

# MODELISATION 3D ET GESTION DE TRAJECTOIRE D'UN DRONE

Emmanuel Douet, Livie Romanet, Tierno-Alpha Tall, Angel Lagrange

7 avril 2023



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Première partie : Algorithmie</b>	<b>3</b>
2.1	Présentation du programme initial en 2D . . . . .	3
2.1.1	Définition des variables . . . . .	3
2.1.2	Etapes de l'algorithme . . . . .	4
2.1.3	Utilisation des programmes donnés . . . . .	4
2.2	Passage de la 2D à la 3D . . . . .	4
2.2.1	La fonction Distance3D . . . . .	5
2.2.2	La fonction NodeIndex3D . . . . .	5
2.2.3	La fonction Min_fn3D . . . . .	5
2.2.4	La fonction Expand_array3D . . . . .	5
2.2.5	La fonction Insert_open3D . . . . .	5
2.2.6	Le programme Astar3D . . . . .	5
2.2.7	Résultats . . . . .	5
2.2.8	Les fonctions Pyramide3D et coordobsta . . . . .	6
2.3	Représentation visuelle des obstacles . . . . .	7
2.4	Complexité . . . . .	7
2.5	Lissage de la trajectoire . . . . .	8
<b>3</b>	<b>Deuxième partie : Modélisation 3D</b>	<b>9</b>
3.1	Fonctionnement de l'éditeur 3D . . . . .	9
3.1.1	Création d'objets . . . . .	9
3.1.2	Création d'un environnement . . . . .	9
3.1.3	Point de vue . . . . .	10
3.1.4	Aspect artistique . . . . .	10
3.2	Lien entre le modèle et le monde 3D . . . . .	11
3.2.1	Différence de proportions . . . . .	11
3.2.2	Données du monde 3D . . . . .	11
3.2.3	Modifier l'environnement 3D . . . . .	12
3.2.4	Tracer une courbe . . . . .	12
<b>4</b>	<b>Conclusion</b>	<b>12</b>
<b>5</b>	<b>Bibliographie</b>	<b>13</b>
<b>6</b>	<b>Annexe</b>	<b>14</b>

# 1 Introduction

This project's aim is to create a functional dynamical drone (or UAV) model, to model it in 3D and to build a program allowing the drone to automatically avoid given obstacles. Our group is in charge of the algorithmic and 3D modelling part on respectively Matlab and Simulink.

Our objectives are to model a realistic 3D drone evolving in an environment and to build an algorithm enabling it to avoid encountering obstacles on its path while taking its dimensions into account.

The drone on which we worked is the Crazyflie product from Bitcraze. These Crazyflies allow the coordination of several drones) with each other or, for example, the experimentation of a trajectory optimisation algorithm, as in our case.



In fact, they include an open source code so that we can modify them and use them for our project, and a simple mechanism, so that we can avoid excessive complication of the task.

To begin our project we had at our disposal an algorithm that was already structured but that only worked in 2D. Therefore we had :

- To adapt the given algorithm to a 3D version
- To model the drone and its surroundings in 3D
- To animate the flight paths in 3D

The second group had to build a functional model of the drone and to understand how to make the drone move in space according to the rotation speed of the propellers, etc...

## 2 Première partie : Algorithmie

### 2.1 Présentation du programme initial en 2D

Pour la partie gestion de trajectoire, notre groupe est parti d'un algorithme déjà fonctionnel mais seulement en 2D. Nous devions donc l'adapter et gérer les obstacles en 3D.

Ce dernier fonctionne grâce à la méthode de "A star" aussi appelé A\* :

Le programme A\* est un algorithme de recherche de chemin qui permet de trouver le chemin le plus court entre un point de départ et un point d'arrivée dans un graphe pondéré à l'aide de [tentatives successives](#). Avant de passer à l'explication du programme donné, tachons de bien comprendre comment fonctionne l'algorithme A\* étape par étape afin de pouvoir l'utiliser sur Matlab :

#### 2.1.1 Définition des variables

Tout d'abord nous devons initialiser notre environnement et plusieurs variables qui vont nous permettre de faire fonctionner l'algorithme :

- **La map** qui est une matrice contenant toutes les coordonnées de notre environnement en 2D. Chaque point correspondant à un obstacle se voit être attribué la valeur -1. On assigne la valeur 2 au reste des points. Cela permettra de ne pas traiter les points "obstacles" dans la suite de l'algorithme.
- **Les obstacles** qui sont toutes les coordonnées des points non atteignables de la map.
- **Les noeuds** qui correspondent aux coordonnées de tous les points atteignables de la map. Ces derniers disposent de 2 variables attribuées :
  1. **le coût du trajet le plus court** pour arriver jusqu'à lui. C'est-à-dire la distance minimale à parcourir pour l'atteindre.
  2. **Un noeud parent** qui correspond au noeud précédent dans le chemin le plus court pour l'atteindre par rapport au noeud de départ.
- **Le noeud de départ** et **le noeud d'arrivée**, qui correspondent respectivement aux coordonnées du point de départ et d'arrivée dans l'espace.
- **La liste OPEN** qui contient tous les noeuds qui doivent encore être examinés par l'algorithme.
- **La liste CLOSED** qui contient tous les noeuds qui ont déjà été examinés par l'algorithme.

### 2.1.2 Etapes de l'algorithme

1. On place dans l'espace le **noeud de départ** et le **noeud d'arrivée**.
2. On initialise la liste OPEN qui ne contient au début que le **noeud de départ**, puis la liste CLOSED vide car aucun noeud n'a encore été étudié au début.
3. On sélectionne le **noeud** avec le coût du trajet le plus faible de la liste OPEN.
4. Pour chaque **noeud adjacent** au **noeud sélectionné**, on calcule le coût du chemin pour aller de la **source** à **l'adjacent** en passant par le **noeud sélectionné**. On ajoute le coût total au coût du **noeud sélectionné** pour obtenir le coût total de **l'adjacent**.
5. Si le **noeud adjacent** n'est pas dans la liste OPEN, nous l'ajoutons à la liste OPEN et nous mettons à jour son coût total et son parent (**le noeud sélectionné**).

Si le **noeud adjacent** est déjà dans la liste OPEN, nous vérifions s'il est plus rapide d'y accéder à partir du **noeud sélectionné**. Si c'est le cas, nous mettons à jour le coût total et le parent du **noeud adjacent**.

6. On répète l'opération pour tout les **nœuds adjacents** du **noeud sélectionné**.
7. On déplace ensuite le **noeud sélectionné** vers la liste CLOSED, qui contient les nœuds déjà examinés.
8. On repète les étapes 3 à 7 jusqu'à ce que le **noeud d'arrivée** soit **sélectionné**. Ainsi on a trouvé le chemin le plus court entre les deux nœuds et on peut remonter le chemin avec à partir du **noeud d'arrivée** en suivant les parents de chaque nœud jusqu'au **noeud de départ**.

**Remarque :** Si la liste OPEN est vide et que le **noeud d'arrivée** n'a pas été atteint alors il n'existe aucun chemin nous permettant de réaliser ce trajet.

### 2.1.3 Utilisation des programmes donnés

Le programme qui nous a été donné est construit sur 6 sous programmes qui nous permettront d'appliquer l'algorithme ci-dessus :

- **"min\_fn.m"** prend en entrée la liste OPEN et retourne le point avec le coût le plus petit  
*Utilisé directement dans l'étape 2*
- **"NodeIndex.m"** retourne la localisation d'un point de la liste OPEN  
*Permet de récupérer les coordonnées des nœuds sélectionnés dans l'étape 3 afin de pouvoir calculer leur coût de trajet, etc...*
- **"Expand\_array.m"** cette fonction prend un point et renvoie la liste de ses successeurs (L'un des critères étant qu'aucun des successeurs ne figure sur la liste CLOSED) ainsi que trois valeurs primordiales pour faire fonctionner le programme :
  1. **hn**, le coût du chemin parcouru depuis le noeud de départ jusqu'au noeud actuel.,
  2. **gn**, une estimation du coût restant pour atteindre l'objectif depuis le noeud actuel.
  3. **fn**, l'estimation totale du coût pour atteindre le noeud de destination en passant par le noeud actuel tel que  $fn = gn + hn$ .

L'algorithme A\* utilise cette valeur pour évaluer la priorité des nœuds à explorer dans le programme **"min\_fn"**. En choisissant les nœuds avec les valeurs fn les plus basses en premier, A\* explore d'abord les nœuds les plus prometteurs en termes de coût pour atteindre la destination.

*Fait tout les calculs présents dans l'étape 4 et permettra dans l'étape 8 de remonter tout les parents du noeud d'arrivée afin de trouver le chemin le plus court*
- **"insert\_open.m"** remplit la liste OPEN avec ses valeurs associées  
*Permet de faire la sélection présente durant l'étape 5*
- **"Distance.m"** permet de donner la distance entre deux points  
*On utilisera ce dernier afin de calculer la distance noeud sélectionné/adjacent dans "Expand\_array.m"*

## 2.2 Passage de la 2D à la 3D

Notre programme fonctionne donc bien en 2D, il nous faut maintenant le passer en 3D pour pouvoir l'adapter à un environnement réel. Ainsi pour chaque programme nous avons rajouté une coordonnée z dans l'équation. La map est devenue une matrice avec 3 colonnes (x, y et z) qui est sera notre environnement en 3D.

### 2.2.1 La fonction Distance3D

Pour le programme "[Distance3D.m](#)" il nous suffit de passer de la norme 2 dans  $R^2$  à la norme 2 dans  $R^3$  :

$$\sqrt{x^2 + y^2} \rightarrow \sqrt{x^2 + y^2 + z^2}$$

### 2.2.2 La fonction NodeIndex3D

Sur "[NodeIndex3D.m](#)" nous devions rajouter une condition supplémentaire sur la recherche d'un point dans la liste OPEN. Le programme de base recherche dans OPEN un nœud avec des coordonnées (x,y) données, nous avions juste à y rajouter une coordonnée en z

### 2.2.3 La fonction Min\_fn3D

Pour "[min\\_fn3D.m](#)" comme pour **Distance3D.m** nous avons ajouté une composante z en plus dans les entrées de la fonction et dans la condition d'arrêt : "si le programme détecte le nœud d'arrivée alors on a trouvé le chemin."

### 2.2.4 La fonction Expand\_array3D

De même pour "[Expand\\_array3D.m](#)" où nous avons ajouté une composante en "z" au nœud sélectionné, au nœud d'arrivée, et sur la taille maximale de la map.

**Remarque :** On repère sur notre programme (cf Annexe) les variables hn, gn et fn :

- **hn** :  $exp\_array(exp\_count, 4) = hn + distance3D(node\_x, node\_y, node\_z, s\_x, s\_y, s\_z)$   
Le hn qui apparaît dans la formule correspond au hn du noeud parent.
- **gn** :  $exp\_array(exp\_count, 5) = distance3D(xTarget, yTarget, zTarget, s\_x, s\_y, s\_z)$
- **fn** :  $exp\_array(exp\_count, 6) = exp\_array(exp\_count, 4) + exp\_array(exp\_count, 5)$

### 2.2.5 La fonction Insert\_open3D

Pour "[Insert\\_open3D.m](#)", nous devions comprendre le fonctionnement de liste OPEN. Elle contient chaque nœud et leurs caractéristiques : coordonnées du nœud, du parent et les valeurs hn, gn et fn. Il fallait donc rajouter une troisième dimension aux 2 premières caractéristiques.

### 2.2.6 Le programme Astar3D

Pour "[Astar3D.m](#)" il nous fallait seulement appliquer l'algorithme de base, cependant en 3 dimensions avec par conséquent une composante supplémentaire.

**Remarque :** A\* remonte les parents du noeud d'arrivée pour récupérer le chemin le plus court dans un programme écrit dans "Astar3D.m". Ce dernier a du être modifié comme tous les autres (Cf "[Figure 7 en Annexe](#)").

### 2.2.7 Résultats

Ainsi avec toutes ces modifications réalisées le programme nous permet d'obtenir ce type de résultat :

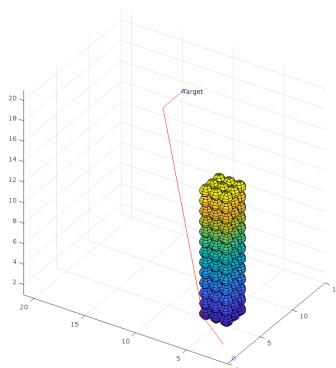


Figure 2.1 Trajectoire du drone esquivant un obstacle (pavé droit coloré) allant du point (0,0,0) au point "Target"

Néanmoins pour pouvoir gérer les obstacles en 3D nous devions les modéliser sur Matlab à travers de simples coordonnées. Pour A\* la map est remplie de points possédant une valeur. Ainsi, il nous fallait des méthodes afin de déterminer comment modéliser une chaise ou un obstacle de forme quelconque dans notre map. Nous avons décidé de simplifier toutes les formes possibles à des cubes ou des pyramides afin de faciliter les calculs et ne pas surcharger de détails la map.

**Remarque :** ce raisonnement ne réduit pas la pertinence des trajets générés. En effet modéliser un obstacle par une zone plus large que celle qu'il occupe en réalité n'aura pour effet principal que d'augmenter les chances de ne pas rentrer en collision avec ce dernier et de simplifier la map.

### 2.2.8 Les fonctions Pyramide3D et coordobsta

Nous avons donc décidé de nous limiter à des obstacles sous forme de pavés droits (pour modéliser des immeubles par exemple) ainsi que des pyramides. Il nous fallait alors des programmes capables de calculer les coordonnées au sein d'un volume que nous pourrions placer aisément sur notre map. Nous avons ainsi codé les fonctions '**Coordobsta**' et '**Pyramide3D**'.

**Coordobsta** génère un parallélépipède rectangle à l'aide de trois points correspondant à ses extrémités, ainsi que d'un dernier point ; son "origine" dans l'espace, qui nous permettra de placer le pavé où l'on souhaite.

**Pyramide3D** génère une pyramide à base carrée de hauteur donnée en réalisant un empilement de carrés de côté de plus en plus petit.

Nous avons d'abord pensé à commencer avec une base carrée de côté  $c$  et à entasser sur celle-ci des bases similaires de côté  $c - 2i$  (avec  $i$  le nombre d'itérations) autant de fois qu'il était possible. Cependant cette approche était incomplète, car pour une longueur de base donnée, nous ne pouvions pas contrôler la hauteur. Il nous a donc fallu implémenter un incrément spécial pour la diminution de la largeur de chaque côté. Celui-ci est de  $\lfloor c/h \rfloor$  ; avec  $c$  le côté de la base de la pyramide, et  $h$  sa hauteur.

Néanmoins une contrainte subsiste : la longueur de la base doit être supérieure à deux fois la hauteur ( $c \geq 2h$ ), sinon certains points de la pyramide auraient des coordonnées non entières, qui ne seraient alors pas reconnues par l'algorithme A\*. Cela pourrait être résolu en multipliant toutes les coordonnées par une puissance de 10, mais reviendrait seulement à générer une pyramide autant de fois plus volumineuse. De plus, avec une hauteur non conforme, des 'trous' pourraient apparaître dans le volume, ce qui comprometttrait sa catégorisation même d'obstacle pour le calcul des trajectoires. Initialement codé en Python, nous avons ensuite retranscrit cet algorithme pour Matlab.

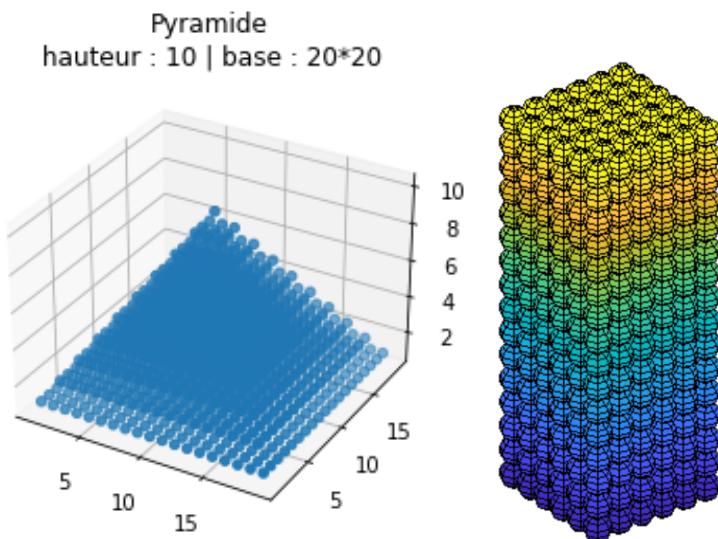


Figure 2.2 Volumes produits à l'aide de Pyramide3D et coordobsta

Les volumes ainsi générés sont des listes de points de l'espace, auxquels on affectera la valeur '-1' dans la map pour qu'ils soient reconnus comme des obstacles.

## 2.3 Représentation visuelle des obstacles

L'affichage par défaut des obstacles dans Matlab, peu adapté à une vue en 3D, ne nous permettait qu'une vision peu claire de ceux-ci. Pour remédier à cela nous avons opté pour une solution alternative : chaque point appartenant à un obstacle sera représenté par un polyèdre approximant une sphère. Celui-ci aura un nombre relativement petit de côtés, ce qui nous permettra de limiter les temps de calcul, tout en atteignant la visibilité désirée. En effet nous calculons les coordonnées nécessaires à l'affichage d'un polyèdre élémentaire une seule fois pour une exécution donnée du programme. Celles-ci sont ensuite translatées dans l'espace pour que chaque point soit affiché.

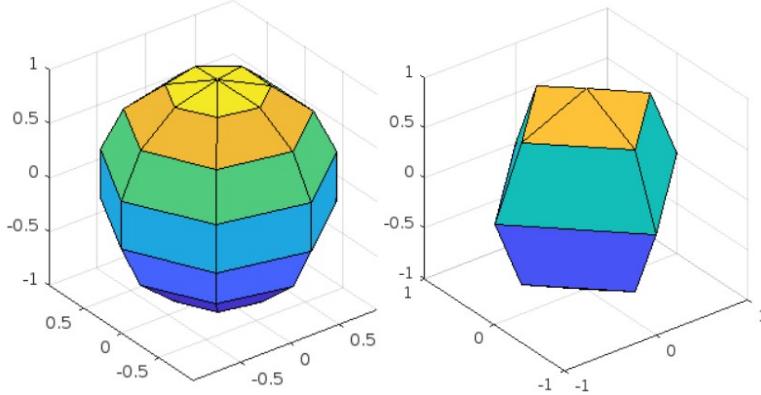


Figure 2.3 Exemples de polyèdres élémentaires

De plus, le nombre limité de faces n'entache pas la clarté visuelle, étant donné que chaque volume élémentaire apparaîtra vu de loin.

Ci-dessous [en annexe](#) un comparatif des visuels en question (Fig. 16).

## 2.4 Complexité

La rapidité d'exécution de notre programme aurait varié avec la puissance de l'ordinateur utilisé, cependant les temps de calcul effectifs ont eu un impact déterminant sur notre approche de la problématique. Nous avons très vite remarqué que faire tourner l'algorithme pouvait s'avérer chronophage. Nous avons donc opté pour travailler dans un espace de taille  $50 \times 50 \times 50$  au maximum pour générer les trajectoires.

Cependant dans le but d'estimer plus précisément les temps de calcul, et de vérifier si un meilleur compromis durée/taille maximale était possible nous avons étudié la complexité en temps du programme de plus près.

**Remarque :** on notera ici  $t$  le temps d'exécution du programme, et  $n$  la longueur d'un côté de la grille.

En modélisant  $t=f(n)$  à l'aide du logiciel Regressi, nous avons trouvé que la durée d'exécution du programme suivait la loi :

$$t = k \times n^p \quad \text{avec } k = 2.58 \times 10^{-5} \text{ et } p = 3.62$$

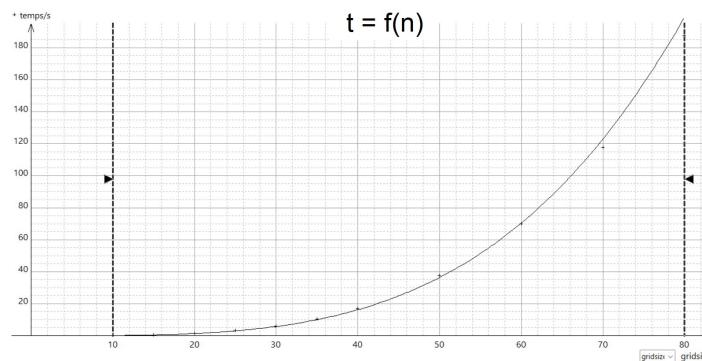


Figure 2.4 Courbe représentant le temps d'exécution du programme en fonction de la taille de la map

En traçant  $t = f(n^3)$  nous avons remarqué que le temps d'exécution était proportionnel au cube de la taille de la grille ; la complexité du programme est donc en  $O(n^3)$ .

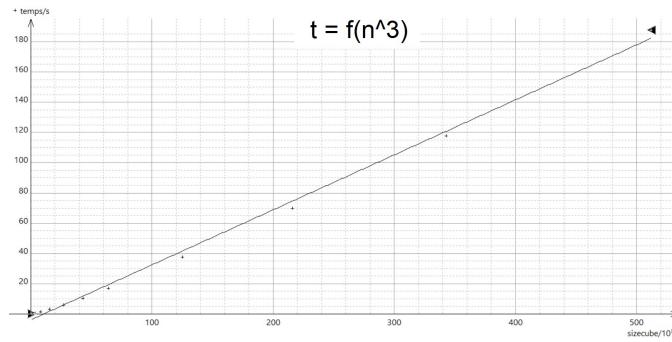


Figure 2.5 Courbe représentant le temps d'exécution du programme en fonction du cube de la taille de la map

Les modélisations ont été effectuées dans un cadre simple : un espace presque vide et avec très peu d'obstacles ; autrement dit des situations moins complexes pour le programme par rapport aux cas où nous aurions à gérer un environnement entier. Cependant, même dans ce cadre limité, ces modèles nous ont aidé à comprendre à quel point les calculs pouvaient durer. Par exemple, pour une grille de  $150 \times 150 \times 150$  (i.e. la taille effective l'environnement 3D que nous allions modéliser), il nous faudrait plus de 30 minutes pour chaque génération de trajectoire.

## 2.5 Lissage de la trajectoire

Les trajectoires générées par l'algorithme A\* sont des successions de droites, ce qui n'est pas suivable par un drone de manière réaliste. Nous avons donc essayé de lisser la trajectoire en utilisant le polynôme d'interpolation de Lagrange dans le but d'arrondir le trajet du drone dans les virages. Cette méthode permet, à partir d'une liste de points du plan, de générer une courbe polynomiale passant par lesdits points.

En voici un exemple [en annexe](#) (Fig. 10). Nous avons donc commencé par tester l'efficacité de cette méthode sur de nombreux chemins. Les trajectoires ainsi calculées introduisaient des déviations par rapport au trajet original qui pouvaient induire des allongements importants de la trajectoire, notamment lors de l'approximation sur un grand nombre de valeurs.

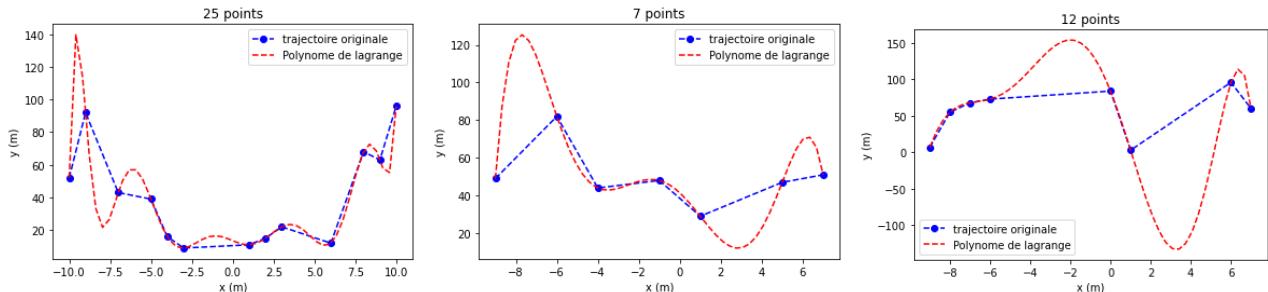


Figure 2.6 Graphes mettant en valeur l'allongement produit par le lissage de la trajectoire à l'aide du polynôme d'interpolation de Lagrange

Utiliser trois points semblait être le meilleur compromis, cependant même dans ce cas les allongements demeuraient non négligeables.

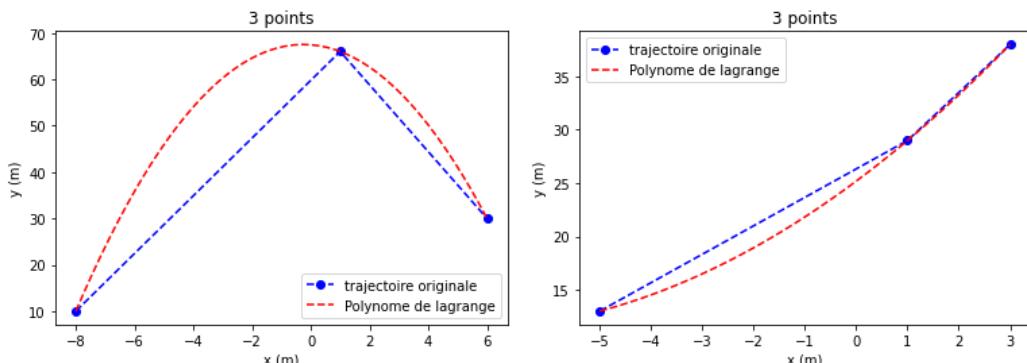


Figure 2.7 Graphes mettant en valeur l'allongement produit par le lissage de la trajectoire sur 3 noeuds à l'aide du polynôme d'interpolation de Lagrange

D'autre part ces détours pouvaient pousser le drone à entrer en collision avec son environnement. Finalement, après avoir effectué les essais en 2D, nous pensons que ces alternatives en 3D de ce type d'interpolation polynomiale aient tendance à mal se comporter.

Suite à cette étude, nous n'avons pas retenu cette solution.

## 3 Deuxième partie : Modélisation 3D

### 3.1 Fonctionnement de l'éditeur 3D

Afin de modéliser l'environnement 3D, nous avons travaillé avec 3D World Editor qui est relié à Simulink. Il s'agit donc de présenter le fonctionnement de cet outil et la manière dont nous avons créé l'environnement.

#### Définitions

**Node Transform** : Element importé sur l'éditeur. Celà peut être une forme géométrique, un décor ou encore un objet provenant de la bibliothèque.

**Node Children** : Sous-élément du Transform. Celà permet d'ajouter une couleur, une texture ou encore un autre objet au Transform.

**Workspace** : inventaire de toutes les variables stockées dans Matlab

#### 3.1.1 Crédit d'objets

Pour créer un environnement 3D, il est nécessaire de modéliser individuellement chaque objet. Ceux-ci sont composés de plusieurs formes basiques, telles que des cubes ou des sphères.

Nous avons donc modélisé la base du drone (Fig. 3.1) à partir de pavés droits et les hélices avec des cylindres (afin de représenter le drone en vol) en modifiant sur leurs dimensions (hauteur, longueur, largeur), leur orientation, leur position et leur apparence (couleur, texture, opacité). Nous avons eu l'occasion de manipuler physiquement le drone en question afin d'y associer les bonnes dimensions.

Voici les étapes pour créer un objet :

1. Créer l'objet avec un **node Transform**
2. Ajouter dans "**children**" les différents éléments composant l'objet
3. Modifier l'aspect des composants (l'orientation dans "**rotation**", la taille dans "**scale**", la position dans "**translation**", l'apparence dans "**appearance**")



Figure 3.1 - Modélisation du drone en 3-dimensions

#### 3.1.2 Crédit d'un environnement

Un environnement (Fig. 3.2) est composé de plusieurs objets qui seront placés dans l'espace.

Il est nécessaire de prendre en compte leurs dimensions afin de faire en sorte que l'affichage corresponde aux attentes de la modélisation 3D que cela soit pour un modèle réaliste (pour lequel les proportions sont respectées), ou pour un modèle pour lequel l'accent est mis sur un élément en particulier, comme pour le cas d'un déplacement ou d'une rotation. Il est possible d'afficher une grille ainsi que les axes de l'espace qui vont aider à positionner les objets avec plus de précision.

Afin de visualiser l'environnement correctement, il est important d'ajouter un arrière-plan. Il va aussi permettre de contextualiser l'environnement.

De plus, il est possible de placer des sources de lumière, ce qui permet d'augmenter le réalisme.

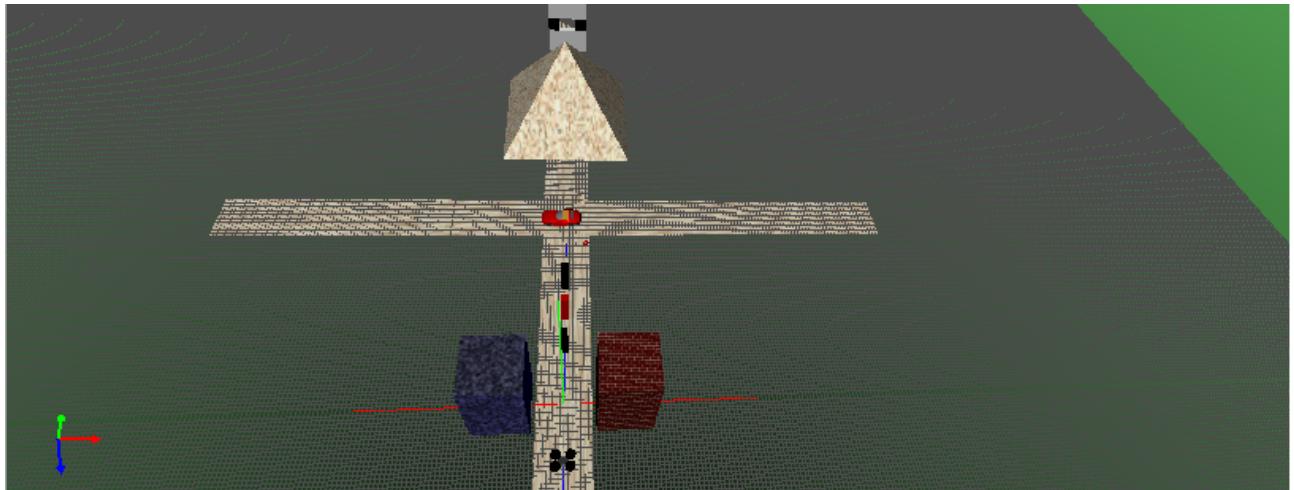


Figure 3.2 - Environnement 3D dans lequel évolue le drone

### 3.1.3 Point de vue

Un point de vue fixé est essentiel pour la visualisation. Bien qu'il soit possible de le modifier manuellement, nous avons privilégié celui du suivi du drone (Fig 3.3).

Afin que la vue suive celui-ci lorsque nous lançons la simulation, nous avons dû créer une caméra placée en retrait par rapport au drone. Le node "viewpoint" est attaché à l'objet du drone (dans "children") afin qu'il suive la même trajectoire lorsque le drone se déplace. Nous avons d'abord défini la vue à la main pour ensuite l'associer au node "viewpoint".

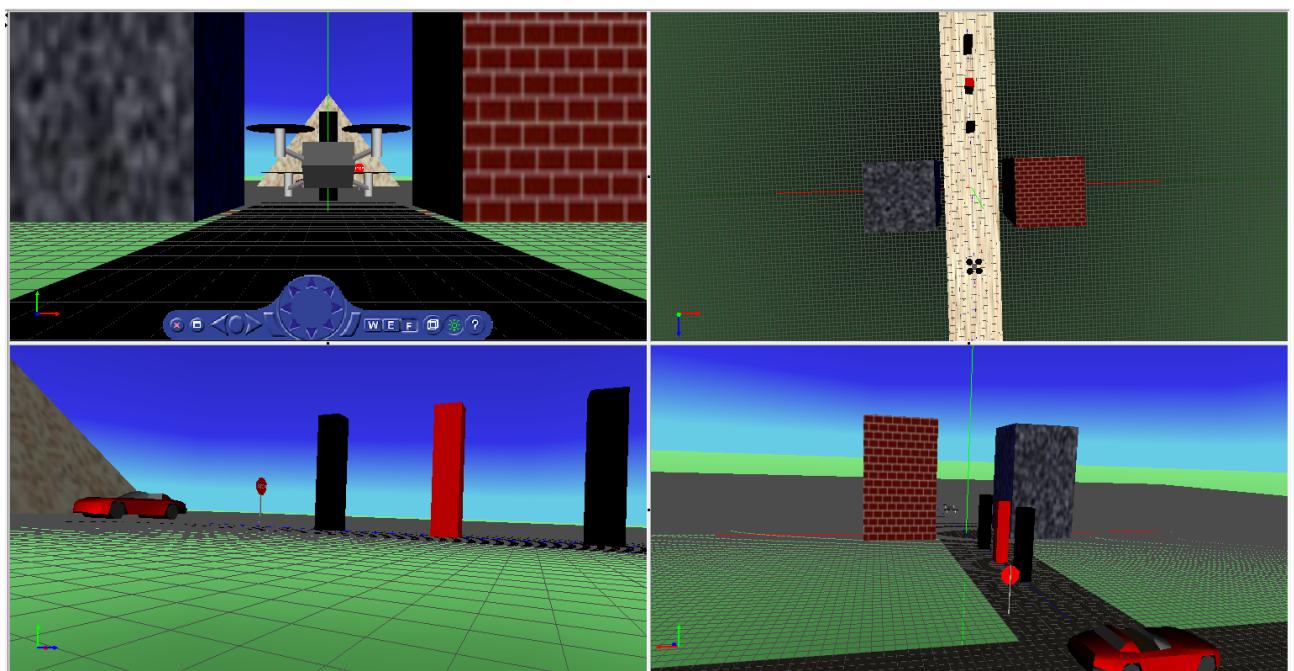


Figure 3.3 - Plusieurs points de vue de l'environnement  
(1) Suivi du drone - (2) Vue de haut - (3)(4) Vue de côté

### 3.1.4 Aspect artistique

L'aspect artistique nous permet d'améliorer le réalisme de la scène. Pour cela, nous avons décidé d'agir sur la texture et l'opacité des obstacles pour créer l'environnement.

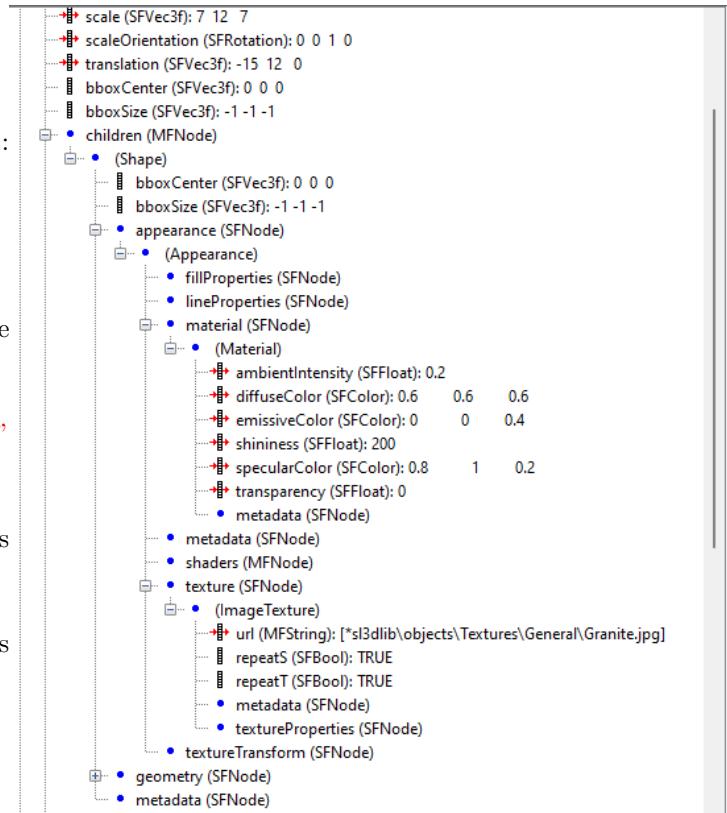
L'éditeur nous permet naturellement d'agir sur la couleur et la transparence des matériaux. Pour mettre une

texture, il suffit de l'importer depuis une librairie de l'éditeur et l'ajouter à l'obstacle en question. Pour la transparence, il faut juste décider de l'indice d'opacité du matériel.

Celle-ci nous permet de disposer des **murs invisibles** (Fig. 11) à l'intérieur du modèle, mais que le drone devra quand même éviter, par exemple pour réaliser un slalom (Fig. 3.4).

Voici les étapes pour ajouter une texture ou un matériel :

1. On ajoute un "**children**" dans l'objet déjà créé.
2. On ouvre un node "**Shape**" du "**children**".
3. Puis on ouvre une sous-node "**appearance**" de "**Shape**".
4. On ajoute une "**texture**" et/ou un "**matériel**" qui sont des sous-nodes de "**appearance**".
5. On peut alors choisir dans le matériel les indices de transparence, les couleurs...
6. Ou bien directement ajouter une texture depuis la bibliothèque de l'éditeur.



## 3.2 Lien entre le modèle et le monde 3D

La prochaine étape a été de relier l'environnement 3D avec l'algorithme qui calcule le trajet que doit prendre le drone, c'est-à-dire en connectant 3D World Editor, Simulink et Matlab.

### 3.2.1 Différence de proportions

Un des problèmes rencontrés était celui des proportions. En effet, pour éviter que le temps de calcul de la trajectoire soit trop important, il nous était impossible de travailler dans un espace trop grand pour la simulation. L'échelle de l'environnement 3D étant trop élevée, nous avons dû appliquer un facteur sur les coordonnées des obstacles. De plus, contrairement aux coordonées du monde 3D, l'algorithme qui détermine la trajectoire ne permet pas au drone d'aller dans des valeurs négatives, nous avons donc ajouté une constante (+10 ici pour l'axe des x) afin de le centrer et que tous les obstacles aient des coordonnées positives.

Ces constantes sont ensuite retranchées lorsque la trajectoire est appliquée à l'environnement 3D.

### 3.2.2 Données du monde 3D

L'identification des caractéristiques de chaque obstacle de l'environnement 3D a été une autre étape du projet.

A l'aide de la grille du monde 3D, nous avons défini manuellement les coordonnées de chaque objet (en prenant 1 carreau = 1 unité).

De plus le bloc **VR Source** sur Simulink permet de récupérer des données du monde 3D. Il suffit d'associer l'environnement de 3D World Editor à ce bloc et d'ensuite sélectionner les différents critères que nous souhaitons récupérer. Le bloc **VR Source** renvoie un signal une fois la simulation lancée. Associé à un bloc **To Workspace**, le signal s'enregistre dans le Workspace sous forme de matrice ou de Timeseries (matrice en fonction du temps ; i.e. un signal).

### 3.2.3 Modifier l'environnement 3D

L'éditeur étant relié à Simulink, il nous est possible de modifier notre environnement depuis celui-ci. En effet le bloc **VR Sink** (Fig. 12) de Simulink nous permet d'agir sur les paramètres des objets créés dans le modèle tels que leurs coordonnées, leurs tailles, leurs couleurs, et ainsi faire des animations en temps réel.

Le fonctionnement du bloc est simple. Tout d'abord, on lui associe un environnement sur lequel on pourra interagir. Ensuite on définit les paramètres sur lesquels on veut agir depuis simulink. Ces paramètres représenteront des entrées auxquelles on pourra connecter un signal pour les modifier.

De plus, le bloc **VR Sink** nous a permis de modifier la vitesse de la simulation en faisant varier l'échelle de temps.

Il nous permet également de visualiser notre simulation.

### 3.2.4 Tracer une courbe

Pour bien visualiser le chemin du drone, on peut représenter la courbe qu'il a suivi à l'aide de Simulink (Fig. 3.4). Le bloc qui nous permet de le faire est **VR Tracer**. Pour la réaliser, il faut d'abord associer l'environnement au bloc, puis envoyer un signal en entrée du bloc. Ce signal doit être une succession de vecteurs dans le temps correspondant aux coordonnées de chaque points. Cependant, ce qui trace la trajectoire dans le workspace sur Matlab est une matrice de taille  $n \times 3$  où les trois colonnes représentent la position d'un point dans l'espace en 3D, et  $n$  le nombre de points constituant la trajectoire.

Or, les blocs **VR** nécessitent une dépendance temporelle que notre matrice n'a pas. Il nous a donc fallu trouver un moyen de faire lire la matrice au bloc en fonction du temps.

Nous avons d'abord essayé d'utiliser le bloc **mux** qui nous permettaient de lire chaque point individuellement en séparant chaque ligne mais nous ne pouvions pas ajouter de rapport de temps.

Une autre solution a été de réaliser une fonction qui lisait chaque ligne à chaque unité de temps. Cette solution fut rapidement abandonnée car elle ne nous renvoyait pas de signal.

Enfin, nous avons trouvé la fonction **Timeseries** (Fig. 13) sur matlab qui nous a permis de lire la matrice en fonction du temps.

D'autre part nous avons rencontré un problème lorsque nous avons tracé la trajectoire, car elle n'était pas dans la bonne direction. Pour cause, lors de la création du drone nous l'avons dirigée selon l'axe des x. Nous avons donc procédé à un **changement de base** (Fig. 14) qui nécessite que l'axe des -y devienne l'axe des z et que l'axe des z devienne l'axe des y.

Nous avons ainsi finalement pu relier la matrice au modèle 3D par **Simulink** (Fig. 15).

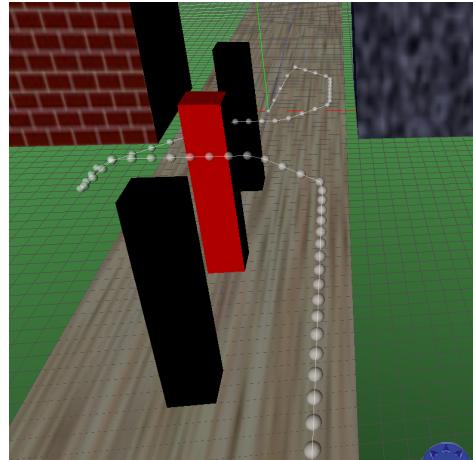


Figure 3.4 - Tracé de la trajectoire d'un slalom

## 4 Conclusion

In conclusion, we have succeeded in understanding the A\* algorithm and in modifying it so that it is able to solve a 3-dimensional problem. We also managed to model obstacles as cuboids and pyramids. We also designed a 3D environment in which the drone evolves. Finally, we were able to link the theoretical trajectory of the drone to the 3D model and thus move it in space.

Furthermore, we honed certain of our skills; such as understanding an algorithm and the capacity of abstraction in 3D. The ability to discover and use multiple software programs (Matlab, Simulink, 3D World Editor and Latex) was essential for this project. This group project also confirmed us that it was really important to communicate altogether in order to achieve good results. Moreover writing this report taught us how to better explain our work synthetically.

Finally, with more time, we could develop certain aspects of the project. For example, we could model other obstacle shapes, such as spheres, or produce a smoother trajectory for the drone. We could also create a more realistic and accurate environment, or focus more deeply on the automatic gathering of object coordinates. In addition, based on our work, a virtual twin of an actual drone evolving in a complex environment could be developed.

Here is a video of the final result we managed to obtain : ["Final trajectory"](#)

## 5 Bibliographie

Nous avons principalement utilisé :

Matlab Onramp ([matlabacademy.mathworks.com](https://matlabacademy.mathworks.com))  
Simulink Onramp ([matlabacademy.mathworks.com](https://matlabacademy.mathworks.com))  
[stackoverflow.com](https://stackoverflow.com)  
[Wikmédia.org](https://fr.wikipedia.org)  
[stackexchange.com](https://fr.stackexchange.com)  
[overleaf.com](https://www.overleaf.com)  
[mathworks.com/help/sl3d/3dworldeditor-app.html](https://mathworks.com/help/sl3d/3dworldeditor-app.html)

## 6 Annexe

```
function dist = distance3D(x1,y1,z1,x2,y2,z2)
%This function calculates the distance between any two cartesian
%coordinates.
% Copyright 2009-2010 The MathWorks, Inc.
dist=sqrt((x1-x2)^2 + (y1-y2)^2 + (z1-z2)^2);
```

FIGURE 1 - "distance3D.m"

```
function n_index = node_index3D(OPEN,xval,yval,zval)
%This function returns the index of the location of a node in the list
%OPEN
%
% Copyright 2009-2010 The MathWorks, Inc.
i=1;
while(OPEN(i,2) ~= xval || OPEN(i,3) ~= yval || OPEN(i,4) ~= zval)
    i=i+1;
end;
n_index=i;
end
```

FIGURE 2 - "NodeIndex3D.m"

```
function i_min = min_fn3D(OPEN,OPEN_COUNT,xTarget,yTarget,zTarget)
%Function to return the Node with minimum fn
% This function takes the list OPEN as its input and returns the index of the
% node that has the least cost
%
% Copyright 2009-2010 The MathWorks, Inc.

temp_array=[];
k=1;
flag=0;
goal_index=0;
for j=1:OPEN_COUNT
    if (OPEN(j,1)==1)
        temp_array(k,:)=[OPEN(j,:);j]; %#ok<AGROW>
        if (OPEN(j,2)==xTarget && OPEN(j,3)==yTarget && OPEN(j,4)==zTarget)
            flag=1;
            goal_index=j;%Store the index of the goal node
        end
        k=k+1;
    end
end%Get all nodes that are on the list open
if flag == 1 % one of the successors is the goal node so send this node
    i_min=goal_index;
end
%Send the index of the smallest node
if size(temp_array) ~= 0
    [min_fn3D,temp_min]=min(temp_array(:,10));%Index of the smallest node in temp array
    i_min=temp_array(temp_min,11);%Index of the smallest node in the OPEN array
else
    i_min=-1;%The temp_array is empty i.e No more paths are available.
end;
```

FIGURE 3 - "min\_fn3D.m"

```

function exp_array=expand_array3D(node_x,node_y,node_z,hn,xTarget,yTarget,zTarget,CLOSED,MAX_X,MAX_Y,MAX_Z)
%Function to return an expanded array
%This function takes a node and returns the expanded list
%of successors,with the calculated fn values.
%The criteria being none of the successors are on the CLOSED list.
%
% Copyright 2009-2010 The MathWorks, Inc.

exp_array=[];
exp_count=1;
c2=size(CLOSED,1);%Number of elements in CLOSED including the zeros
for k= 1:-1:-1
    for j= 1:-1:-1
        for p=1:-1:-1
            if (k~=j || k==0 || j~=p || j==0) %The node itself is not its successor
                s_x = node_x*k;
                s_y = node_y*j;
                s_z = node_z*p;
                if((s_x >0 && s_x <=MAX_X) && (s_y >0 && s_y <=MAX_Y) && (s_z >0 && s_z <=MAX_Z))%node within array bound
                    flag=1;
                    for c1=1:c2
                        if(s_x == CLOSED(c1,1) && s_y == CLOSED(c1,2) && s_z == CLOSED(c1,3))
                            flag=0;
                        end
                    end%End of for loop to check if a successor is on closed list.
                    if (flag == 1)
                        exp_array(exp_count,1) = s_x;
                        exp_array(exp_count,2) = s_y;
                        exp_array(exp_count,3) = s_z;
                        exp_array(exp_count,4) = hn+distance3D(node_x,node_y,node_z,s_x,s_y,s_z);%cost of travelling to node
                        exp_array(exp_count,5) = distance3D(xTarget,yTarget,zTarget,s_x,s_y,s_z);%distance between node and goal
                        exp_array(exp_count,6) = exp_array(exp_count,4)+exp_array(exp_count,5);%fn
                        exp_count=exp_count+1;
                    end%Populate the exp_array list!!!
                end% End of node within array bound
            end%End of if node is not its own successor loop
        end
    end%End of j for loop
end%End of k for loop

```

FIGURE 4 - "Expand\_array3D.m"

```

function new_row = insert_open3D(xval,yval,zval,parent_xval,parent_yval,parent_zval,hn,gn,fn)
%Function to Populate the OPEN LIST
%OPEN LIST FORMAT
%-----%
%IS ON LIST 1/0 |X val |Y val |Z val |Parent X val |Parent Y val |Parent Z val |h(n) |g(n) |f(n) |
%-----%
%
% Copyright 2009-2010 The MathWorks, Inc.
new_row=[1,8];
new_row(1,1)=1;
new_row(1,2)=xval;
new_row(1,3)=yval;
new_row(1,4)=zval;
new_row(1,5)=parent_xval;
new_row(1,6)=parent_yval;
new_row(1,7)=parent_zval;
new_row(1,8)=hn;
new_row(1,9)=gn;
new_row(1,10)=fn;

end

```

FIGURE 5 - "insert\_open3D.m"

```

while((xNode ~= xTarget || yNode ~= yTarget || zNode ~= zTarget) && NoPath == 1)
exp_array=expand_array3D(xNode,yNode,zNode,path_cost,xTarget,yTarget,zTarget,CLOSED,MAX_X,MAX_Y,MAX_Z);
exp_count=size(exp_array,1);
%UPDATE LIST OPEN WITH THE SUCCESSOR NODES
%OPEN LIST FORMAT
%-----
%IS ON LIST 1/0 |X val |Y val |Z val |Parent X val |Parent Y val |Parent Z val |h(n) |g(n)|f(n) |
%-----
%EXPANDED ARRAY FORMAT
%-----
%|X val |Y val |Z val ||h(n) |g(n) |f(n) |
%-----
for i=1:exp_count
flag=0;
for j=1:OPEN_COUNT
if(exp_array(i,1) == OPEN(j,1) && exp_array(i,2) == OPEN(j,3) && exp_array(i,3) == OPEN(j,4))
OPEN(j,1)=min(OPEN(j,1),exp_array(i,6)); %#ok<*SAGROW>
if OPEN(j,10)== exp_array(i,6)
%UPDATE PARENTS,gn,hn
OPEN(j,5)=xNode;
OPEN(j,6)=yNode;
OPEN(j,7)=zNode;
OPEN(j,8)=exp_array(i,4);
OPEN(j,9)=exp_array(i,5);
end%End of minimum fn check
flag=1;
end%End of node check
if flag == 1
break;
end%End of j for
if flag == 0
OPEN_COUNT = OPEN_COUNT+1;
OPEN(OPEN_COUNT,:)=insert_open3D(exp_array(i,1),exp_array(i,2),exp_array(i,3),xNode,yNode,zNode,exp_array(i,4),exp_array(i,5),exp_array(i,6));
end%End of insert new element into the OPEN list
end%End of i for
%%%%%%%%%%%%%
%END OF WHILE LOOP
%%%%%%%%%%%%%
%Find out the node with the smallest fn
index_min_node = min_fn3D(OPEN,OPEN_COUNT,xTarget,yTarget,zTarget);
if (index_min_node ~= -1)
%Set xNode and yNode to the node with minimum fn
xNode=OPEN(index_min_node,2);
yNode=OPEN(index_min_node,3);
zNode=OPEN(index_min_node,4);
path_cost=OPEN(index_min_node,8);%Update the cost of reaching the parent node
%Move the Node to list CLOSED
CLOSED_COUNT=CLOSED_COUNT+1;
CLOSED(CLOSED_COUNT,1)=xNode;
CLOSED(CLOSED_COUNT,2)=yNode;
CLOSED(CLOSED_COUNT,3)=zNode;
OPEN(index_min_node,1)=0;
else
%No path exists to the Target!!
NoPath=0;%Exits the loop!
end%End of index_min_node check
end%End of While Loop

```

FIGURE 6 - Algorithme A\*

```

i=size(CLOSED,1);
Optimal_path=[];
xval=CLOSED(i,1);
yval=CLOSED(i,2);
zval=CLOSED(i,3);
i=1;
Optimal_path(i,1)=xval;
Optimal_path(i,2)=yval;
Optimal_path(i,3)=zval;
i=i+1;

if ((xval == xTarget) && (yval == yTarget) && (zval == zTarget))
inode=0;
%Traverse OPEN and determine the parent nodes
parent_x=OPEN(node_index3D(OPEN,xval,yval,zval),5);%node_index returns the index of the node
parent_y=OPEN(node_index3D(OPEN,xval,yval,zval),6);
parent_z=OPEN(node_index3D(OPEN,xval,yval,zval),7);
while( parent_x ~= xStart || parent_y ~= ystart || parent_z ~= zStart)
Optimal_path(i,1) = parent_x;
Optimal_path(i,2) = parent_y;
Optimal_path(i,3) = parent_z;
%Get the grandparents:-)
inode=node_index3D(OPEN,parent_x,parent_y,parent_z);
parent_x=OPEN(inode,5);%node_index returns the index of the node
parent_y=OPEN(inode,6);
parent_z=OPEN(inode,7);
i=i+1;
end
j=size(Optimal_path,1);
Optimal_path

```

FIGURE 7 - Programme qui remonte les parents du noeud d'arrivée pour trouver le chemin le plus court

```

function [xcoordObsta,ycoordObsta,zcoordObsta]=coordObsta3D(CoordCube)
%cube=[xinf,xsupp;yinf,ysupp,zinf,zsupp]
xcoordObsta=[];
ycoordObsta=[];
zcoordObsta=[];
for i=CoordCube(1,1):1:CoordCube(1,2)
    for j=CoordCube(2,1):1:CoordCube(2,2)
        for k=CoordCube(3,1):1:CoordCube(3,2)
            xcoordObsta=[xcoordObsta,i];
            ycoordObsta=[ycoordObsta,j];
            zcoordObsta=[zcoordObsta,k];
        end
    end
end

```

FIGURE 8 - "coordosta3D.m"

```

function [x_pyra,y_pyra,z_pyra]=Pyramide3D(Pyramid_coordinates)
% Pyramid_coordinates = [x_base,y_base, z_base,cote_base,pyramid_height]
%rem on doit avoir cote_base >= 2*height
x_pyra=[];
y_pyra=[];
z_pyra=[];
%parameters
height=Pyramid_coordinates(5) ;
b=Pyramid_coordinates(4);
x0=Pyramid_coordinates(1) ;
y0= Pyramid_coordinates(2);
z0=Pyramid_coordinates(3);
xmax = x0+height ;
ymax=y0+height;
increment=floor(b/height) ;%increment
c=0;
for k=z0:z0+height
    c=c+floor(increment/2);
    for i= x0+c:xmax-c
        for j= y0+c:ymax-c
            x_pyra=[x_pyra,i];
            y_pyra=[y_pyra,j];
            z_pyra=[z_pyra,k];
        end
    end
end

```

FIGURE 9 - "Pyramide3D.m"

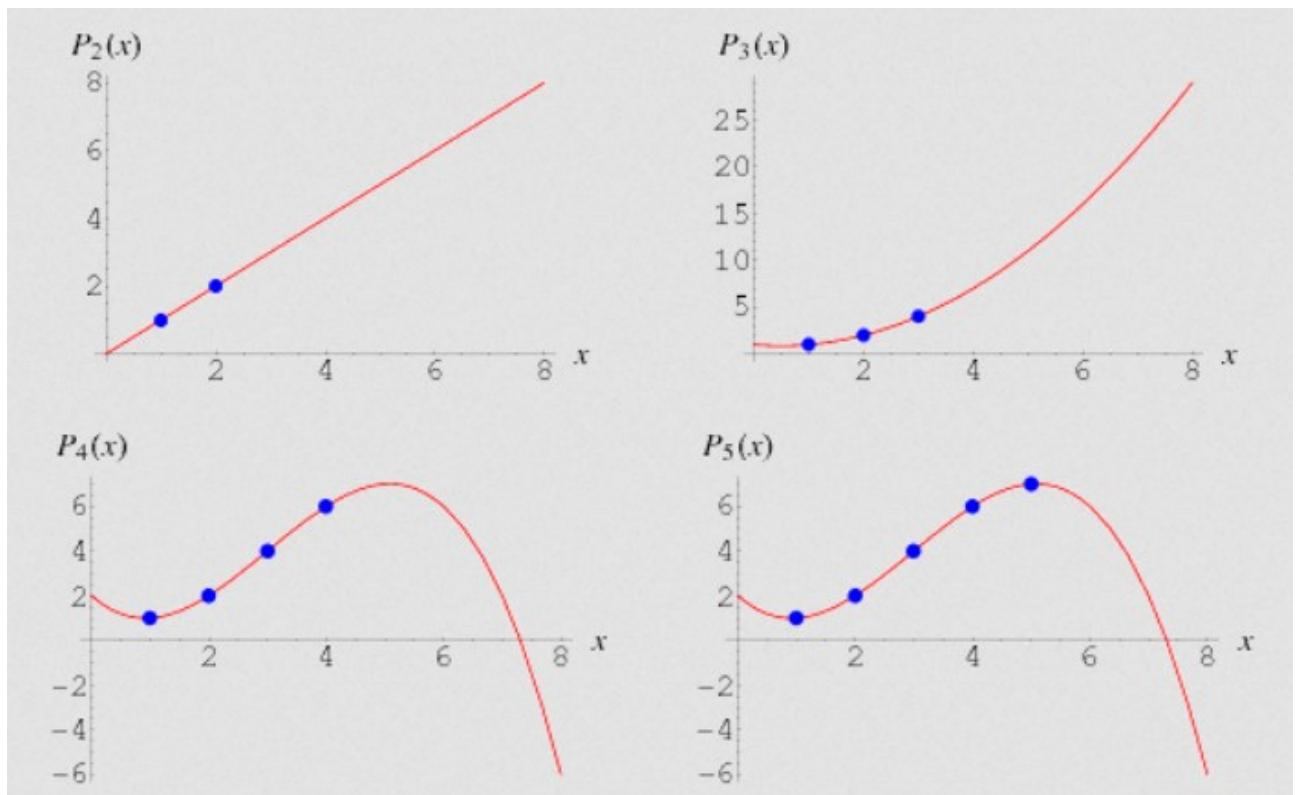


FIGURE 10 - Exemples d'application du polynôme de Lagrange

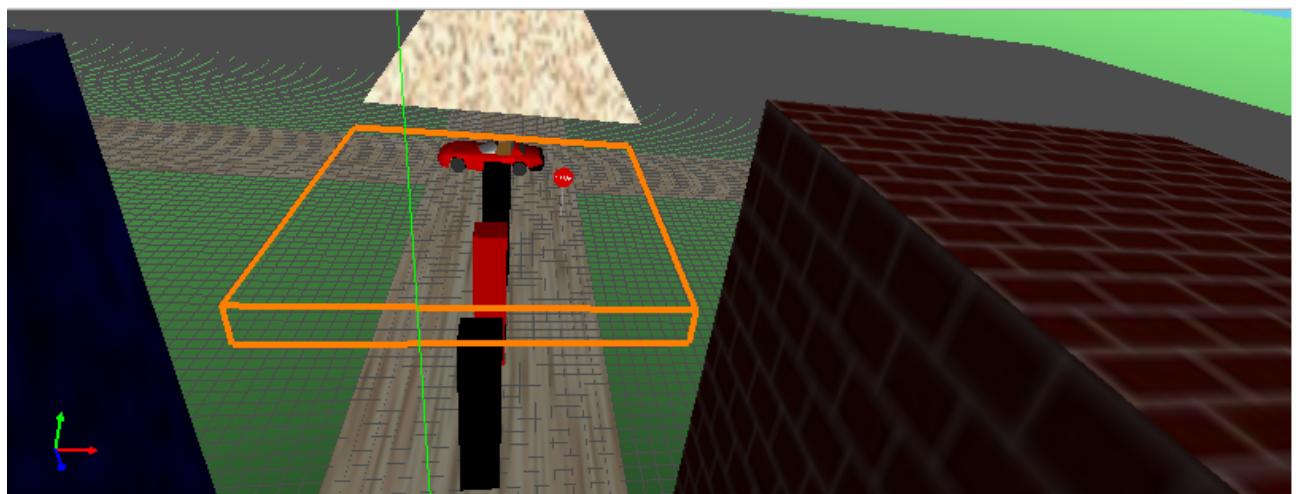


FIGURE 11 - Mur invisible

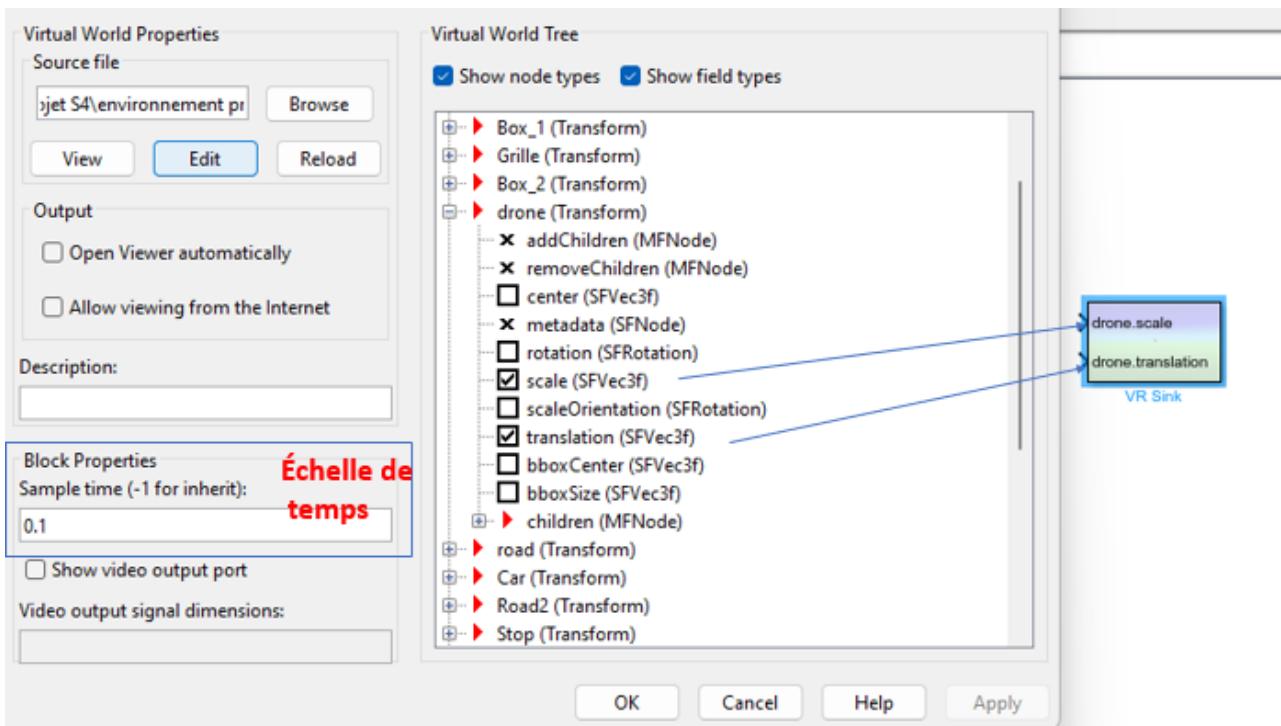


FIGURE 12 - Bloc VR Sink

```

M=cat(1,3*Optimal_path,[10 0 0]*3)
%pour remettre a bonne echelle et on ajoute le point de départ
x=[]
y=[]
z=[]
A=[]
for i=size(M,1):-1:1
%on lit la matrice à l'envers pour que cela soit ordonné pour la lecture des valeurs
x=[x;M(i,1)-30] %pour recentrer la trajectoire
z=[z;-M(i,2)]
y=[y;M(i,3)]
%parce qu'on a tourné le drone donc changement de base
end

A=[x y z]
%matrice de la trajectoire

t=linspace(0,10,size(A,1))
%linspace sépare ti = 0 à tf = 10 en un nombre de valeurs de la taille de la matrice
% t les différents temps
trajectoire=timeseries(A,t)
%timeseries associe un temps à une ligne de la matrice => crée un signal

```

FIGURE 13 - Algorithme qui transforme une matrice en un signal

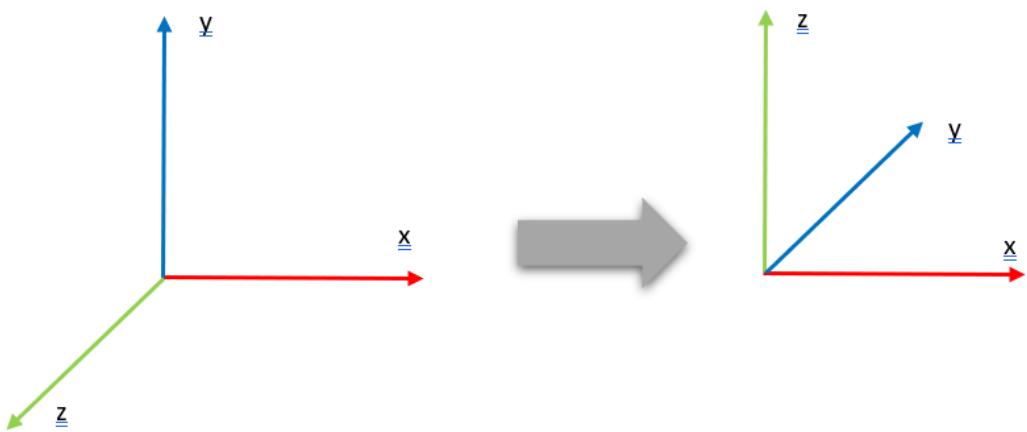


FIGURE 14 - Changement de base du drone

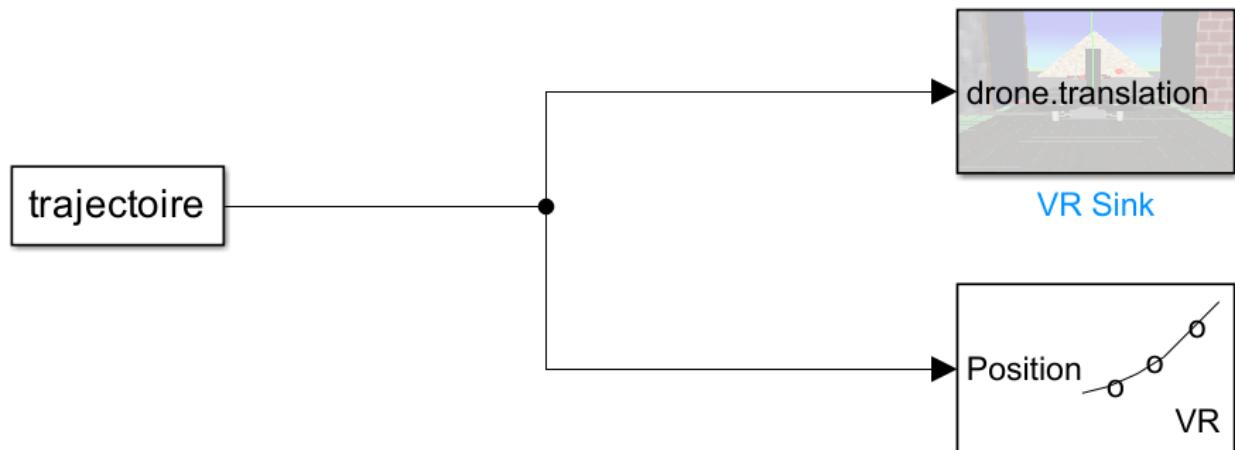


FIGURE 15 - Programme Simulink

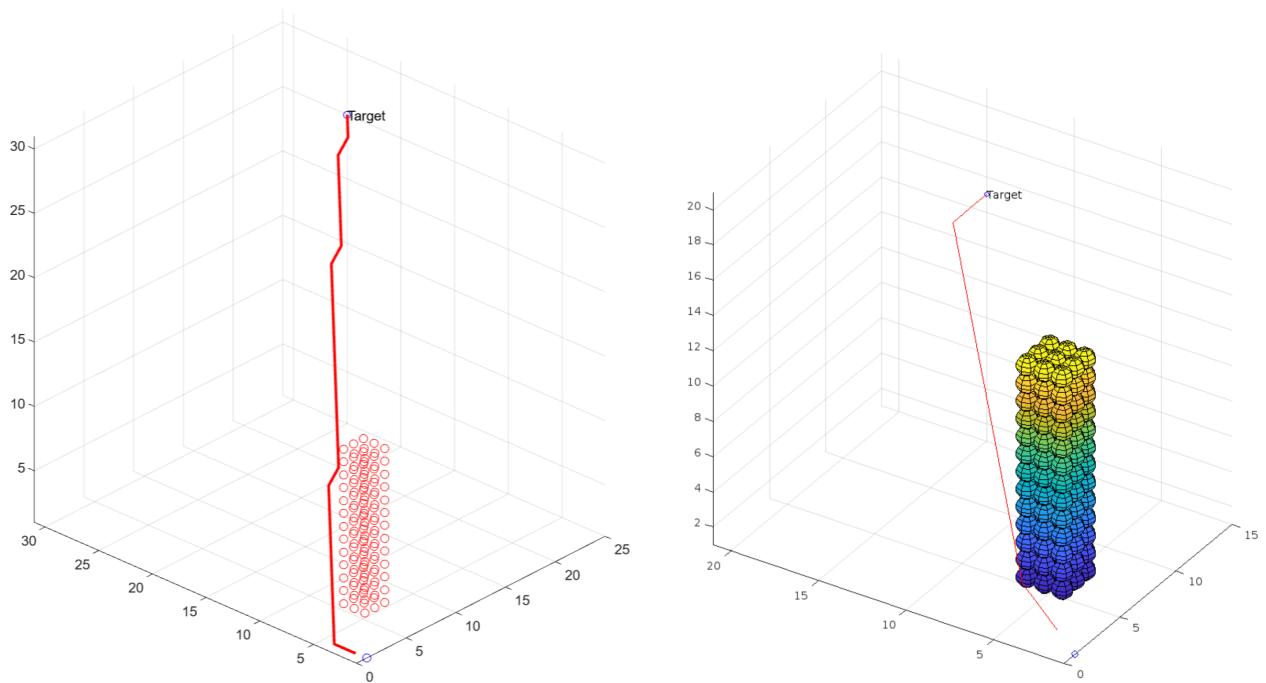


FIGURE 16 - Comparatif des visuels pour la modélisation