut ensuite ajouter un « Rigidbody » au drone, décocher l'option « UseGravity », et cocher « isKinematic ».

Ces options vont permettre au cône de pouvoir détecter lorsqu'un autre objet le traverse, et également de lui permettre de traverser les autres objets.



Au niveau du code, La fonction « OnTriggerEnter » est appelé lors d'une collision. On utilise également la fonction « OnTriggerStay » qui est appelé régulièrement toute la durée de la collision. Ainsi, on peut rajouter deux attributs privés à la classe, un booléen qui est à « Vrai » lorsqu'un obstacle se trouve dans le cône, faux sinon, et un nombre flottant contenant la distance minimale à l'obstacle. On peut alors construire une fonction (ici « realMonopointCaptor »). Qui renvoie la distance à l'obstacle, à la manière d'un vrai capteur monopoint.

Voici l'extrait de code :

```
private bool isTouched = false;
public float touched_dist;

float realMonopointCaptor()
{
    if (isTouched)
    {
        return touched_dist;
    }

    return float.PositiveInfinity ;
}
```

```

}

void OnTriggerEnter(Collider collision)
{
    isTouched = true;
    touched_dist = Vector3.Distance(this.transform.position,
collision.gameObject.GetComponent<Collider>().ClosestPointOnBounds(transform.position));
}

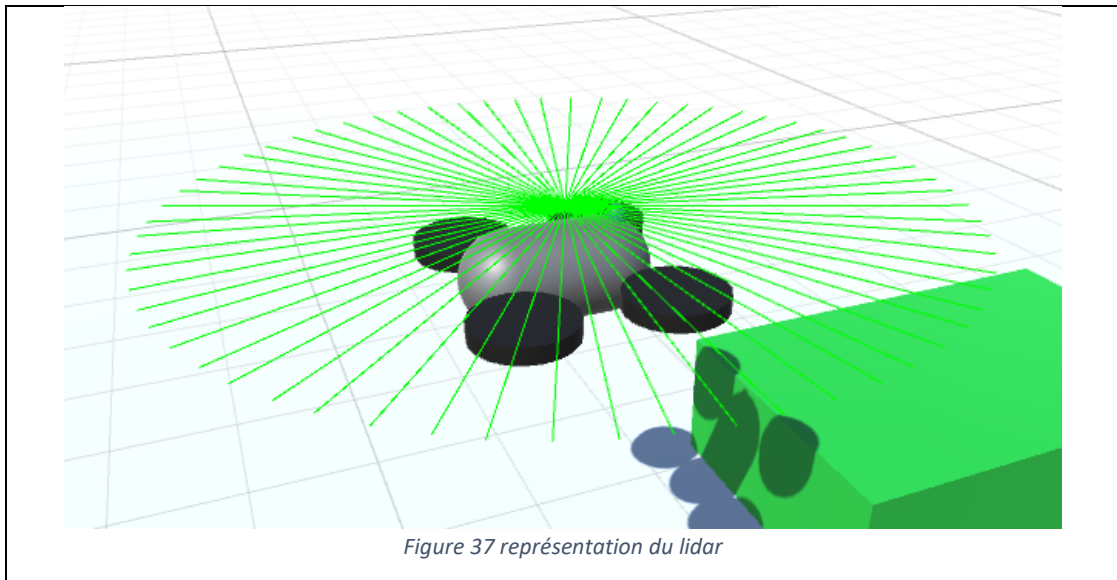
void OnTriggerStay(Collider collision)
{
    isTouched = true;
    touched_dist = Vector3.Distance(this.transform.position,
collision.gameObject.GetComponent<Collider>().ClosestPointOnBounds(transform.position));
    Debug.Log(touched_dist);
}

void Update(){
    (...)
    realMonopointCaptor()
    (...)
    isTouched = false;
}

```

III-3-3 Lidar

Les lidars ont été modélisés grâce à plusieurs Raycasts. Tout se passe comme si le lidar était fixé sur le drone : Les Raycasts ont tous pour origine un point au-dessus du centre du drone, et leur direction est obtenue en effectuant une rotation du vecteur « vers l'avant » du drone par rapport à l'axe verticale.



On peut paramétrer la résolution angulaire en changeant le nombre de rayons émis (sur la figure, 64). A chaque appel à la fonction lidar, le nuage de point détecté est retourné.

Voici le code de la fonction « lidar » :

```

List<Vector3> lidar(int nbRayon = 64)
{
    List<Vector3> pointList = new List<Vector3>();
    for (int i=0; i< nbRayon; i++)
    {
        Vector3 vec = Quaternion.AngleAxis(i*360/ nbRayon, transform.up) *
transform.forward;
        vec.Normalize();
        RaycastHit hit;
        Vector3 centerLidar = transform.position + transform.up * 0.33f; // le
centre se trouve au-dessus du lidar (suffisamment haut pour dépasser la led, mais le
plus bas possible)
        Ray downRay = new Ray(centerLidar, vec * minD);
        if (Physics.Raycast(downRay, out hit))
        {
            if (hit.distance < minD) // attention, le raycast détecte une
collision même si l'obstacle est supérieur à minD.
            {
                Debug.DrawRay(centerLidar, vec * minD, Color.red, 20, true);
                pointList.Add(hit.point);
            }
            Else
            {
                Debug.DrawRay(centerLidar, vec * minD, Color.green, -1, true);
            }
        }
        else
        {
            Debug.DrawRay(centerLidar, vec * minD, Color.green, -1, true);
        }
    }
    return pointList;
}

```

III-4 Algorithmes simulés

III-4-1 Variantes de Shift Avoidance

Cet algorithme ressemble très fortement à l'algorithme Shift Avoidance décrit dans l'état de l'art. La différence est que le décalage vers la droite s'effectue non pas en faisant une rotation avec pour centre l'objectif, mais en faisant une translation vers la droite.

Voici le pseudo-code :

```

etat = 1
delta = 0
// on exclu ici les phases d'atterrissage et de décollage, elle ne sont pas spécifique au problème

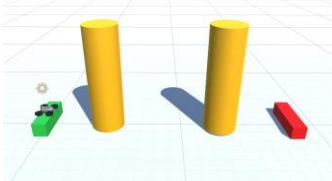
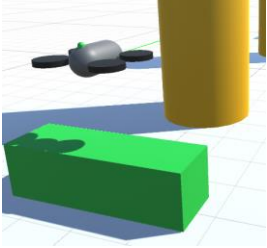
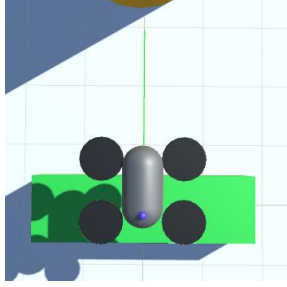
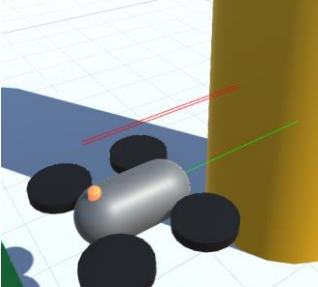
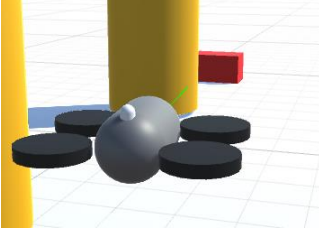
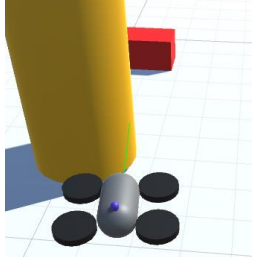
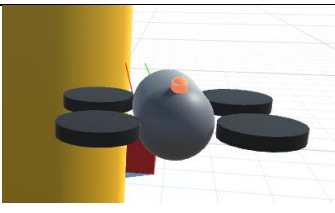
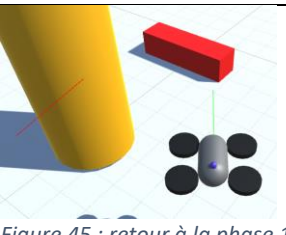
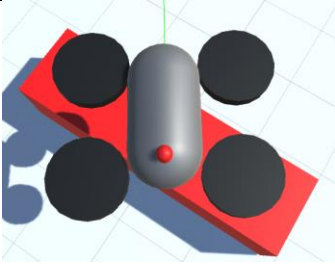
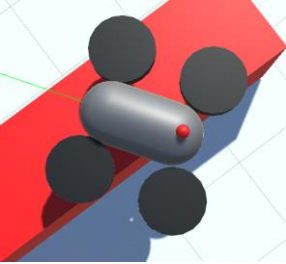
tant que vrai:
    si etat == 1: // état normal
        si !obstacle:
            avancer vers la cible

```

```
        sinon:
            etat = 2
        si etat == 2: // manœuvre d'évitement
            se décaler vers la droite
            si obstacle:
                delta = 0
            si delta < SEUIL: // il faut rajouter une marge, sinon il y aura contact entre
l'obstacle et les hélices
                delta+=deplacement
            sinon:
                etat = 1
                rotation // de manière à faire face à l'objectif
```

Tout comme Shit avoidance, son plus gros défaut réside dans le fait qu'il n'y a rien de prévu si jamais un obstacle se trouve sur la droite du drone.

Voici une mise en situation (à lire de gauche à droite, puis de haut bas):

| | | | |
|--|--|---|--|
|  <p>Figure 38 Illustration de la situation</p> | <p>Le drone part du bloc vert pour aller au bloc rouge. Deux obstacles sont situés sur sa trajectoire.</p> |  <p>Figure 39 Phase 0 : Décollage</p> | <p>La première phase est la phase de décollage. La led est verte lors de cette phase. Le drone monte jusqu'à atteindre une hauteur suffisante.</p> |
|  <p>Figure 40 Phase 1 : déplacement vers l'objectif</p> | <p>Une fois le décollage fini, le drone passe en phase 2 : il avance en ligne droite vers l'objectif. Si son capteur l'informe d'un obstacle sur sa route, il passera en phase 2</p> |  <p>Figure 41 Phase 2 : Manoeuvre d'évitement</p> | <p>La led du drone est orange. Cela signifie qu'il est passé en phase 2, manœuvre d'évitement. Il se décale sur la droite tant qu'il y a un obstacle, puis continue de se décaler pour laisser une marge. Ici, l'obstacle a été détecté 2 fois, puis le drone attend la fin de la marge.</p> |
|  <p>Figure 42 Phase 4 : Rotation</p> | <p>Après la manœuvre d'évitement, le drone effectue une rotation sur lui-même, de manière à regarder l'objectif</p> |  <p>Figure 43 Retour à la phase 1</p> | <p>Une fois la phase 4 terminée, le drone repasse à la phase 1, et avance en direction de l'objectif.</p> |
|  <p>Figure 44 Retour à la phase 2</p> | <p>Le drone a détecté le deuxième obstacle (raycast rouge), puis de la même manière, se déporte sur la droite.</p> |  <p>Figure 45 : retour à la phase 1</p> | <p>Après les phases 2 et 4 terminée, le drone revient à la phase 1.</p> |
|  <p>Figure 46 Phase 5 : atterrissage</p> | <p>Arrivé au-dessus de l'objectif, le drone passe en phase 3, atterrissage, et sa led devient rouge.</p> |  <p>Figure 47 Fin de la simulation</p> | <p>Enfin, le drone est posé.</p> |

III-4-2 Implémentation de Virtual Force Field sans carte

Le VFF a été implémenté de façon à ne réagir qu'aux points détectés à un instant par le lidar. Le fait de ne pas sauvegarder le nuage de points à chaque étape permet d'effectuer moins de calcul, et de pouvoir gérer les objets mouvants.

On peut également noter que comme le lidar est un capteur omnidirectionnel, l'orientation du drone a peu d'importance, et donc il n'y a pas de rotation du drone.

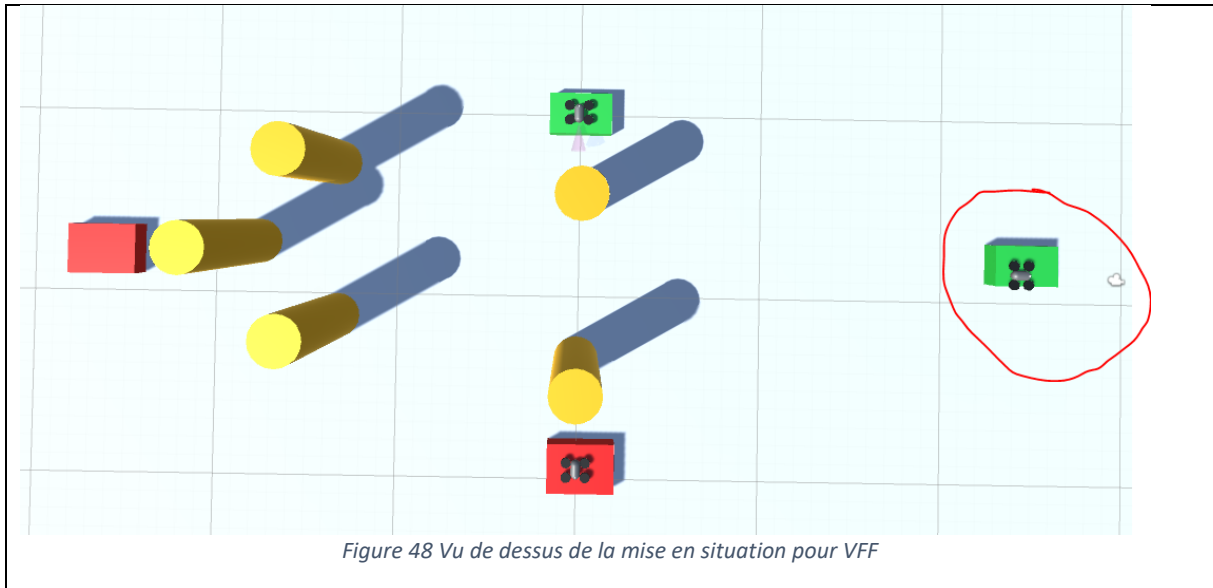
Voici le code utilisé pour déterminer la direction du drone :

```
else if (state == 1) // aller vers la cible
{
    direction = (target.transform.position + new Vector3(0, minH +
target.GetComponent<Collider>().bounds.size[1], 0) - this.transform.position);

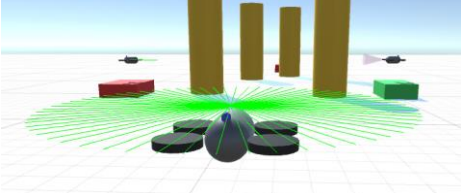
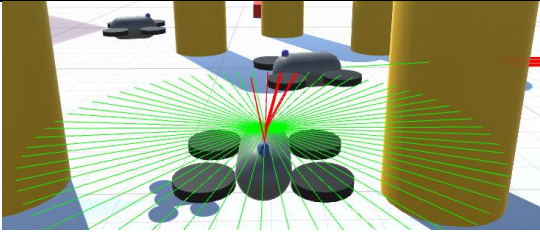
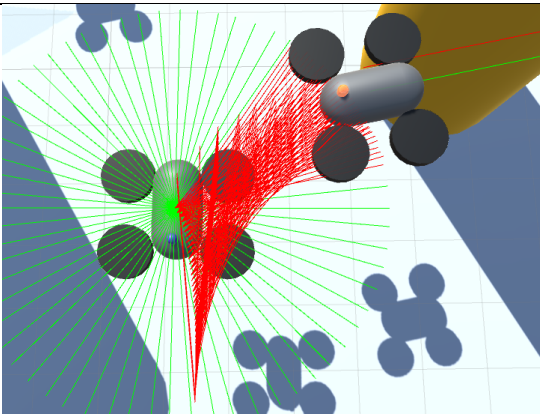
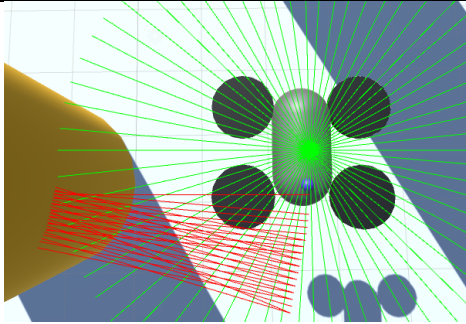
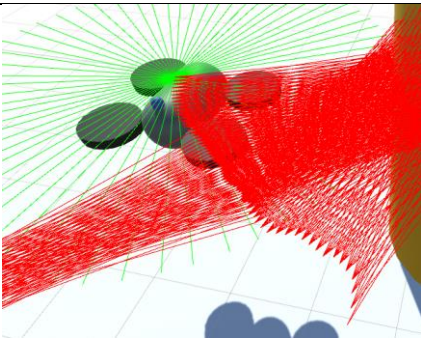
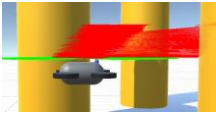
    if (direction.magnitude < 0.5) // si le drone est arrivé
    {
        setState(3);
    }
    else
    {
        List<Vector3> points = lidar();
        Vector3 dir_obstacle;
        for (int i = 0; i < points.Count; i++)
        {
            dir_obstacle = transform.position - points[i];
            direction += coef * dir_obstacle / (dir_obstacle.magnitude *
dir_obstacle.magnitude);
        }
        this.transform.Translate(direction.normalized * speed, Space.World);
    }
}
```

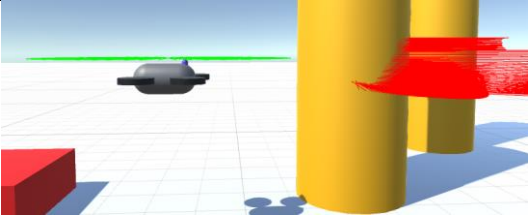
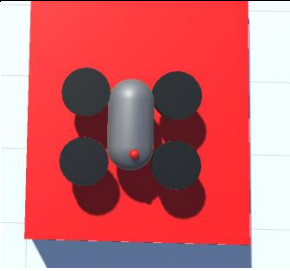
Ce code est très proche du pseudo code décrit précédemment. Direction est le vecteur destination, la valeur 0.5 correspond au epsilon utilisé pour déterminer le moment où le drone est arrivé. L'état 3 est l'état d'atterrissage. Points contient la liste des obstacles.

Voici une mise en situation :



Le drone étudié a été entouré en rouge. Son objectif est d'aller au bloc rouge le plus à gauche. Les deux autres drones vont servir d'obstacles mouvants. Ils vont échanger de places lors de la simulation. Ils utilisent tous les deux la première variante de shift avoidance détaillé ci-dessus. Le drone du dessus va cependant utiliser le capteur monopoint amélioré.

| | |
|---|--|
|  <p><i>Figure 49 décollage et déploiement du lidar</i></p> | <p>Après la phase décollage, le drone utilise le lidar pour repérer les obstacles. Ici, le drone se dirige en ligne droite vers l'objectif.</p> |
|  <p><i>Figure 50 Contact avec une cible mouvante</i></p> | <p>Ici, le drone détecte l'autre drone. Il est relativement loin, mais le drone va tourner légèrement à gauche pour éviter la collision.</p> |
|  <p><i>Figure 51 fin de la première manoeuvre d'évitement</i></p> | <p>On remarque ici un léger changement de direction vers la gauche par rapport aux premier lancé de rayons (ceux partant du bas de l'écran). On remarque surtout la prise en compte de l'obstacle mouvant : sa trajectoire aurait été beaucoup plus impacté si l'algorithme conservait le nuage de point d'une étape à l'autre.</p> |
|  <p><i>Figure 52 autre détection d'obstacle</i></p> | <p>Là encore, on remarque un changement de trajectoire à l'approche d'un obstacle.</p> |
|  <p><i>Figure 53 Comportement dans un couloir</i></p> | <p>Deux choses sont à noter ici. La première est le caractère légèrement oscillatoire du drone entre deux obstacles. Ici l'oscillation car le coefficient de VFF a été réglé à postériori. L'autre chose remarquable, c'est que le drone est descendu en altitude pour passer cette difficulté.</p> <div data-bbox="810 1821 1026 1933">  </div> <p><i>Figure 54 autre vue pour illustrer la baisse d'altitude</i></p> |

| | |
|--|--|
|  <p>Figure 55 Reprise d'altitude du drone</p> | <p>Après la difficulté, le drone reprend son altitude en se dirigeant vers l'objectif.</p> |
|  <p>Figure 56 Le drone atteint son objectif</p> | <p>Finalement, le drone atteint son objectif et se pose.</p> |

III-5 Autres simulations

Afin de créer les schémas d'explication par mise en situation de l'algorithme Virtual Force Field, J'ai effectué une simulation 2D (vue dessus) en python.

Le code a été écrit en python 3 pur. Seule la bibliothèque standard intégré à python « matplotlib » a été utilisé pour visualiser les courbes. Cette bibliothèque permet d'afficher toutes sorte de données, comme des graphes 2D (utilisé ici), des graphes 3D, ou encore des images.

On peut contrôler la liste des obstacles en modifiant « obstacles », la destination en modifiant « but », ainsi que le coefficient lié aux vecteurs obstacles avec la variable « coef ».

Le drone commence toujours sa course en [0, 0]

Voici le code :

```
import matplotlib.pyplot as plt
epsilon = 0.1
n = 500
x = [i*epsilon for i in range(n)]
y = [0]*n

obstacles = [[x[n//3], 1], [30, -4]]
but = [x[-1], 0]

coef = 50
seuil = 0.1

for i in range(1,n):
    position = [x[i-1], y[i-1]]
    vecteur = [but[0]-x[i-1], but[1]-y[i-1]]
    print(vecteur)
    norme = (vecteur[0]**2+vecteur[1]**2)**0.5
    if norme < 10: # augmente artificiellement le vecteur objectif
```

```

lorsque le drone est trop proche
    vecteur = [10*vecteur[0]/norme, 10*vecteur[1]/norme]
    for o in obstacles:
        vec_ob = [-o[0]+x[i-1], -o[1]+y[i-1]]

        norme = (vec_ob[0]**2+vec_ob[1]**2)**0.5
        vecteur[0] += coef*vec_ob[0]/(norme**2)
        vecteur[1] += coef*vec_ob[1] / (norme **2)
    # vecteur à la bonne valeur, maintenant, il faut l'appliquer pour
    connaitre y[i]
    print(vecteur)
    y[i] = y[i-1]+epsilon*vecteur[1]/vecteur[0]

plt.plot(x,y)
xo = [o[0] for o in obstacles]
yo = [o[1] for o in obstacles]
plt.plot(xo, yo, 'ro')
plt.plot([but[0]], [but[1]], 'gs')
plt.axis('equal')
plt.show()

```

IV Bibliographie

1. *Drone*, Wikipedia, 20/11/2018, <https://fr.wikipedia.org/wiki/Drone>
2. *Drone quadrirotor*, Wikipedia, 20/11/2018, https://fr.wikipedia.org/wiki/Drone_quadrirotor
3. *Multirotor*, Wikipedia, 20/11/2018, <https://en.wikipedia.org/wiki/Multirotor>
4. *Emploi : pilote de drones, un marché du travail saturé*, Adrien Danglarde, <https://www.pratique.fr/actu/emploi-pilote-de-drones-un-marche-du-travail-sature-3723622.html>
5. *Système lidar*, deltadrone, 28/11/2018, <http://www.deltadrone.com/fr/systemes/systeme-lidar>
6. *12 Top Collision Avoidance Drones And Obstacle Detection Explained*, Fintan Corrigan, 15 Octobre 2018, <https://www.dronezon.com/learn-about-drones-quadcopters/top-drones-with-obstacle-detection-collision-avoidance-sensors-explained/>
7. *Safe Local Exploration for Replanning in Cluttered Unknown Environments for Micro-Aerial Vehicles*, Helen Oleynikova, 12 mars 2018 <https://arxiv.org/pdf/1710.00604.pdf>
8. *Collision Avoidance Algorithms For Unmanned Aerial Vehicles Using Computer Vision*, Hani Sedaghat-Pisheh, 2017, <https://pdfs.semanticscholar.org/5bd2/1f685e20d78f5a72656c9c63a16d60534520.pdf>
9. *Collision Avoidance Library*, Intel, mars 2017 <https://github.com/intel/collision-avoidance-library>
10. *THE VECTOR FIELD HISTOGRAM - FAST OBSTACLE AVOIDANCE FOR MOBILE ROBOTS*, J. Borenstein et Y. Koren, Juin 1991, <http://www-personal.umich.edu/~johannb/Papers/paper16.pdf>

L'intégralité du code est disponible et téléchargeable à l'adresse :

https://github.com/supermouette/drone_collision_avoidance/

Il a été écrit avec unity 5.5.0f3. Pour le faire fonctionner, téléchargez le projet, ouvrez le dans Unity, et si besoin, ouvrez la scène « drone.unity ».