UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES

Département d'informatique
Université de Genève
Travail de bachelor

DroneProgrammer

Application iOS
github: mathiasTonini/droneProgrammer

Étudiant:
Sungurtekin Deniz

Professeur:
Buchs Didier

semestre de printemps 2020

Table des matières

Chapter 1: Introduction

1.1 Introduction

Ce travail de Bachelor a pour but d'implémenter une application iOS pour iPad permettant de faciliter le pilotage et la planification de vol pour le drone bebop 2 de chez Parrot. Pour réaliser cette tâche, un ancien projet nous a été fourni. Ce projet possède les installations nécessaires afin de pouvoir utiliser les outils de Parrot, essentielles pour le maniement du drone à travers un programme informatique, ainsi que l'implémentation d'un interpréteur définissant un langage pour ce drone.
Nous verrons dans un premier temps, la méthodologie suivie afin de visualiser, comprendre et analyser notre application avant de la concevoir. Même si cette première analyse n'est pas forcément toujours exacte, elle joue un rôle majeur dans l'aboutissement et la compréhension de ce projet puisqu'elle donne une idée simple du fonctionnement global. C'est donc pour ces raisons que nous nous pencherons sur l'apport indispensable que nous a fournis l'ancien projet. Ensuite nous présenterons un premier storyboard permettant de se familiariser avec l'application et visualiserons une architecture pour donner une première idée de son fonctionnement.
C'est seulement après avoir compris ce modèle que nous examinerons en détail le développement de l'application à travers ses fonctionnalités majeures et les composantes qui la constitue. Finalement, nous aborderons d'un ton plus technique le fonctionnement de nos pages en clarifiant les différentes technologies utilisées.

Chapter 2: Planification

2.1 Ancien projet

Dans un premier temps, avant toute réflexion à propos de notre projet, nous avons dû passer par une phase d'apprentissage et de compréhension vis à vis de ce premier travail, afin de pouvoir mieux comprendre le fonctionnement du drone et la manière dont nous pouvions interagir avec. Cela nous a permis de déterminer les parties que nous voudrions intégrées à notre propre système tout en nous refamiliarisant avec le langage swift et l'environnement de développement que représente Xcode. Après plusieurs tests sur les fonctionnalités déjà implémentées, nous avons donc décider de reprendre le même procédé pour définir le langage de notre drone et la manière dont celle-ci est traduite pour communiquer avec la librairie fournie par Parrot afin de contrôler le drone. *(Plus de détails sous la section développement et dans la documentation en annexe)*

Maîtrisant les fondamentaux, il nous a donc fallu passer à l'analyse et la planification de notre projet en commençant par la grammaire de notre langage.

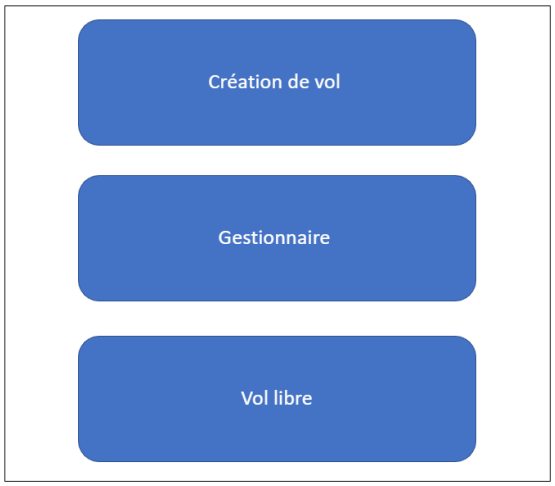
2.2 Grammaire

Dans le but de donner du sens à notre langage, il est nécessaire de lui définir les expressions qui lui appartiennent afin d'exprimer les contraintes élémentaires amenées par la planification d'un vol. Voici donc une grammaire simple sous la forme EBNF décrivant les séquences de mouvement possibles lors d'un vol:

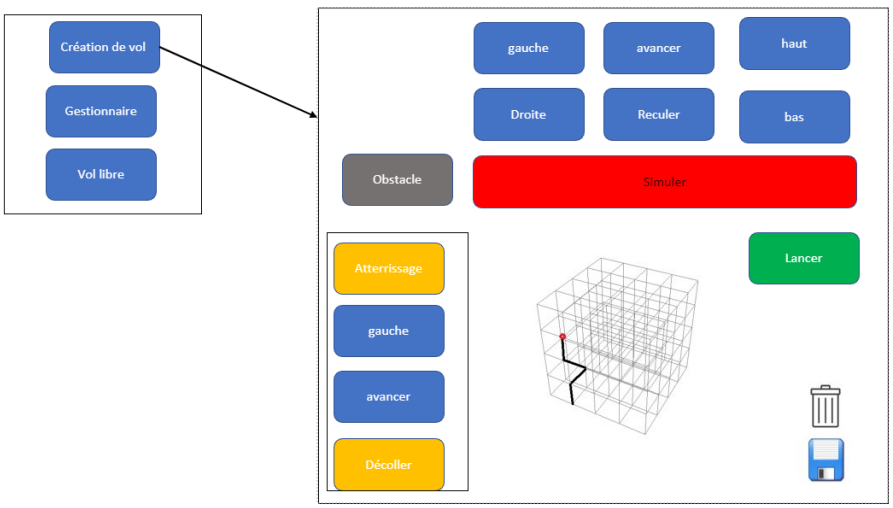
<expr> -> décollage + <terme>
<terme> -> <commande> + <terme> | fin
<commande> -> droite | gauche | avancer | reculer | monter | descendre
<fin> -> atterrissage

2.3 Storyboard

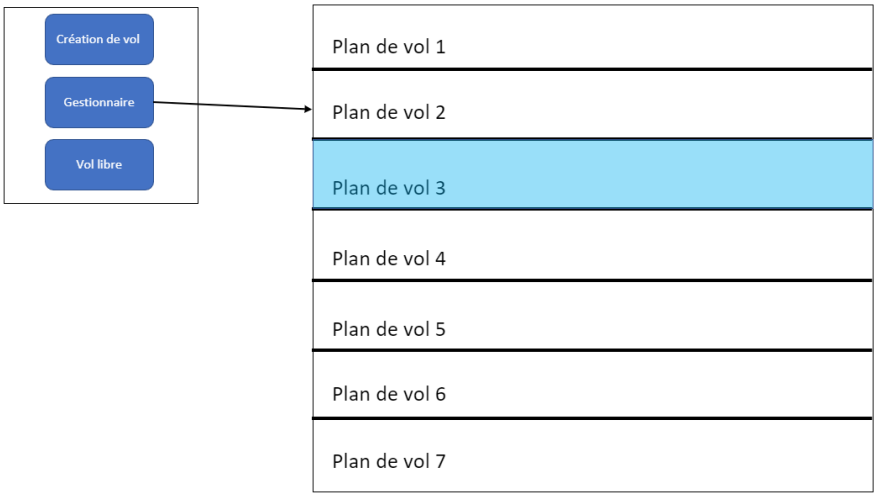
Pour avoir une première idée claire de notre futur application, nous avons esquisser un storyboard simple permettant de visualiser son utilisation à travers les différentes fonctionnalités que nous voulions implémenter. Il est constitué d'un menu permettant d'accéder aux trois modes principaux: la création de vol, le gestionnaire et le vol libre.



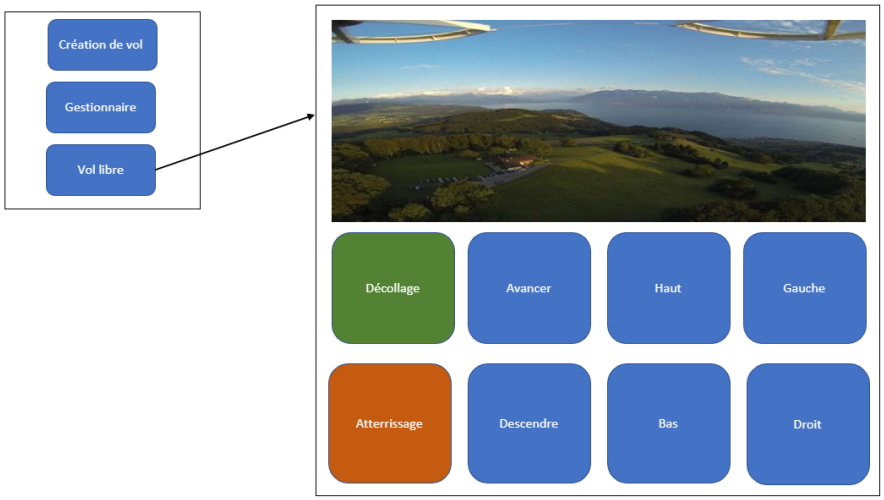
La création de vol permet à l'utilisateur de définir une liste de commande, d'obstacle et d'objectif, puis de simuler le vol afin de vérifier si celui-ci satisfait toutes les conditions nécessaires. Le système doit donc s'assurer que la liste de commande soit conforme à la grammaire définie, que le drone ne touche aucun obstacle et remplisse tous les objectifs donnés.



Le gestionnaire sert à afficher les plans de vols sauvegardés puis de les charger dans la création de vol afin de les modifier ou de les compléter.



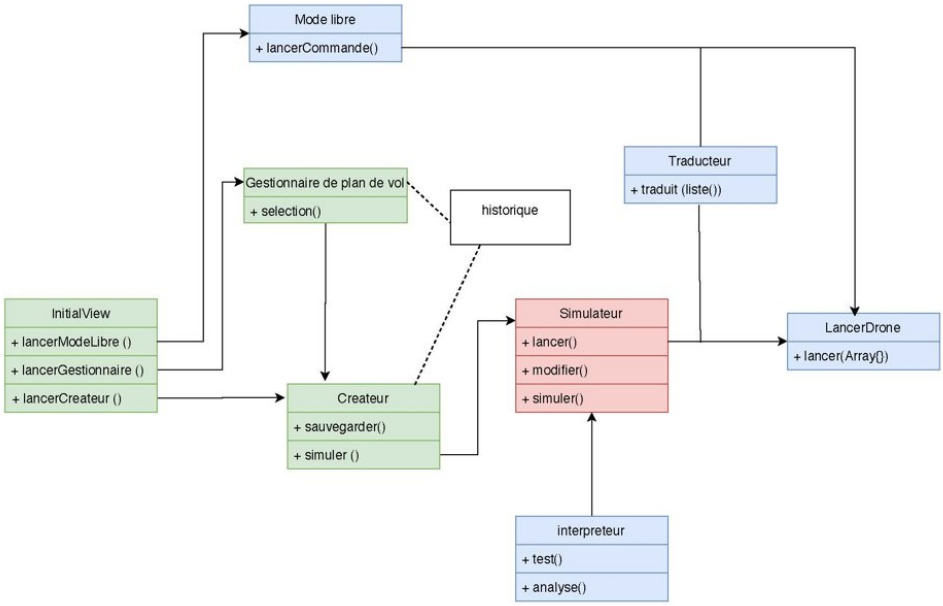
Le vol libre offre la possibilité à l'utilisateur de se déplacer librement tout en ayant un aperçu sur la caméra du drone et sera en mesure de prendre des photos.



2.4 Architecture

Après avoir décidé de la forme de notre application, il est nécessaire de se pencher sur son fond et d'assurer la compréhension des composantes qui construiront notre système afin de minimiser les potentiels erreurs de développement. Pour cela, nous

avons divisé notre système et défini les rôles et fonctions de chaque composante. Cela nous a permis de mettre en évidence les dépendances existantes et de structurer notre méthode de développement afin d'obtenir un projet qui soit le plus clair possible pour d'éventuelle modification par un autre développeur. Voici donc l'architecture de notre application.



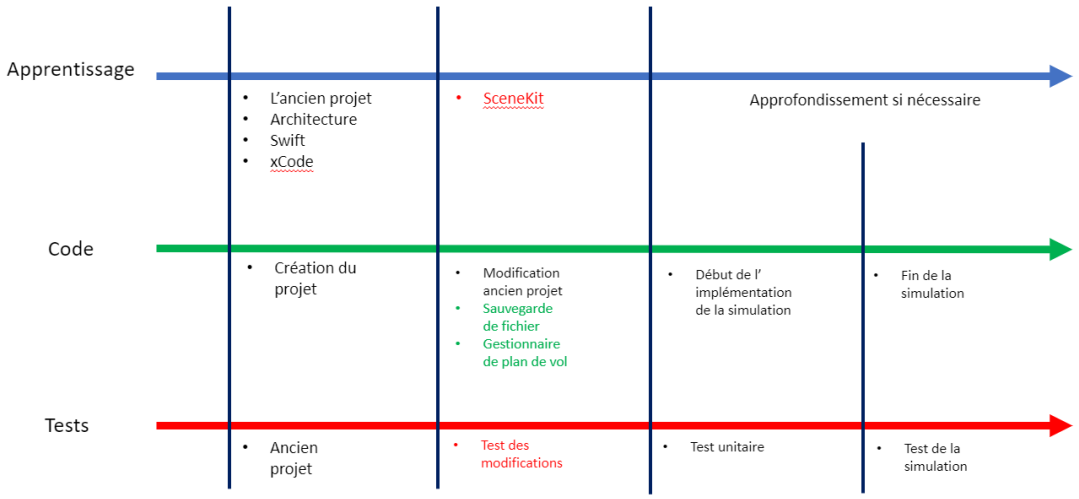
Comme précisé dans le storyboard, l'application commence sur une vue permettant de choisir l'une des trois fonctionnalités, le mode libre donne un contrôle immédiat sur le drone en retournant un entier au moment du clique sur une commande qui selon sa valeur fait appel à la fonction correspondante de la librairie Parrot. (Rôles du traducteur mais avec un entier au lieu d'une liste d'entier)

Le gestionnaire affiche depuis un historique les sauvegardes et permet leur sélection, ce qui mènera l'utilisateur dans le mode de création de vol avec les valeurs sauvegardées.

A partir du mode de création, il est possible de sauvegarder dans l'historique les valeurs crées/modifiées et de simuler le vol sur une autre vue. Cette nouvelle page vérifie la liste de commande à l'aide d'un interpréteur et simule le vol. Si toutes les conditions sont remplies, alors il est possible de lancer la liste de commande sur le drone, celle-ci sera convertie en une liste d'entier puis lue dans le traducteur afin de retourner au drone les fonctions correspondantes à chacun de ses éléments.

2.5 Planning

Avec tous ces éléments définis, il nous est maintenant possible d'esquisser un planning, nous permettant ainsi d'avoir une idée claire sur nos objectifs. Il va de soi que ce planning n'a pas pu être complètement respecté mais il a été indispensable pour ordonner notre méthode de développement et à diviser nos tâches en un ensemble de petits objectifs à satisfaire. Ce planning est partagé en trois sections, l'apprentissage qui spécifie les technologies à étudier pour implémenter les fonctionnalités désirées, le code qui détermine les tâches implémentées et les tests qui désignent la/les composante(s) testée(s).

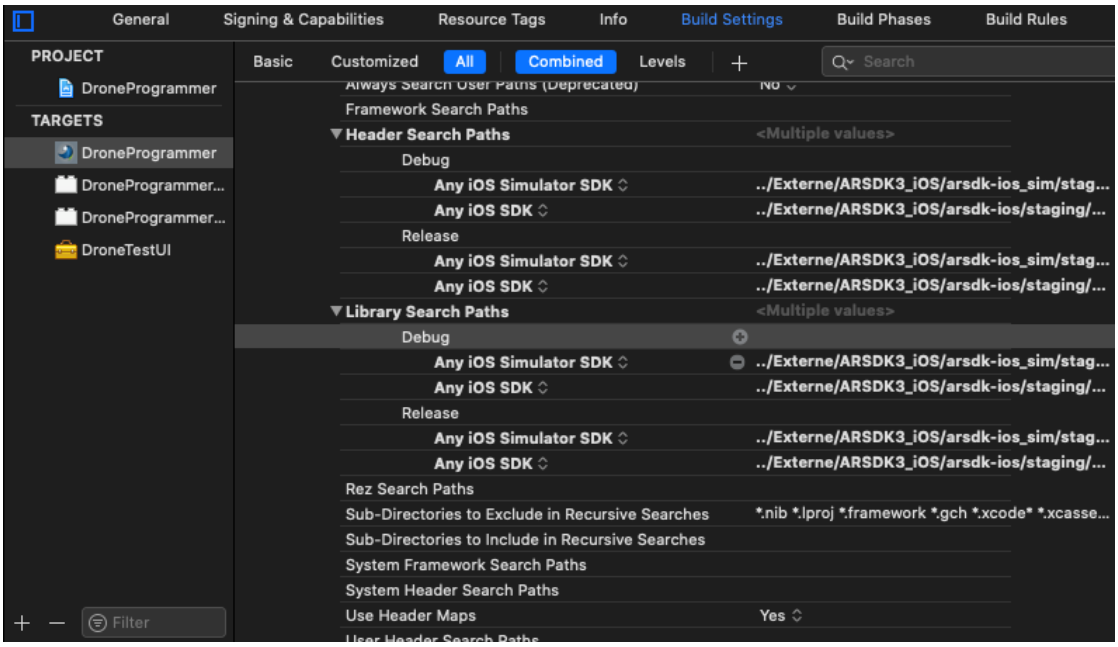


Ce planning a été mis à jour chaque deux semaines en spécifiant sur la deuxième colonne ce qui a été fait les deux semaines précédentes. Les éléments en rouge marquent les tâches qui n'ont pas pu être effectués, celles en verte les tâches terminées et en noir les éléments qui nécessite une éventuelle modification. La première colonne affiche le travail fait précédemment et les deux dernières donnent les objectifs pour le mois à venir.

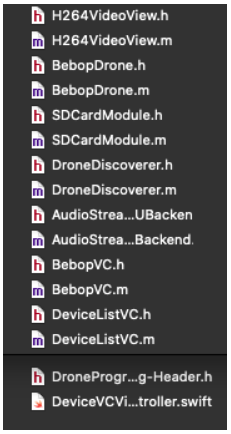
Chapter 3: Développement

3.1 Création du projet

Dans un premier temps, il nous a fallu créer une application sur Xcode et mettre en place les composantes qui nous permettent de travailler dans un environnement de programmation efficace. D'abord, nous avons installé le dossier "ARSDK3IOS" nous permettant de simuler le lancement de l'application sur un iPad après chaque build. Pour cela, il suffit de préciser le 'Library Search Paths' et le 'Header Search Paths' dans le "Build settings" du projet pour qu'il aille chercher le simulateur.

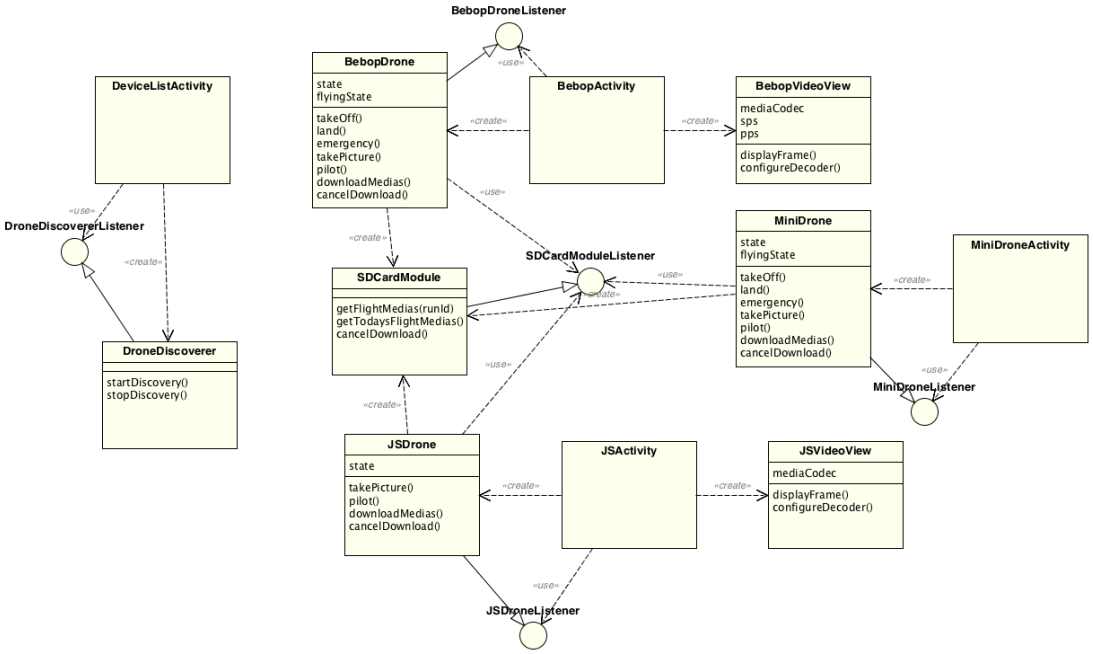


Ensuite, nous avons intégré les fichiers indispensables pour l'utilisation du drone de l'ancien projet dans le nôtre. Ils sont nécessaires pour définir la classe 'bebopDrone' qui permet la manipulation du drone à travers notre code.



Ces fichiers Objectif-C, importent des fonctions depuis le dossier "Samples" pour construire la classe 'bebopDrone'. A noté qu'il est nécessaire de spécifier au compilateur swift un "Objective-C Bridging Header" afin de permettre l'importation de header écrit en Objective-C.

Voici maintenant l'architecture de "Samples" disponible sur le site officiel:

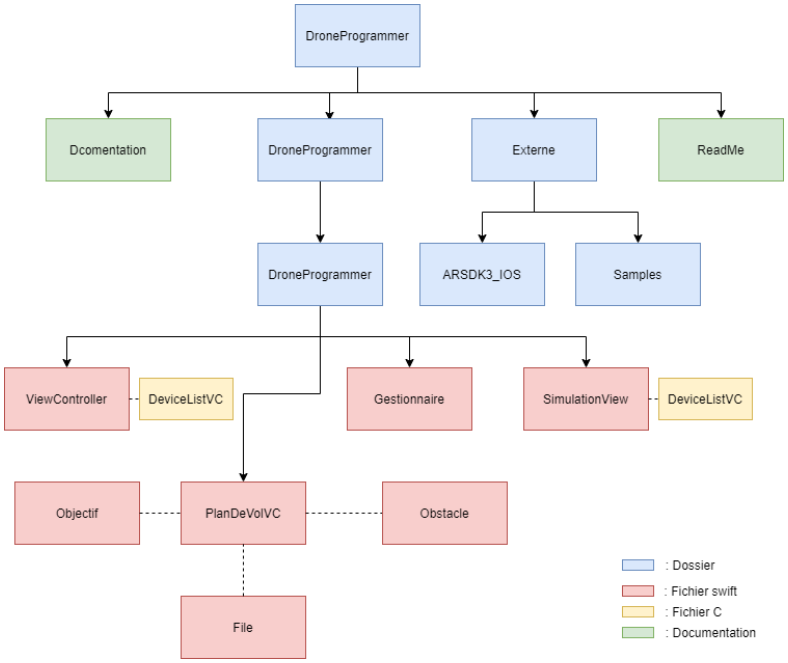


(Lien de la documentation complète de Parrot: Cliquez Ici)

Il est également important de constater que les fichiers "FightPlanVc.swift" et "Interpreter.swift" sont deux fichiers de l'ancien projet qui ne sont pas appelé lors

d'un build mais qui nous ont servis d'exemple pour le maniement de la classe "bebobDrone" et à l'élaboration de notre propre création de vol et interpréteur. Tous les autres fichiers sont automatiquement générés par xCode lors de la création de l'application.

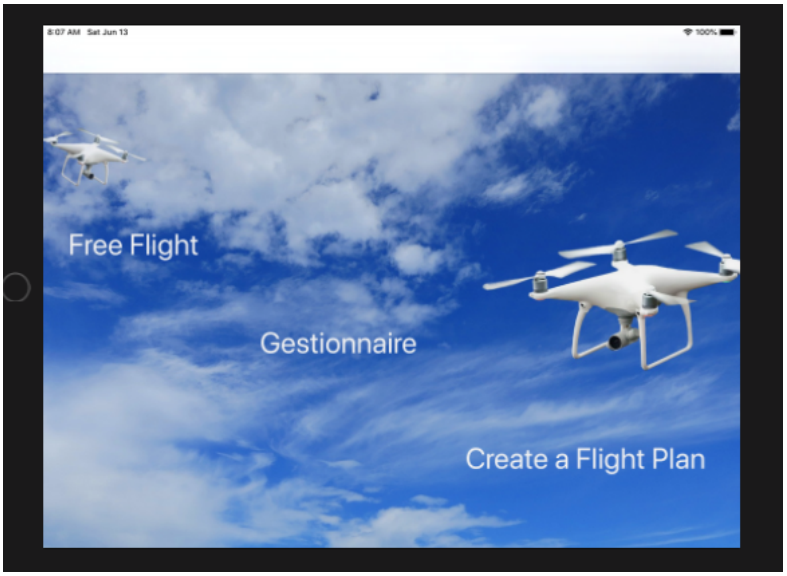
Voici maintenant l'arborescence de notre projet pour plus simplement comprendre sa construction:



L'implémentation de l'application se trouve sous le dossier "DroneProgrammer" et se compose de quatre fichiers swift principaux pour chacune des vues disponibles.

Ces vues sont toutes atteignables depuis le menu principal, celui-ci est définit par le fichier "SelectionVC.swift", cependant il ne contient quasiment aucun code mis à part l'initialisation de la page, puisque toute son implémentation se fait via le "Main.storyboard" qui fournit des outils de haut niveau suffisant pour créer des boutons, des segues et définir une image en fond sur une page.

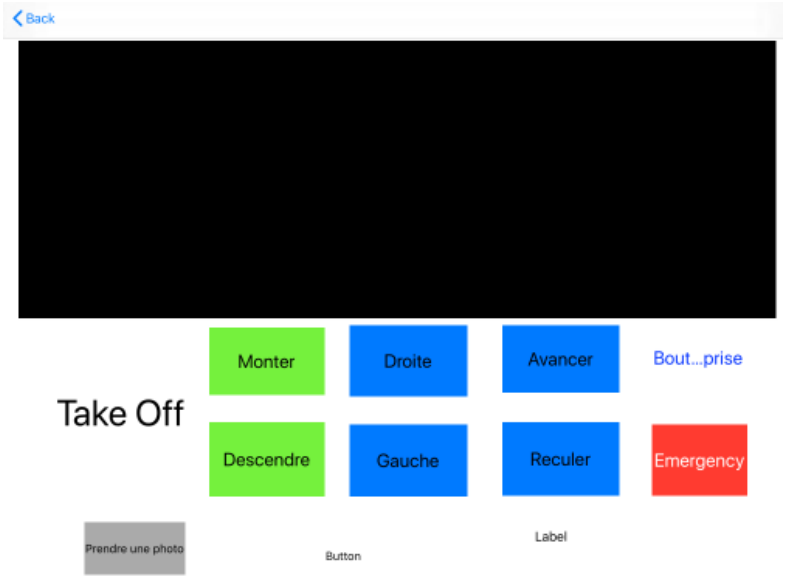
Le menu se présente comme ceci:



3.2 Vol libre

En choisissant "Free Flight" l'utilisateur est conduit sur une page intermédiaire lui permettant de sélectionner le drone auquel il veut se connecter. Le ViewController.swift appelle les fichiers DeviceListVC2.m et DeviceListVC2.h contenue dans le dossier "Samples" afin d'afficher les réseaux disponibles dans une tableView et d'établir la connexion Wi-Fi au drone. Une fois le drone sélectionné, l'utilisateur est amené sur la vue permettant le maniement du drone. Cette page est constituée d'une zone diffusant la vidéo prise par la caméra du drone, de bouton directionnel, d'un bouton permettant la prise de photo puis d'un label spécifiant la batterie restante.
En ce qui concerne la vidéo, une fois que les importations au dossier "Samples" sont établies, il suffit de définir une variable liée à une vue dans le storybord avec le type "H264VideoView" donné par le fichier du même nom, puis d'utiliser deux méthodes, configureDecoder() et displayFrame() pour afficher la vidéo.
Pour les boutons directionnels, il faut déclarer un bouton dans le storyboard puis appeler la méthode correspondante de la classe "bebopDrone" lors d'un clique. Les méthodes primordiales sont takeOff() qui permet le décollage du drone, land() son atterrissage (à utiliser en cas d'urgence, car il est possible de simplement descendre), setGaz(Integer) qui prend un entier en argument, si celui-ci est positif le drone prendra de l'altitude si non il descendra. Le même raisonnement est suivi pour les méthodes setPitch(Integer)/setRoll(Integer)/setYaw(Integer) qui avance/se déplace latéralement à droite/tourne à droite si l'entier est positif. Cependant, les rotations ne sont pas effectives sur notre application, leur fonction est implémentée

mais elles ne sont pas liées à des boutons sur le storyboard.
A propos de la prise de photo, il suffit d'appeler la fonction takePicture() de la classe "bebopDrone" lorsque le bouton est cliqué. Néanmoins, c'est dans le téléchargement de la photo et de son affichage que réside la complexité de cette tâche. Faute de temps, nous avons pu uniquement réaliser le téléchargement de la photo sans pouvoir l'afficher. Par conséquent, nous avons mis en commentaire ces fonctionnalités en espérant qu'elle puisse être profitable à un futur groupe d'étudiant. La dernière spécificité de cette page est celle d'informer l'utilisateur de la batterie restante sur le drone. Pour cela, nous appelons une méthode de bebopDrone qui met à jour le label lorsque le pourcentage de la batterie change.
Voici un aperçu du résultat sans connexion établie:



3.3 Création de vol

Comme vu précédemment, il est possible de créer ses propres trajets en sélectionnant "Create a Flight Plan". Cette fois, il est possible d'atteindre cette vue sans établir de connexion au drone, puisqu'elle permet dans un premier temps uniquement la création et la sauvegarde d'une liste de commande, d'obstacle et d'objectif. Cette page est constituée d'un ensemble de bouton correspondant aux commandes effectuable par le drone, d'un bouton simuler et sauvegarder, de trois entrées permettant d'indiquer les position x, y et z de nos obstacles ou objectifs, et de trois tableView contenant les listes à sauvegarder ou à envoyer au drone.
Lorsqu'une commande est entrée, elle vient s'ajouter à la première tableView, au même moment un entier est ajouté à une variable nommée "listeCommande". Cet

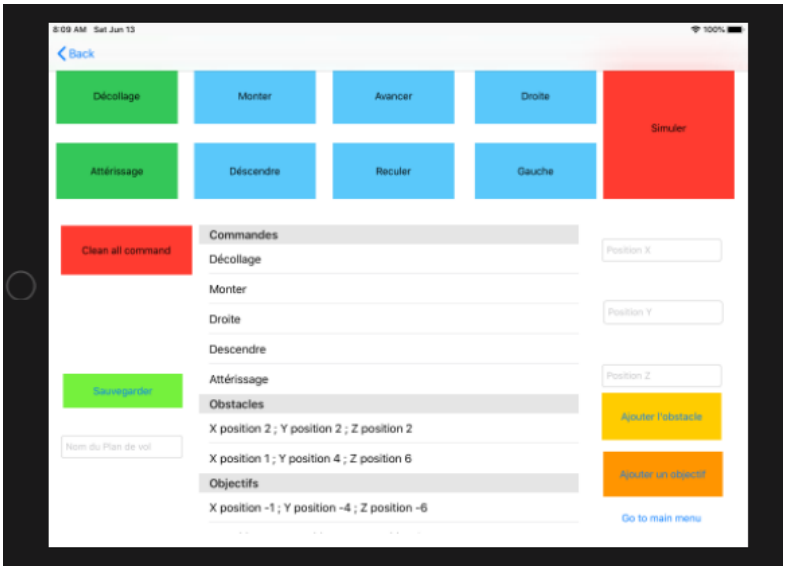
entier correspond à celui lu par le traducteur vu au point **2.4** et permettra dans la partie simulation de faire communiquer au drone la bonne instruction. Voici le lien entre chaque commande et numéro:

- 0: Décollage
- 1: Atterrissage
- 2: Droite
- 3: Gauche
- 4: Avancer
- 5: Reculer
- 6: Monter
- 7: Descendre

En ce qui concerne les objectifs et les obstacles qui sont deux classes conçues de la même manière et définies dans Objectif.swift et Obstacle.swift, elles représentent des objets possédant trois attributs: une position en X, en Y et en Z qui permettront leur placement dans un cube 3D. Tout comme la liste de commande, leur tableView respectif se met à jour lorsqu'on appuie sur le bouton "ajouter un obstacle/objectif". Une fois les valeurs entrées, il est possible de supprimer toute la liste de commande ou de supprimer un élément d'une des trois tableView en effectuant un slide à gauche.
Si l'utilisateur désire sauvegarder son plan de vol, il a la possibilité de remplir la boite de texte située sous le bouton de sauvegarde pour lui donner un nom, puis d'appuyer sur ce bouton. La sauvegarde s'effectue dans le format Json et utilise deux structures définit dans le fichier File.swift. La structure "File" permet d'écrire un objet Json, elle possède uniquement les attributs nécessaires à la sauvegarde et un constructeur. La deuxième structure "Fichier" est utilisé pour lire un fichier Json, elle possède une énumération de CodingKeys qui permet la reconnaissance et l'extraction des attributs des objets Json.
Lorsque le bouton est appuyé, le système vérifie si un fichier existe dans le cheminement donné par la variable "jsonURL". Ce cheminement correspond au seul emplacement où nous possédions le droit d'écrire depuis un programme swift, il est donné par la variable userDomainMask auquel s'ajoute le nom du fichier que nous créons, "flight_plans.json". Dans le cas où un fichier de sauvegarde n'existe pas, une variable "fichier" de la structure "File" est créé avec toutes les informations contenues dans un plan de vol. Cette variable est ensuite convertie en Json et stocké dans le fichier "flight_plans.json". Si un fichier de sauvegarde existe déjà,

le programme y lit tous les éléments pour les stocker dans une variable de la structure "Fichier" et les réécrit en ajoutant également la nouvelle sauvegarde. Cette manœuvre peut sembler absurde mais le format Json nous empêche d'ajouter simplement un nouvel élément à un ensemble d'objets déjà existant.
La dernière spécificité de cette page est le bouton simuler. Son fonctionnement est clair, il vérifie si la liste de commande appartient à la grammaire définit au point **2.2**, si l'utilisateur entre des commandes erronées, il sera informé de la raison exacte du problème. Si la liste est conforme à la grammaire alors l'application passe à la vue simulation en transmettant les variables définis dans le plan de vol. Nous verrons son fonctionnement en détail au point **3.5**.

Le résultat est le suivant:



3.4 Gestionnaire

Après avoir sauvegardés des plans de vols, l'utilisateur peut sélectionner le gestionnaire pour les afficher et les charger. Cette vue est constituée d'une unique tableView qui va chercher, dans l'emplacement défini à la création de vol, le même fichier "flight_plans.json". A l'initialisation, elle va appeler la fonction "Download-JSON" qui utilise la structure "Fichier" pour lire et charger tout les objets json dans une liste nommée "files". La tableView s'occupe ensuite d'afficher sur chacune de ces lignes le nom des éléments se trouvant dans "files".
En outre, il est également possible de supprimer un plan de vol depuis le gestionnaire en effectuant un slide à gauche. Cette fonctionnalité est similaire à la sauvegarde d'un nouveau plan de vol, sauf qu'au lieu d'ajouter un objet json,

nous en enlevons un. Par conséquent, son implémentation suit presque le même raisonnement. Puisque le format Json nous empêche d'ajouter ou de supprimer un élément d'un ensemble d'objet json, le programme lit tous les plans de vols à l'exception de celui qui se trouve à la ligne supprimée. Ainsi, il suffit ensuite de réécrire ce qui a été lu afin de retirer la bonne sauvegarde. Finalement, lorsqu'un plan de vol est sélectionné, l'utilisateur est mené sur la page de création de vol et sa variable "sauvegarde" devient l'élément dans "files" à l'index de la ligne sélectionnée. A chaque fois qu'elle est initialisée, la vue création de vol vérifie si cette sauvegarde n'est pas vide. Si oui, la liste de commande, d'obstacle et d'objectif reste vide, si non elles prennent celles stockés dans la variable "sauvegarde".

Un aperçu de la page avec des noms de sauvegarde aléatoire:

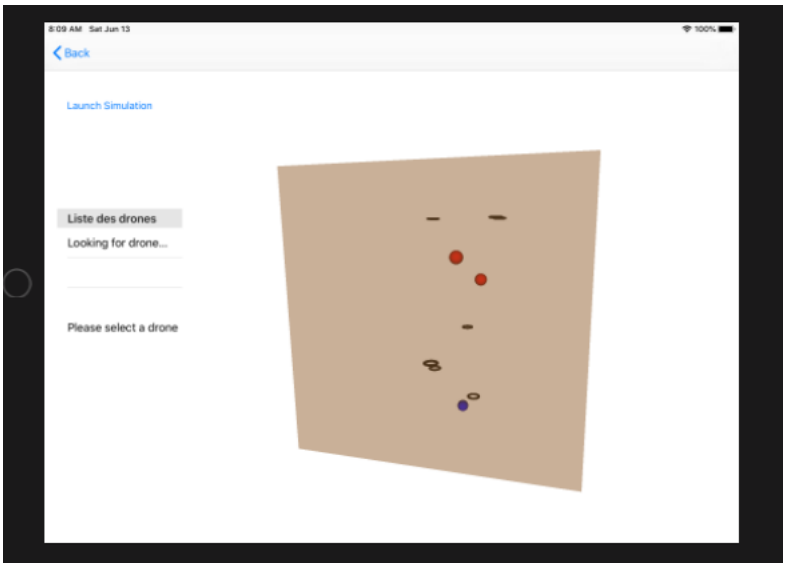
3.5 Simulation

Une fois que l'utilisateur se trouve dans la page de simulation, il verra une représentation en trois dimensions du plan de vol qu'il désire simuler. Afin de la réaliser, nous avons utilisé SceneKit, une interface de programmation de haut niveau permettant de créer des graphiques 3D. Dans un premier temps, nous avons créé une scène qui constitue la zone d'affichage des éléments que nous voulons modéliser. Dans cette espace, il est possible de déclarer des "Nodes" qui possède un large éventail de méthode permettant d'obtenir le rendu désiré. Le premier Node essentiel à la visualisation de notre simulation est celui qui définira une caméra. Cette caméra possède un vecteur positon (x,y,z) qui donne le point de vue sur notre graphique. Ensuite, nous avons défini un cube de taille 20x20x20 dont les valeurs sur les axes

varient entre -10 et 10. Ce cube représente la zone de vol limite à ne pas dépasser lors d'un trajet. Afin de pouvoir représenter le drone, les obstacles et les objectifs à l'intérieur de celui-ci, nous avons dû lui donner une couleur transparente le distinguant du blanc apporté par la scène.
Le drone est représenté par une sphère bleue de rayon 0.5 positionnée initialement au centre de la face du bas (x:0, y:-10, z:0). Lors de l'initialisation de la page, la liste d'obstacles et d'objectifs, reçus par la vue de création de vol, sont traitées afin de positionner des sphères rouges (les obstacles) et des anneaux noirs (les objectifs) aux positions indiquées par les attributs de leurs objets.
Une fois toutes les composantes en place, il est nécessaire de définir un ensemble d'action exécutable par le drone. Pour cela, nous exprimons nos commandes par des mouvements d'une seule unité dans chaque direction sous forme vectorielle. Par exemple, le décollage correspond à un mouvement d'une unité sur l'axe y (x:0 ,y:1, z:0). Toutes nos commandes seront donc représentées de cette manière excepté l'atterrissage qui nécessite le calcul du mouvement total sur l'axe des Y afin de déterminer le nombre à y soustraire pour obtenir la valeur initiale de -10. Pour ce faire, nous avons mis en place un compteur qui s'incrémente ou se décrémente si le drone effectue un mouvement positif ou négatif sur l'axe y. Avec tous ses éléments, il est possible d'obtenir une séquence d'action en traduisant la liste d'entier que représente notre liste de commande à l'aide d'un "switch statement". C'est donc en utilisant la méthode "runAction" de SceneKit sur la sphère modélisant le drone que la simulation s'effectue au moment où le bouton "Lauch Simulation" est appuyé.
Cependant, il est maintenant nécessaire de vérifier si le drone ne touche aucun obstacle et passe par tous les objectifs donnés sans dépasser la limite imposée par le cube. SceneKit ne nous permettant pas d'obtenir la position d'un objet en mouvement, nous avons également ajouté des compteurs sur les axes x et z. A l'aide de ces compteurs nous avons pu simuler la position du drone en les mettant à jour dans le même "switch statement" qui détermine la séquence de mouvement que le drone va exécuter. Nous avons également défini deux flags qui prennent respectivement la valeur de 1 si une des conditions n'est pas satisfaite. Par exemple, si la position dépasse la valeur -10 ou 10 sur un des axes, un flag est activé et si la position est identique à celle d'un obstacle alors un autre flag est activé. Dès qu'un flag est actif, le programme quitte le "switch statement" et indique à l'utilisateur la raison de l'échec de la simulation.
Toutefois, si la simulation se termine correctement, l'utilisateur pourra se connecter au drone en sélectionnant dans une tableView le réseau affiché. La connexion s'effectue exactement de la manière décrite au point **3.2** et la communication des méthodes est très similaire puisque ici au lieu d'appeler une méthode de la classe "bebopDrone" lorsqu'un bouton est appuyé, on effectue un autre "switch

statement" qui selon l'entier présent dans la liste de commande appelle une fonction effectuant une action spécifique sur le drone. Chaque entier représente une commande comme décrit au point **3.3** et lorsqu'il est lu, appelle la méthode correspondante décrite au point **3.2**. Il est important de noter que nous appelons presque toutes les fonctions de la classe "bebopDrone" avec l'entier 50 ou -50 en argument, cela correspond à peu près à un déplacement de 10 centimètres du drone. Les seules méthodes qui ne prennent pas de valeurs en arguments sont le décollage et l'atterrissage.

Voici la page à son état initial, il est possible de tourner le cube ou d'effectuer un zoom.



Chapter 4: Conclusion

4.1 Conclusion

En résumé, pour pouvoir développer cette application, nous avons d'abord analyser et compris le travail fait en amont afin de visualiser sur la forme comme sur le fond notre application. Cela nous a permis de diviser le travail en un ensemble de sous-tâche planifié que nous avons essayé de réaliser dans le temps prévu. C'est donc en suivant ce plan que nous avons pu construire notre application étape par étape.
Ce projet nous a permis de bonifier notre capacité à travailler de manière autonome, car nous avons dû chercher et comprendre les bonnes documentations afin d'implémenter les fonctionnalités désirées. C'est d'ailleurs peut être ce point qui nous a parfois fait défaut pour certaines tâches qui ont nécessitées plus de temps que prévu. De plus, ce travail ayant été réalisé à deux, il peut arriver que des avis divergent sur certaine manière d'aborder un problème, mais nous avons su partager le travail de manière équitable et intelligente pour maximiser au mieux notre productivité sans le moindre souci.
Nous avons pu à travers ce projet, nous rendre compte du travail conséquent que représente la réalisation d'une application. Cette tâche nous a contraint à fournir un effort régulier et ordonné pour parvenir à un résultat satisfaisant. Il est vrai qu'un ancien projet nous a été fournis mais il est nécessaire de prendre conscience que cela ne rend pas forcément la tâche plus aisée. Cela nous oblige à assimiler une grande diversité de nouvelles technologies très rapidement en nous référant à un code que nous n'avons pas conçu et dont nous ne possédions que très peu de documentation. C'est ce plongeon dans l'inconnu qui marque d'autant plus l'importance d'une bonne analyse permettant d'éclaircir le sujet et de planifier un ensemble de tâche nous guidant vers le résultat escompté.
Pour conclure, voici le lien de notre github contenant l'intégralité de notre projet et une documentation supplémentaire décrivant le fonctionnement de chaque page en plus des commentaires se trouvant sur leur fichier associé.