Rapport - **Programmation C++**

MARIÉ Brayan

Master 1 - Informatique
Université **Gustave EIFFEL**

2021 - 2022

Contents

1 Introduction

Ce projet est un travail à réaliser dans le cadre du second semestre de Master Informatique à l'université Gustave EIFFEL pour le cours de programmation C++. Ce projet est déjà fourni et codé mais il manque un bon nombre de choses que nous allons devoir compléter au fur et à mesure de l'apprentissage des notions du cours pour appliquer les connaissances acquises dans le projet.

2 TASK 0 - Se familiariser avec l'existant

2.1 Exécution

Dans le fichier tower_sim.cpp. On remarque assez rapidement que la fonction responsable des inputs est la fonction **create_keystrokes()**.

Nous avons alors plusieurs commandes pour différentes fonctionnalités :

- x ou q pour quitter le programme
- + ou - pour zoomer, dezoomer
- f pour passer en plein écran
- c pour ajouter des avions

Après avoir ajouté des avions, ceux-ci vont arriver à l'écran puis tourner dans la carte avant d'avoir un terminal de libre. Ensuite, ils vont se poser dans un terminal, puis décoller à nouveau. On peut voir dans la console les différents états dans lesquelles les avions sont.

2.2 Analyse du code

Dans ce code, il y différentes classes, nous allons les lister et expliquer leurs rôles :

- **Displayable**, représente un objet qui sera affiché à l'écran
- **DynamicObject**, représente un objet qui aura un comportement changeant au cours du programme.
- **Texture2D**, représente la classe graphique de ce projet
- **Aircraft**, représente un Avion
- **AircraftTyp** (struct) représente le type d'un Avions
- **Airport** représente un aéroport
- **AirportType** représente le type d'un aéroport
- **RunWay**, représente la piste d'atterrissage
- **Terminal**, représente un terminal

- **Tower**, représente la tour du contrôle de la station
- **TowerSimulation**, cette classe est la classe motrice du projet c'est-à-dire c'est dans cette classe que tout le processus va ce produire.
- **Waypoint**, représente le chemin d'un avion.

Fonction membres de la classe Tower :

- **get_instructions**, fonction qui va fournir un avion, un chemin a un avion.
- **arrived_at_terminal**, sert a gerer lorsqu'un avion arrive a un terminal

Fonction membres de la classe Aircraft :

- **get_flight_num**, renvoi du numéro de vol de l'avion
- **distance_to**, retourne la distance entre l'avion et un point
- **display**, fonction d'affichage d'un avion
- **move**, fonction de mouvement de l'avion qui va changer l'état de l'avion dans le projet et dans les structures de données.

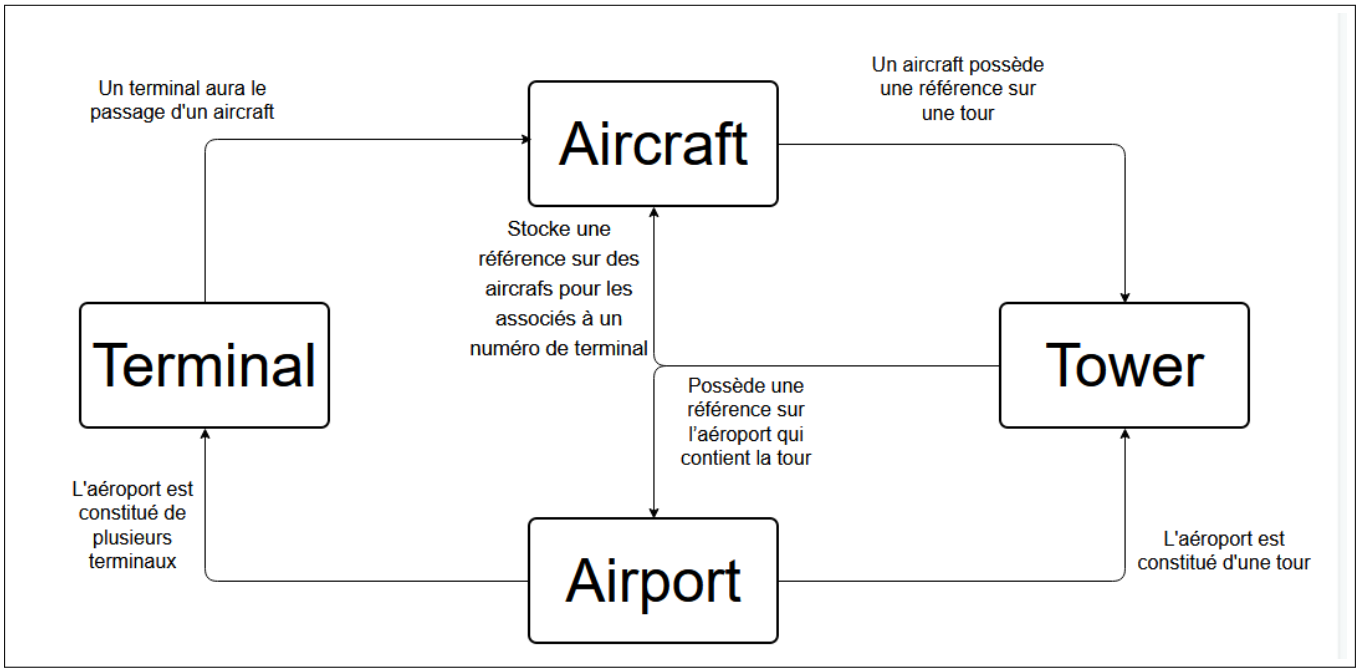
Fonction membres de la classe Airport :

- **get_tower**, retrourne la tower qui est lié a l'aéroport
- **display**, fonction d'affichage d'un aéroport
- mettre a jour l'état de l'aéroport

Fonction membres de la classe Terminal :

- **in_use**, indique si le terminal est libre ou pas
- **is_servicing**, indique si il y a un avion sur le terminal
- **assign_craft**, assigne un avion au terminal
- **start_service**, commence le processus de service
- **finish_service**, termine le processus de service
- **move**, mettre à jour l'état du terminal

Schéma de l'architecture du projet :



Explication de la génération de chemin :

On remarque assez rapidement que le "chemin", va être stocké dans un champ du type WaypointQueue présent dans la classe Aircraft. Le processus sera le suivant. Les aircrafts vont demander aux tours de contrôle un chemin.
La tour de contrôle (Tower) va alors lui envoyer un chemin à devoir suivre pour rejoindre un terminal ou alors un chemin qui va faire le faire tourner en cercle si aucun terminal n'est disponible.

Pour représenter cela, une dequeue est très utile dans ce car on va utiliser les différents points et les sortir de la queue dans l'ordre d'insertion des différents points du chemin.

2.3 Bidouillons

Les vitesses des avions sont définis dans la classe **AircraftType** via l'élément **max_air_speed**. Si on modifie cet élément, on va alors changer l'accélération de l'avion.

La variable qui contrôle le *framerate* de la simulation est la variable **ticks_per_sec**. Il faut faire attention avec cette variable car il ne faut pas qu'elle atteigne 0 sous peine de faire une division par zéro, ce qui n'est pas possible.

Le temps de débarquement d'un avion est géré par la variable **SERVICE_CYCLES** dans le fichier **config**.

2.4 Théorie

Pour que seulement la classe Tower puisse réserver un Terminal nous avons défini la classe Tower comme classe dite **friend** pour qu'elle ait l'accès aux champs privés de la classe **Airport**.

Il ne faut ici pas passé par une référence constante car on ne veut pas toucher à la valeur du point car il sera modifié dans une autre fonction c'est pour cela que l'on va préférer faire une simple copie.

3 TASK 1 - Gestion des Ressources

3.1 Objectif 1 - Référencement des avions

3.1.1 Choisir l'architecture

Maintenant grâce à la mise en place de notre **AircraftManager**, c'est celui-ci qui sera donc maintenant responsable de la destruction de nos avions.
Lorsqu'un avion est détruit, la move_queue ainsi que Terminal possède une référence sur l'avion.
Pour supprimer ces références, on va mettre les réfrences à **null_ptr**.

Il ne faut pas faire la même chose pour notre AircraftManager car sa durée de vie n'est pas la même. En effet, le programme aura besoin d'un accès du début jusqu'à la fin à notre AircraftManager.

3.1.2 Objectif 2 - Usine à avions

Nous allons maintenant voir comment créer une usine à avions qui aura pour rôle de générer des avions. Cette factory va avoir aussi dans un second temps le rôle de supprimer certaines variable global de notre code. C'est aussi grâce à cela que l'on va donner *l'owernership* au manager.

La factory sera alors situé dans un champ dans notre classe **tower_sim**. Pour éviter le fait d'avoir plusieurs avions qui possèdent le même nom, on va avoir en mémoire le nom des différents avions qui ont déja été générés par la factory. De ce fait il va pouvoir vérifier les informations facilement et rapidement.

4 TASK 2 - Algorithmes

4.1 Objectif 1 - Refactorisation de l'existant

Pour cette partie de la task, nous avons simplement modifié notre code pour se servir des méthodes de **algorithm** et **numeric**.

4.2 Objectif 2 - Rupture de Kérosène

Nous allons dans cette section ajouter un champ fuel qui sera décrementer au fur et à mesure des mouvements d'un avion. Ce champ représentera la quantité de carburant présent dans le réservoir de l'avion. Si cette valeur passe à 0, alors cela signifie que l'avion n'a plus de carburant pour continuer sa route et donc on imagine que ses moteurs ne fonctionnent plus, par conséquent l'avion crash. Ce crash implique la destruction de l'avion dans un premier temps, on verra dans la suite du projet que la panne de carburant d'un avion lève une exception.

Pour qu'un avion puisse se réapprovisionner en carburant, il va devoir faire un arrêt sur un terminal. Pour cela nous allons trier les avions par ordre de priorité. En effet un avion qui aura moins de carburant sera prioritaire par rapport à un avion qui possède son réservoir rempli dans la réservation d'un terminal. Ils seront donc triés par rapport à la quantité de carburant présent dans leurs réservoirs. C'est au moment du mouvement d'un avion que le tri sera fait.

5 TASK 3 - Assertions et exceptions

5.1 Objectif 1 - Crash des avions

Nous allons maintenant mettre en place un système pour savoir combien d'avion se sont crashés grâce à la touche 'm'. Pour cela, nous allons introduire dans la classe **AircraftManager** un nouveau champ privé qui est un entier qui va nous servir de compteur.

Maintenant, dans la fonction il va falloir détecter qu'un avion s'est crashé. Pour cela lorsqu'un avion est en panne, il va lancer une Aircrash exception. Et nous allons attraper cette exception dans notre fonction move de notre manager. De cette façon, il pourra incrémenter le champ qui gère cela en fonction.

5.2 Objectif 2 - Détecter les erreurs de programmation

Pour cette section, il est question d'ajouter des assert partout ou il sera nécessaire dans notre programme pour éviter les erreurs états, d'initialisation ou encore de vérifier les paramètres d'une fonction.

6 TASK 4 - Templates

6.1 Objectif 1 - Devant ou derrière ?

La fonction add_waypoint permet de rajouter une étape au début ou à la fin du parcours de l'avion. Pour voir la différence dans ces deux cas, elle prend un argument booléen front (on parle alors de "flag"). On veut maintenant que l'évaluation de ce flag ait lieu au moment de la compilation et non pendant l'exécution ce qui est le cas pour le moment. Nous allons donc mettre en place un template ainsi qu'un if constexpr pour que l'expression qui est après le if soit évaluée à la compilation.

Conclusion des deux codes à comparer :
Ce sont deux fonctions qui sont fondamentalement les mêmes a l'exception que l'une utilise un template et l'autre non. Dans le cas de la fonction sans template, tous les cas sont écrits en assembleurs avec la fonction templatée il n'y a pas tant de cas. La fonction templatée ne teste pas en temps d'exécution si le booléen mais en temps de compilation (2 fonctions différentes sont généré: une avec true et une avec false)

6.2 Objectif 2 - Points génériques

Nous devons dans cette section, changer les classes Point3D et Point2D. Il faut alors se servir d'un template pour définir un point qui pourrait être un Point d'une certaines dimension.

Pour ce qui est de la partie où il faut créer un constructeur unique pour tous les types, j'ai eu quelques difficultés donc cette partie est commentée dans le code.

7 Conclusion du projet

7.1 Ce que j'ai aimé

Tout d'abord, le langage c++, en effet c'est un langage qui est assez difficile à prendre en main à première vu mais qui au final s'avère être un excellent langage de programmation notamment pour la programmation objet. Ce langage est très puissant et la documentation est très explicite ce qui rend le langage très agréable à utiliser.

Le format du projet, c'était pour moi tout nouveau de faire un projet déjà fait et de le compléter sous forme de task à réaliser, de plus les enseignants étant très à l'écoute cela permet de travailler efficacement et de bien comprendre les subtilités du langage.

7.2 Ce que j'ai appris

Lors de ce projet, j'ai renforcé ma façon de programmer en programmation objet, en effet, au cours de la formation je ne connaissais que le langage Java et le C++ m'a permis de me rendre compte des alternatives en matière de programmation objet.
De plus, j'ai aussi appris à comprendre du code déjà fait et non par moi-même ce qui nécessite une meilleure appréhension de ce que je fais sur un code existant.