Documentation

Grand-projet programmation

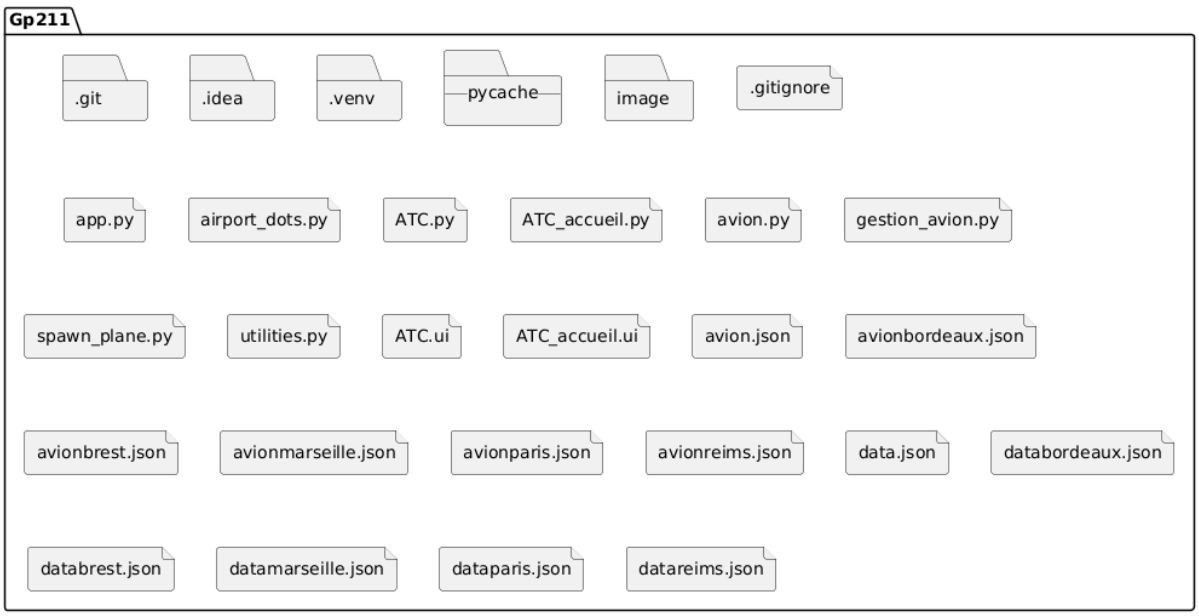
Pour ce code, voici les bibliothèques que nous avons utilisé :

Math pour calculer des distances, des caps, …

Pyside6 pour l'interface graphique

Json pour l'import et l'export de fichier json

Nous utilisons 3 types de fichiers, les fichier python, les fichier json et les fichiers pyside. Voici donc les fichiers :



Les fichiers python sont :

- avions.py contient la classe « avion ».
- utilities.py qui contient des fonctions pour importer des fichier json vers du python, mettre à jour les fichiers « data.json » et « avion.json » et également calculer la distance entre 2 avions.
- gestion_avion.py gère les avions en mettant à jours leurs paramètres et en créant un dictionnaire contenant tous les objets « avion ».
- airport_dots.py qui gère l'affichage du point de l'aéroport.

- spawn_plan.py gère l'affichage des avions.
- ATC.py est le fichier graphique qui correspond à la page du jeu.
- ATC_accueil est le fichier graphique qui correspond à l'accueil du jeu.
- app.py est le fichier du jeu, c'est lui qui gère le timing, les liens entre les actions du joueur et les fichiers concernés, ...

Les fichiers json sont :

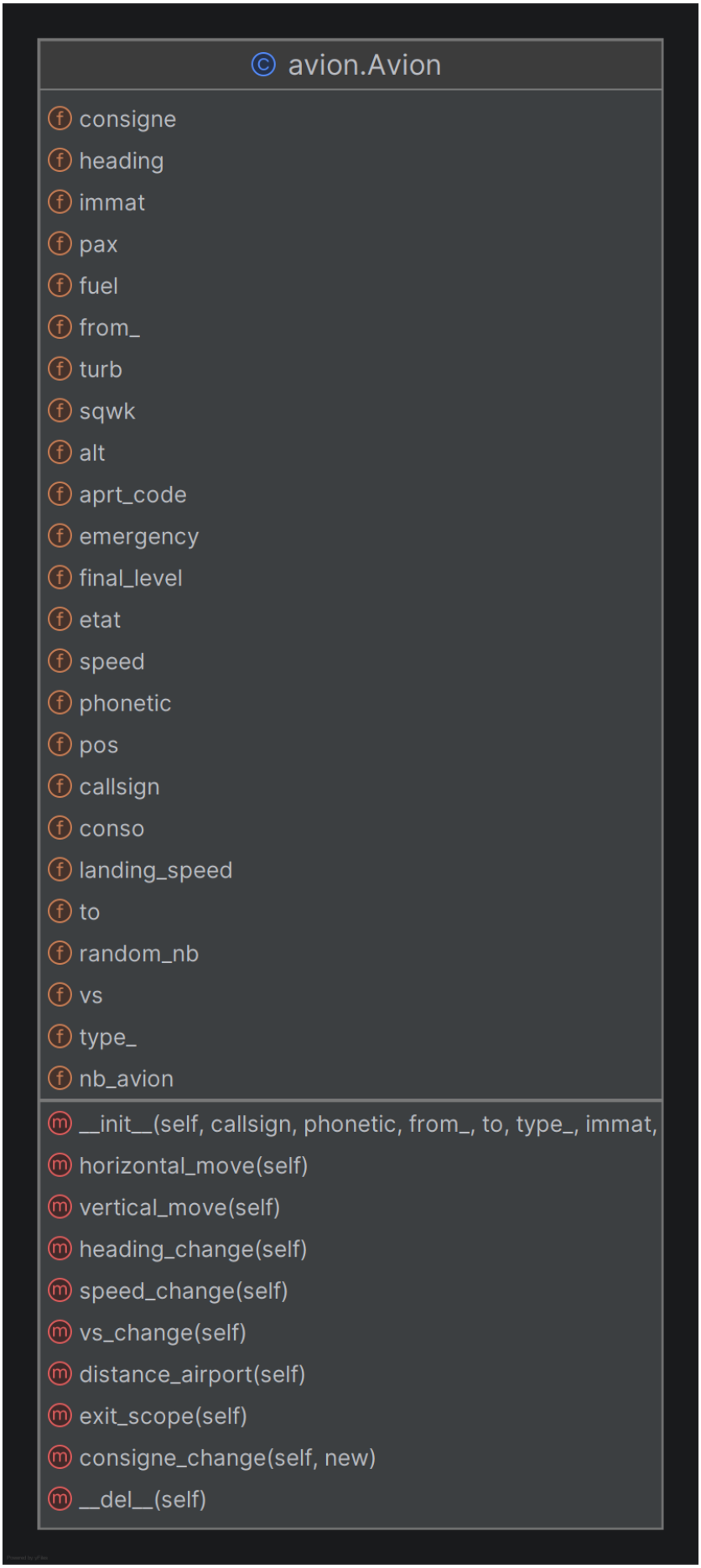
- data{FIR}.json (ex : databrest.json) qui contiennent les informations des aéroports ainsi que des points d'apparitions des avions de la zone de contrôle.
- avion{FIR}.json (ex : avionbrest.json) qui contiennent les informations des avions présent dans la zone de contrôle.
- data.json et avion.json contiennent des informations pour la zone en cours et sont mis à jour à chaque changement de zone.

Les fichiers pyside :

- ATC.ui a permis de générer le fichier python du même nom.
- ATC_accueil.ui a également permis de générer le fichier python du même nom.

Il y a également des images qui sont stocké dans le fichier du même nom, elles sont au format png.

Voyons maintenant la classe avion.


avion.Avion
consigne
heading
immat
pax
fuel
from_
turb
sqwk
alt
aprt_code
emergency
final_level
etat
speed
phonetic
pos
callsign
conso
landing_speed
to
random_nb
vs
type_
nb_avion
__init__(self, callsign, phonetic, from_, to, type_, immat,
horizontal_move(self)
vertical_move(self)
heading_change(self)
speed_change(self)
vs_change(self)
distance_airport(self)
exit_scope(self)
consigne_change(self, new)
__del__(self)

La variable de classe ‘nb_avion’ renseigne sur le nombre d’avion dans la classe.

L’ensemble des attributs ne sont pas forcément actif mais sont présents car ils devaient être affiché en dessous de la carte. Les attributs jouant un rôle actif dans l’exécution du code sont :

- self.callsign renseigne le callsign de l'avion et donc identifie l’avion sur l’écran.
- self.to car il renseigne sur l’aéroport d’arrivé de l’avion et donc les coordonnées d’où il doit aller.
- self.fuel car il stocke le niveau de carburant restant dans l’avion
- self.pos car c’est une liste qui renseigne les positions x et y de l’avion.
- self.heading car il stocke le cap actuel de l’avion.
- self.speed car il stocke la vitesse actuelle de l’avion.
- self.vs renseigne sur la vitesse verticale actuelle de l’avion.
- self.conso car il sert à savoir de combien doit décroitre le carburant de l’avion par mouvement.
- self.alt renseigne l’altitude actuelle de l’avion.
- self.landing_speed stocke la vitesse d’atterrissage de l’avion.
- self.consigne conserve les ordres donnés à l’avion par le joueur.
- self.etat conserve l’état du TCAS donc si l’avion est trop proche d’un autre avion et l’état d’atterrissage donc si l’avion est suffisamment proche de l’aéroport pour initier la méthode « landing ». Il renseigne également sur le fait que l’avion soit posé ou non.
- self.to conserve les données de l’aéroport d’arrivé (position et orientation de la piste)
- self.sqwk renseigne sur la valeur du transpondeur de l’avion et qui renseigne sur une potentielle urgence rencontrée par l’avion.
- self.random_nb contient le numéro de ligne du fichier json. Il est stocké pour supprimer ce numéro de la liste des lignes utilisées lors de la suppression de l’avion.
- self.aprt_code renseigne le code de l’aéroport de destination. Il peut être modifier en fonction de si l’avion est en situation d’urgence
- self.max est un dictionnaire qui contient les valeurs max de l’avion (altitude max, vitesse max, ...)

Les attributs ne jouant pas de rôle actif dans l’exécution du code et n’étant là que pour être affiché, sont :

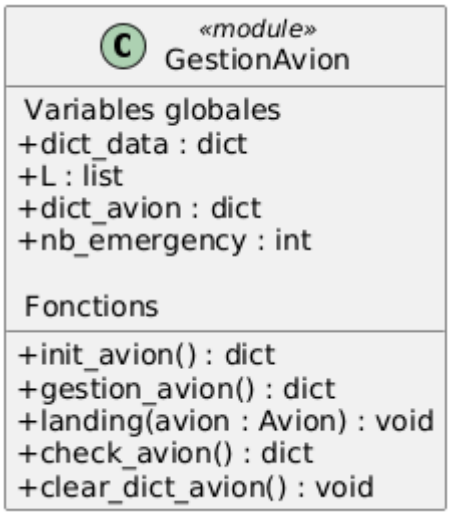
- self.phonetic renseigne l’indicatif radio (comment le contrôleur doit appeler l’avion à la radio).
- self.from renseigne l’aéroport de départ de l’avion.

- self.type renseigne le type d'avion (A320, B777, ...).
- self.immat renseigne l'immatriculation de l'avion.
- self.turb renseigne sur l'intensité des turbulences de sillage de l'avion.
- self.pax renseigne le nombre de passagers à bord de l'avion.
- self.final_level renseigne le niveau de vol final de l'avion.

Des différentes méthodes de la classe sont :

- __init__ : cette méthode définit les attributs en fonctions d'un des paramètres rentrés lors de son appel.
- horizontale_move : cette méthode prend pour argument les valeurs de l'échelle sur x et y et renvoie la position de l'avion sur l'écran. Pour faire bouger l'avion, la méthode se base sur la vitesse de l'avion, sur son cap et sa position pour calculer le nouveau point.
- vertical_move : cette méthode permet de faire varier l'altitude de l'avion en fonction de la consigne d'altitude donné. Si une consigne de vitesse verticale est renseignée, cette consigne sera appliquée, et l'avion descendra à cette vitesse. Dans le cas contraire, la vitesse de 1500 ft/min est automatiquement appliqué. Tout ces paramètres sont maintenus jusqu'à ce que la nouvelle altitude soit atteinte.
- heading_change : cette méthode permet de faire varier le cap de l'avion en fonction de la consigne de cap donné. Cette méthode calcul la rotation la plus rapide pour se rendre au nouveau cap et fais tourner l'avion à un rythme de 3°/s jusqu'à arriver au nouveau cap.
- speed_change : cette méthode permet de faire varier la vitesse de l'avion en fonction de la consigne de vitesse donné. Cette fais accélérer ou décélérer l'avion à un rythme de 5kts/s jusqu'à arriver à la nouvelle vitesse.
- vs_change : cette méthode permet de faire varier la vitesse verticale de l'avion en fonction de la consigne donné. Cette méthode fait augmenter ou diminuer la vitesse verticale à un rythme de (150ft/min)/s jusqu'à arriver à la nouvelle vitesse verticale.
- distance_airport : cette méthode prend comme paramètre l'aéroport d'arrivé et permet calculer la distance entre l'avion et ce dernier. Si la distance est inferieure à 50, l'attribut self.etat est modifié et la valeur « False » associé à la clé « can_land » devient « True ».
- exit_scope : cette méthode permet de vérifié que l'avion est toujours dans la zone de jeu, sinon le cap de l'avion est modifié et l'avion tourne à 180°.
- consigne_change: permet de mettre à jour le dictionnaire de consigne de l'avion
- __del__ : supprime l'objet avion.

Nous allons maintenant nous intéresser au fichier gestion avion. Voici le diagramme UML de ce fichier.



Ce dernier est constitué de 2 dictionnaires un pour les avions en cours de vol, l'autre pour conserver les données de la carte actuelle (aéroport, point d'apparition des avions). Il y a également une liste qui conserve les valeurs utilisées pour choisir les avions qui s'affichent et donc évité que le même avion s'affiche plusieurs fois ainsi qu'une variable qui permet de cpnnaitre le nombre d'urgence en jeu et évité qu'il y en ait trop.

La fonction init_avion permet de générer les objets à partir de la classe avion et sont conservés dans le dictionnaire « dict_avion ».

La seconde fonction permet de faire varier tous les paramètres des avions en fonction des consigne données. Elle permet également de vérifier la distance entre les avions et d'activer les TCAS des appareils (modéliser dans l'attributs self.etat à la clé 'TCAS') qui passe de False à True. Cette fonction active également l'atterrissage de l'avion si le joueur a cliqué sur le bouton land et que l'attribut self.etat à pour valeur True pour la clé 'can_land'. Cette fonction permet également de gérer les urgences et d'appliqué des consignes spécifique en fonction de l'urgence rencontré par l'avion.

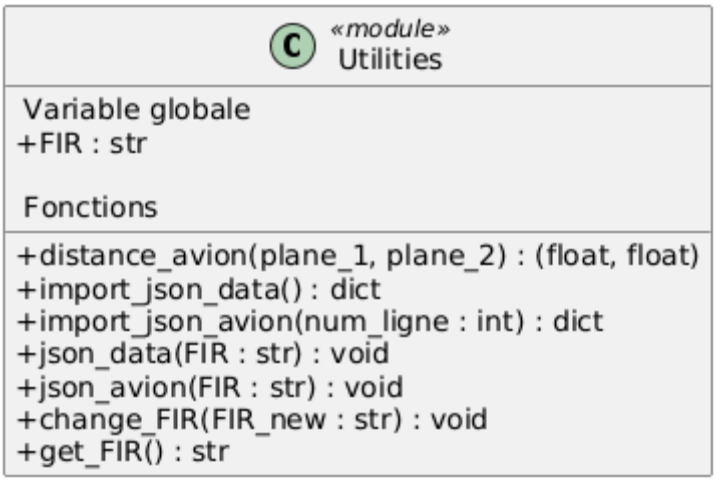
La fonction check_avion permet de supprimer tous les avions qui doivent l'être (en cas de collision, atterrissage ou autre).

Ces trois fonctions retournent le dictionnaire avion qui peut alors être utilisé dans d'autre fichier notamment pour afficher les avions.

La fonction landing prend comme paramètre le dictionnaire contenant les données d'un avion et permet de faire atterrir l'avion. Pour cela, il calcul le cap pour se rendre à l'aéroport et s'y rend en même temps d'amorcer sa descente vers 1500ft. Quand il arrive à l'aéroport, il continue de descendre en spirale si son altitude est toujours supérieure à 1500ft et en même temps, il réduit sa vitesse vers sa vitesse d'atterrissage. Quand l'altitude est à 1500ft et que sa vitesse est celle d'atterrissage, l'avion se dirige vers le cap opposé à la piste pendant 1min et descend à une vitesse verticale de 500ft/min. Après cela, il fait un 180° pour s'orienté dans l'axe de la piste tout en descendant à 500 ft/min. Enfin, l'avion approche dans l'axe de la piste en descendant à 500 ft/min.

Enfin la fonction clear_dict_avion, permet de supprimer toutes les données de la liste L et du dictionnaire contenant les avions.

Nous allons enfin voir le fichier utilities dont voici le diagramme UML :



Ce fichier contient 7 fonctions :

- distance_avion : cette fonction prend comme paramètre 2 objets de la classe avion et retourne la distance ainsi que la différence d'altitude entre les 2.
- import_json_data : cette fonction permet d'importer le fichier « data.json » et de renvoyer le dictionnaire comportant toutes les informations du fichier.
- import_json_avion : cette fonction prend comme paramètre l'indice de l'élément à retourner et permet d'importer le fichier « avion.json » et de renvoyer le dictionnaire comportant toutes les informations du fichier à l'indice choisit.
- json_data : cette fonction prend comme paramètre la zone de contrôle dans laquelle on est sous le nom « FIR » et permet d'importer le fichier « data{FIR]}.json » et de transférer ses données dans le fichier « data.json ».

- import_json_avion : cette fonction prend comme paramètre la zone de contrôle dans laquelle on est sous le nom « FIR » et permet d'importer le fichier « avion{FIR]}.json » et de transférer ses données dans le fichier « avion.json ».
- change_FIR permet de changer la valeur de la variable globale du fichier.
- get_FIR permet de renvoyer la variable globale FIR.

Le fichier ATC contient du code qui gère la partie graphique du jeu. Il permet d'afficher la fenêtre principale du jeu.

Le fichier ATC_accueil fait la même chose mais pour la fenêtre d'accueil.

Le fichier airport_dots permet l'affichage des points d'aéroports sur la carte.

Le fichier spawn_plane permet l'affichage des cercles autours des aéroports représentant la zone d'approche. Il permet également la gestion des objets graphiques des avions, il modifie le cap, la position et la couleur en fonction de la situation. Il permet également d'afficher les informations du vol quand on passe la souris dessus et si on clique dessus, d'autre informations apparaissent en bas de l'écran et on peut modifier les paramètres de l'avion.

Le fichier app permet de faire la gestion entre les consignes entrée par l'utilisateur et l'objet avion. Il permet également de faire fonctionner le jeu, de charger les aéroports sur la carte et d'interfacé les boutons de l'interface.

Pour finir, voici un diagramme montrant tous les liens qui existent entre les fichiers.