Rapport projet C++

-

DURAND Florian

Table des matières

Lien GIT : 3
Réponses aux questions : 3
TASK_0 3
TASK_1 4
TASK_2 5
TASK_3 5
TASK_4 5
Architecture du programme : 5
AircraftManager et AircraftFactory : 5
TexturePool : 6
Général : 6
Difficultés : 6
Smart pointers : 6
Gestion du fuel : 6
Constructeurs : 6
Variadic template : 7
Bon et mauvais aspects : 7
Ce que j'ai appris : 7

Lien GIT :

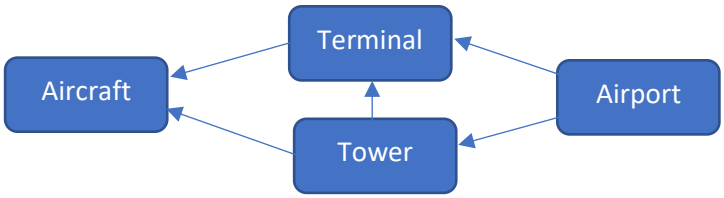
https://github.com/DurandFlorian/CPP_Learning_Project.git

Réponses aux questions :

TASK_0

A. Il faut appuyer sur 'C' pour ajouter un avion, il faut appuyer sur 'Q' ou 'X' pour quitter le programme. La touche 'F' permet d'afficher le programme en plein écran.
L'avion apparaît à un endroit aléatoire de l'écran entrain de planner et il se pose sur l'aéroport, la console affiche le numéro de l'avion qui se pose sur l'aéroport et le numéro de l'avion qui décolle de l'aéroport. Si un avion ne peut pas obtenir un terminal il continue son envol.

B. Aircraft : représente un avion, elle permet de créer des avions avec différentes propriétés.
Airport : représente un aéroport, elle permet de créer un aéroport avec une liste de terminal et une tour de contrôle.
AirportType : représente un type d'aéroport, elle permet d'avoir différents types d'aéroport dans le programme.
Terminal : représente l'un des terminaux d'un aéroport, elle permet d'accueillir un avion sur l'aéroport s'il est enregistré au près du terminal en question.
Tower : représente la tour de contrôle d'un aéroport, elle permet de faire le lien entre les terminaux et les avions en leur donnant le chemin à suivre.
TowerSimulation : représente la classe qui gère le programme en général, elle permet initialiser le lancement du programme et de gérer les inputs de l'utilisateur.
Waypoint : Représente un point par lequel un avion va passer, elle permet de créer un chemin qu'un avion va suivre.
Tower::get_instructions : elle décide du chemin qu'un avion doit suivre et enregistre l'avion avec un terminal si c'est possible.
Tower::arrived_at_terminal : lorsque qu'un avion arrive à la position du terminal lance le service du terminal.
Aircraft::get_flight_num : renvoie le numéro de l'avion
Aircraft::distance_to : renvoie la distance entre la position de l'avion et un autre point.
Aircraft::display : affiche l'avion
Aircraft::move : met à jour l'avion en fonction du contexte
Airport::get_tower : renvoie sa tour de contrôle.
Airport::display : affiche l'aéroport.
Airport::move : met à jour les terminaux en fonction du contexte.
Terminal::in_use : renvoie vrai un avion lui a été assigné et faux sinon.
Terminal::is_servicing : renvoie vrai si il a fini de servir un avion et faux sinon.
Terminal::assign_craft : assigne un avion au terminal.
Terminal::finish_service : supprime l'avion assigné au terminal si il fini son service.
Terminal ::move : si le terminal est entrain de servir il met à jour valeur du service_progress.



Les classes Waypoint,Tower et AirportType avec la méthode Tower::get_instructions sont impliquées dans la génération de chemin. Le chemin est représenté par une deque, elle permet d'efficacement ajouter un point dans le chemin au début et à la fin, ainsi que la suppression du premier point par lequel l'avion va passer par exemple.

C. 2. Le programme se met en pause mais il ne redémarre plus.

4. L'avion doit être supprimé quand l'avion décolle dans Aircraft::operate_landing_gear. Il n'est pas sûr de le supprimer directement ici car une autre classe pourrait essayer de l'utiliser après la suppression. Il faudrait le supprimer une fois que tout a été update.Il faut renvoyer un booléen (dans mon cas j'ai choisi d'utiliser un champ).

5. C'est plus logique de créer et supprimer manuellement un DynamicObject sachant que, contrairement à Displayable, aucun destructeur parent pourrait le supprimer dans le cas de notre programme.

D. 1. C'est la classe Tower qui possède les informations de quel avion est enregistré à quel terminal.

2. la méthode turn n'est utilisé qu'avec un point qui résulte de la création d'un nouveau point, on aurait pu initialisé une variable rien que pour lui et ensuite passer la référence à la méthode turn mais ça ne vaut pas le coup autant passer le point par copie directement.

TASK_1

Analyse de la gestion des avions :

Il faudrait parcourir la move_queue ce qui n'est pas très logique.

Objectif 1 :

A. Le deuxième choix fonctionnerait mais c'est plus simple et compréhensible de créer un AircraftManager pour respecter le single responsibility principle.

B. 1. C'est la move_queue qui responsable de la suppression des avions.

2. La display_queue (et la move_queue) contient une référence sur un avion au moment où il doit être détruit.

3. On la supprime dans le destructeur.

4. Il n'est pas judicieux de créer un lien fort entre Aircraft et AircraftManager ça obligerai d'avoir un AircraftManager pour faire exister un Aircraft.

Objectif 3 :
C'est TowerSimulation qui assume l'ownership de TexturePool, c'est à l'initialisation qu'on va charger les textures.

TASK_2

TASK_3

TASK_4

Objectif 1 :

2. Il faut ajouter la valeur du template ici c'est <false>.

3. Le code-assembleur est similaire, seulement dans le code de la fonction template les conditions concernant la valeur de min ne sont pas créées sachant qu'on connaît la valeur de min à la compilation contrairement à la fonction sans template.

Objectif 2 :

4. Erreur à l'éxécution, on ne vérifie pas le nombre d'arguments donc le compilateur utilise le Constructeur de Point avec trois arguments au lieu de lancer une erreur de compilation. Avant le constructeur Point2D était lié au nom Point2D mais ça n'est plus le cas tous les constructeurs de Point sont valides.

5. De même pour Point3D avec 2 arguments mais aucune erreur n'est lancée on aura juste une case avec une valeur aléatoire dans le tableau de values.

6. Il faut préciser le type du premier argument (type T) pour pas que le compilateur utilise le mauvais constructeur.

Architecture du programme :

AircraftManager et AircraftFactory :

J'ai décidé de créer une classe AircraftManager pour respecter le single responsibility principle elle s'occupe de gérer toutes les méthodes qui concernent les avions en général.
La classe AircraftFactory permet de la création d'un avion avec le chargement des textures ainsi que la gestion des airlines. Ainsi j'ai décidé qu'un AircraftManager devait être agrégé à

une AircraftFactory cela implique qu'il n'est pas possible de créer une AircraftFactory sans la présence d'un AircraftManager. J'ai choisi l'agrégation plutôt que la composition car ça me semblait plus logique qu'un AircraftFactory n'ai pas l'ownership d'un AircraftManager.
Ces deux classes sont own par TowerSimulation qui utilise les fonctionnalités de ces deux classes.

TexturePool :

TexturePool s'occupe de stocker les textures, elle permet à l'AircraftFactory de créer et stocker les textures d'avion. C'est TowerSimulation qui own une instance de TexturePool c'est lui qui va fournir les textures aux classes qui en ont besoin. J'aurai également pu faire en sorte que ce soit TexturePool qui load les textures plutôt que la méthode AircraftFactory::loadTypes mais j'ai préféré la première solution car on load les textures d'un avion.

Général :

En général j'ai suivi l'architecture demandée au fil des TASK , les méthodes qui s'appliquaient spécialement à un avion sont dans Aircraft comme Aircraft::refill, les méthodes qui s'appliquaient à plusieurs avions sont dans AircraftManager comme AircraftManager::get_required_fuel, les méthodes liées à l'enregistrement des terminaux sont dans Tower comme Tower:: unbook_terminal.

Difficultés :

Smart pointers :

J'ai eu beaucoup de mal avec les smart pointers, le plus gros problème que j'ai eu était de comprendre pourquoi vouloir utiliser des smart pointers alors que l'on pouvait utiliser des local objects. Mais plus j'ai essayé de les utiliser et plus j'ai compris leur utilité (par exemple le fait ne pas avoir besoin d'initialiser un unique_ptr).

Gestion du fuel :

Je ne comprenais pas pourquoi vouloir implémenter une méthode Tower::reserve_terminal plutôt que d'implémenter la fonctionnalité directement dans Tower::get_instructions. Et je crois que je n'ai toujours pas compris et cela m'a vraiment fait perdre beaucoup de temps sachant que je ne voyais pas où on voulait en venir. De même j'ai dû créer une méthode Tower::unbook_terminal mais je ne suis pas sûr qu'elle était nécessaire sachant que possiblement je n'utilise pas la méthode Tower::reserve_terminal correctement.

Constructeurs :

J’avais du mal à comprendre quels constructeurs étaient automatiquement créés par le compilateur et lesquels pouvait disparaître avec la création d’un constructeur. Mais comme pour les smart pointers, c’était une question de temps pour comprendre tout ça.

Variadic template :

Je n’avais pas du tout compris pourquoi le compilateur n’était pas capable d’utiliser le constructeur de copie directement plutôt que de vouloir passer par le constructeur générique. Alors qu’il suffisait de lui passer un argument de type T pour sache qu’il devait utiliser le constructeur de copie quand il recevait un Point.

Bon et mauvais aspects :

+ Un thème qui parle à tout le monde.
+ Les TASK suivent le déroulement des chapitres de cours.
+ Travailler sur le projet pendant les TD.
+ Toujours quelqu’un pour nous aider si on a besoin.
- La formulation des questions n’était pas toujours clair sur des concepts un peu abstraits.
- Devoir partir d’un programme avec une architecture déjà existante.
+ Ça nous habitue à s’adapter à du code extérieur.

Ce que j’ai appris :

J’ai appris des nouveaux concepts liés à la programmation d’un jeu avec interface graphique, en tout cas plus en profondeur qu’avant.

J’ai appris à coder en essayant d’optimiser les arguments passés dans les fonctions et coder en C avec d’avantage l’utilisation de const.

J’ai appris à quel point les concepts de programmations objets pouvaient rester les mêmes tout en étant très différents sur certains points en comparant JAVA et C++. De ce fait j’ai pu comprendre des concepts que j’utilisais en JAVA mais que je ne comprenais pas en profondeur.

J’ai appris tous ce qu’on a vu pendant le semestre et plus particulièrement en faisant le projet.