Université Gustave Eiffel
Institut-D'électronique-Et-
D'informatique-Gaspard-Monge
Master 1 Informatique
Programmation C++
Groupe 1
Année universitaire 2021-2022

Rapport

Étudiant : Hassani Mehdi

Table des matières

Task 0....4
Objectif A :....4
Objectif B :....4
Objectif C :....7
Objectif D :....8
Task 1....9
Gestion mémoire :....9
Objectif 1 :....9
A - Choisir l'architecture....9
B - Déterminer le propriétaire de chaque avion....9
Objectif 2 :....10
A - Création d'une factory....10
B – Conflits....10
Task 2....11
Objectif 1 :....11
A - Structured Bindings....11
B - Algorithmes divers....11
C - Relooking de Point3D....11
Objectif 2:....12
C - Minimiser les crashs....12
D - Réapprovisionnement....12
Task 3....13
Objectif 1 :....13
Task 4....14
Objectif 1 :....14
Objectif 2 :....14
Points Positifs:....15
Points Négatifs:....15
Ce que j'ai appris:....15

Task 0

Objectif A :

Recherchez la fonction responsable de gérer les inputs du programme.

La fonction responsable de la gestion des inputs du programme est *"TowerSimulation::create_keystrokes()"*.

Sur quelle touche faut-il appuyer pour ajouter un avion ? Il faut appuyer sur la touche "C".

Comment faire pour quitter le programme ? Il faut appuyer sur la touche "X" ou "Q".

A quoi sert la touche 'F' ? Elle sert à passer en mode plein écran.

Ajoutez un avion à la simulation et attendez.

Quel est le comportement de l'avion ? L'avion commence par atterrir et va se garer au terminal, et ensuite il reprend son service et ainsi de suite.

Quelles informations s'affichent dans la console ?

Les informations qui s'affichent dans la console sont:

-les informations sur l'atterrissage et décollage des avions.

-les informations sur la fin et reprise de service des avions.

Ajoutez maintenant quatre avions d'un coup dans la simulation.

Que fait chacun des avions ?

Les avions ont le même comportement sauf que si les trois terminaux sont occupés, l'avion restant continuera de voler jusqu'à libération d'un des terminaux.

Objectif B :

Listez les classes du programme à la racine du dossier src/.

Pour chacune d'entre elle, expliquez ce qu'elle représente et son rôle dans le programme.

- **Aircraft :** Représente un avion, elle peut dessiner un avion, le faire se déplacer et d'avoir les propriétés de l'avion.

- **Airport :** Représente un aéroport, elle permet de créer un aéroport avec une liste de terminal et une tour de contrôle.
- **AirportType :** Représente le type d'un aéroport, contient les coordonnées importantes du centre de chaque aéroport comme le début et la fin des runways. Chaque AirportType peut générer des chemins pour atterrir et décoller.
- **AircraftTypes :** Représente le type d'un avion, elle stocke les limites de vitesse et d'accélération mais aussi de la texture. Il y a 3 types prédéfinis.
- **Terminal :** Représente un terminal, elle gère le débarquement d'un avion, chaque terminal ne peut gérer le débarquement que d'un seul avion à la fois.
- **TowerSimulation :** Classe qui s'occupe de la gestion de la simulation (création de l'aéroport, création des avions etc.).
- **Tower :** Représente la tour de contrôle. S'occupe de donner des instructions aux avions (par exemple si un terminal est libre, Tower donne l'autorisation à un avion d'atterrir etc.).
- **Waypoint :** Représente un point du chemin d'un avion, c'est un Point3D contenant l'information de si ce point se trouve au sol, sur un terminal ou en l'air.
- **Displayable :** Classe abstraite qui forme la base pour tous les objets pouvant être affichés sur l'écran.
- **DynamicObject :** Classe abstraite qui forme la base de tous les objets pouvant se déplacer.
- **Texture2D :** Représente une texture qui sera affichée avec Texture2D::draw.
- **Image :** Classe permettant la gestion des octets d'une image dans la mémoire qui sera utilisé par Texture2D.
- **MediaPath :** Classe permettant de gérer l'accès aux PNG.

Pour les classes `Tower`, `Aircaft`, `Airport` et `Terminal`, listez leurs fonctions-membre publiques et expliquez précisément à quoi elles servent.

Aircraft:

- ➔ ***get_flight_num*:** Renvoie l'identifiant du vol.
- ➔ ***distance_to*:** Renvoie la distance de l'avion par rapport à un point.
- ➔ ***display*:** Affiche l'avion.
- ➔ ***move*:** Permet de déplacer l'avion.

Airport:

- ➔ ***get_tower:*** Renvoie la tour de contrôle de l'aéroport.
- ➔ ***display:*** Affiche l'aéroport.

➔ ***move:*** Met à jour les terminaux de l'aéroport par rapport au contexte.

Terminal:

➔ ***in_use:*** Permet de savoir si le terminal est utilisé ou pas.

➔ ***is_servicing:*** Permet de savoir si le terminal a fini l'entretien d'un avion ou non. (vrai si l'entretien n'est pas fini et faux sinon).

➔ ***assign_craft:*** Assigne un avion au terminal et donc ce terminal devient occupé.

➔ ***start_service:*** Commence l'entretien d'un avion (affichage du message "now servicing" suivi de l'identifiant de l'avion).

➔ ***finish_service:*** Libère le terminal une fois l'entretien de l'avion fini.

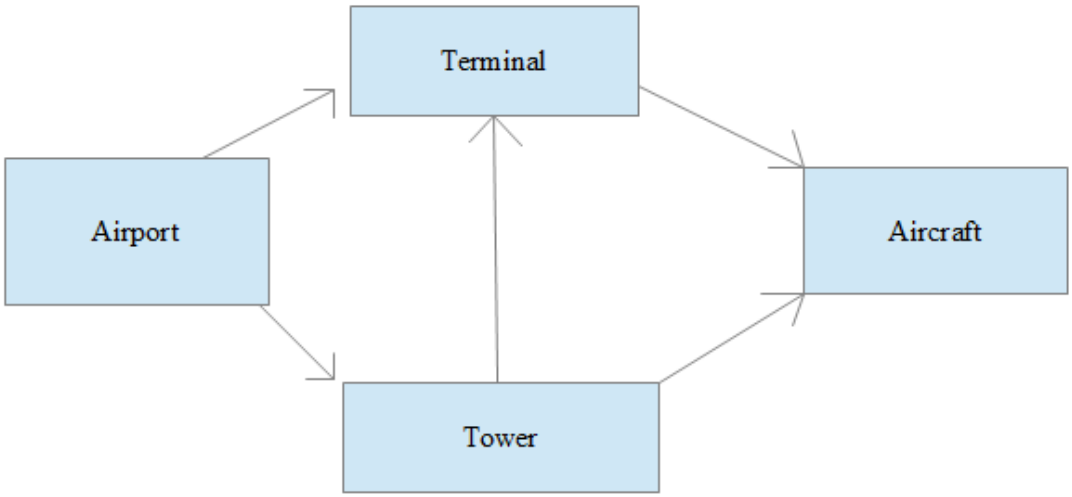
➔ ***move:*** Permet d'actualisation du terminal.

Tower:

➔ **get_instructions:** Elle décide du chemin que doit suivre un avion et enregistre un avion si un des terminaux est libre.

➔ **arrived_at_terminal:** Vérifie qu'un avion est bien arrivé à un terminal et lance l'entretien de l'avion.

Réalisez ensuite un schéma présentant comment ces différentes classes interagissent ensemble.



Quelles classes et fonctions sont impliquées dans la génération du chemin d'un avion ?

Depuis la méthode *Aircraft::move()* : si *WaypointQueue* est vide, on fait appel à *Tower::get_instruction()*.

Si l'avion n'est pas dans un terminal, *Tower::get_instruction()* fait appel à *Airport::reserve_terminal* qui va réserver un terminal pour l'avion.

Si un terminal est libre, *Airport::reserve_terminal* renvoie une séquence de waypoints jusqu'au terminal libre ainsi que le numéro du terminal, en revanche la fonction renvoie un vector vide et n'importe quel nombre si aucun terminal n'est libre.

Cela se fait grâce à l'appel à *Terminal::assign_craft()* qui va assigner l'avion au terminal correspondant.

Quel conteneur de la librairie standard a été choisi pour représenter le chemin ? Expliquez les intérêts de ce choix.

Une *std::deque* est utilisée pour représenter le chemin.

Les intérêts de ce choix sont que cela permet l'ajout d'éléments à la fin de la queue ainsi que la récupération des éléments au début de la queue et tout cela en complexité constante.

Objectif C :

1) Déterminez à quel endroit du code sont définies les vitesses maximales et accélération de chaque avion.

Le Concorde est censé pouvoir voler plus vite que les autres avions.

Modifiez le programme pour tenir compte de cela.

Elles sont définies dans *"aircraft_types.hpp"* dans la structure *"AircraftType"* avec les champs *max_ground_speed, max_air_speed, max_accel*.

On change les valeurs de l'avion censées représenter l'avion Concorde, de telle sorte que sa vitesse et accélération max soient supérieures à celle des autres avions.

2) **Identifiez quelle variable contrôle le framerate de la simulation.**

La variable qui contrôle le framerate de la simulation est *DEFAULT_TICKS_PER_SEC* dans le fichier *config.hpp*.

Essayez maintenant de mettre en pause le programme en manipulant ce framerate. Que se passe-t-il ? Le programme crash car une division par 0 est effectuée.

3) **Identifiez quelle variable contrôle le temps de débarquement des avions et doublez-le.**

La variable qui contrôle le temps de débarquement des avions est *SERVICE_CYCLES* dans le fichier *config.hpp*.

4) **A quel endroit pouvez-vous savoir que l'avion doit être supprimé ?** Dans la fonction *timer* de *opengl_interface*.

Pourquoi n'est-il pas sûr de procéder au retrait de l'avion dans cette fonction ?

Car on risque d'avoir une erreur de segmentation.

A quel endroit de la callstack pourriez-vous le faire à la place ?

Dans la fonction *move* de *dynamic_object.hpp*.

Que devez-vous modifier pour transmettre l'information de la première à la seconde fonction ?

Un booléen qui renverra false si l'avion doit être retiré.

On ajoute un attribut *"service_done"* qui indique si l'avion est déjà passé par un terminal

(ajout dans *aircraft.hpp*). Il vaudra false au départ et passera à true dans *Tower::get_instruction* après service effectué.

5) **Lorsqu'un objet de type `Displayable` est créé, il faut ajouter celui-ci manuellement dans la liste des objets à afficher.**

Il faut également penser à le supprimer de cette liste avant de le détruire.

Faites en sorte que l'ajout et la suppression de `display_queue` soit "automatiquement gérée" lorsqu'un `Displayable` est créé ou détruit.

On modifie le constructeur et le destructeur de *Displayable* pour respectivement ajouter des objets qui doivent être afficher à la *display_queue* mais aussi supprimer des objets contenus dans cette même *display_queue*.

On rend le champs *display_queue* static pour pouvoir y accéder depuis le constructeur et le destructeur de la classe *Displayable*.

Pourquoi n'est-il pas spécialement pertinent d'en faire de même pour `DynamicObject` ?

Parce que les objets qui sont des *DynamicObject* sont aussi des objets *Displayable* (*Aircraft* et *Airport* étendent *DynamicObject* et *Displayable*).

6) **La tour de contrôle a besoin de stocker pour tout `Aircraft` le `Terminal` qui lui est actuellement attribué, afin de pouvoir le libérer une fois que l'avion décolle.**

Cette information est actuellement enregistrée dans un `std::vector<std::pair<const Aircraft*, size_t>>` (size_t représentant l'indice du terminal).

Cela fait que la recherche du terminal associé à un avion est réalisée en temps linéaire, par rapport au nombre total de terminaux.

Cela n'est pas grave tant que ce nombre est petit, mais pour préparer l'avenir, on aimerait bien remplacer le vector par un conteneur qui garantira des opérations efficaces, même s'il y a beaucoup de terminaux.

Modifiez le code afin d'utiliser un conteneur STL plus adapté. Normalement, à la fin, la fonction `find_craft_and_terminal(const Aicraft&)` ne devrait plus être nécessaire.

On utilisera comme conteneur STL *std::map<const Aircraft*,size_t>* plutôt que *std::vector<std::pair<const Aircraft*,size_t>>*.

Ce conteneur contiendra pour chaque avion l'index du terminal qui lui est associé, les accès se font en complexité O(1).

Objectif D :

1) **Comment a-t-on fait pour que seule la classe `Tower` puisse réserver un terminal de l'aéroport ?** La classe *Tower* est la seule classe qui possède les informations sur le terminal dans lequel un avion est enregistré.

2) **En regardant le contenu de la fonction `void Aircraft::turn(Point3D direction)`,**

pourquoi selon-vous ne sommes-nous pas passer par une référence ?

Car en passant par une référence il aurait fallu créer une variable qu'on aurait passé à *turn* or cela n'est pas nécessaire car *turn* est seulement utilisé dans *turn_to_waypoint* et le paramètre n'est pas « gros » donc le coût n'est pas grand.

Pensez-vous qu'il soit possible d'éviter la copie du `Point3D` passé en paramètre ?

Oui, en utilisant le passage par une référence.

Task 1

Gestion mémoire :

Si à un moment quelconque du programme, vous souhaitiez accéder à l'avion ayant le numéro de vol "AF1250", que devriez-vous faire ?

On doit parcourir la liste des avions en comparant à chaque fois le numéro que l'on recherche aux numéros des avions de la liste.

Objectif 1 :

A - Choisir l'architecture

Pour trouver un avion particulier dans le programme, ce serait pratique d'avoir une classe qui référence tous les avions et qui peut donc nous renvoyer celui qui nous intéresse.

Vous avez 2 choix possibles :

- créer une nouvelle classe, `AircraftManager`, qui assumera ce rôle,

- donner ce rôle à une classe existante.

Réfléchissez aux pour et contre de chacune de ces options.

Le deuxième choix fonctionnerait mais la premier choix est plus compréhensible pour respecter le principe de responsabilité unique (single responsability principle)

B - Déterminer le propriétaire de chaque avion

1. Qui est responsable de détruire les avions du programme ?

La fonction *timer* contenue dans le fichier *opengl_interface.cpp* est responsable de détruire les avions du programme.

2. Quelles autres structures contiennent une référence sur un avion au moment où il doit être détruit ?

Les deux structures qui contiennent une référence sur un avion au moment de sa destruction sont *GL::Displayable::display_queue* et *GL::move_queue*.

3. Comment fait-on pour supprimer la référence sur un avion qui va être détruit dans ces structures ?

Il faut utiliser la fonction "*erase*" car les avions sont référencés par un *unique_ptr* dans ces structures.

4. Pourquoi n'est-il pas très judicieux d'essayer d'appliquer la même chose pour votre

`AircraftManager` ? Car on risque de perdre la responsabilité unique de la classe.

Objectif 2 :

A - Création d'une factory

- Création du fichier *aircraft_factory.hpp* contenant la classe AircraftFactory.
- Ajout des fonctions *create_aircraft*, *create_random_aircraft*, *airlines, aircraft_types* ainsi que des attributs *aircraft_types* et *airlines*.
- On ajoute dans le fichier *aircraft_manager.hpp* un attribut de type *AircraftFactory* qu'on utilisera pour créer des avions (On met tout le code concernant la gestion des avions dans une même classe qui est *AircraftManager*).
- On remplace toutes les fonctions concernant la création d'avion du fichier *tower_sim.hpp* par celles de l'attribut de type *AircraftManager*(qui appellera quant à elle les fonctions de la classe *AircraftFactory*).
- On retire/met en commentaire toutes les fonctions désormais inutilisées.

B – Conflits

Faites maintenant en sorte qu'il ne soit plus possible de créer deux fois un avion avec le même numéro de vol.

On vérifie à chaque fois qu'un numéro de vol est généré, que celui ci n'a pas déjà été crée en vérifiant s'il n'est pas présent dans le conteneur des numéros de vol.

Task 2

Objectif 1 :

A - Structured Bindings

`TowerSimulation::display_help()` est chargé de l'affichage des touches disponibles.

Dans sa boucle, remplacez `const auto& ks_pair` par un structured binding adapté.

On remplace `*const auto& ks_pair*` par `*const auto& [key, value]*`.

B - Algorithmes divers

2. Pour des raisons de statistiques, on aimerait bien être capable de compter tous les avions de chaque airline.

A cette fin, rajoutez des callbacks sur les touches `0`..`7` de manière à ce que le nombre d'avions appartenant à `airlines[x]` soit affiché en appuyant sur `x`.

Rendez-vous compte de quelle classe peut acquérir cet information. Utilisez la bonne fonction de `<algorithm>` pour obtenir le résultat.

On utilise une fonction `*number_aircraft_by_airline*` qui permet d'avoir cette information dans la classe `*AircraftManager*`.

Cette fonction est ajoutée dans la fonction `*TowerSimulation::create_keystrokes()*`.

C - Relooking de Point3D

La classe `Point3D` présente beaucoup d'opportunités d'appliquer des algorithmes.

Particulièrement, des formulations de type `x() = ...; y() = ...; z() = ...;` se remplacent par un seul appel à la bonne fonction de la librairie standard.

Remplacez le tableau `Point3D::values` par un `std::array` et puis,

remplacez le code des fonctions suivantes en utilisant des fonctions de `<algorithm>` / `<numeric>`:

1. `Point3D::operator*=(const float scalar)`

2. `Point3D::operator+=(const Point3D& other)` et `Point3D::operator-=(const Point3D& other)`

3. `Point3D::length() const`

On utilise la fonction `*std::transform*` pour les fonctions `*Point3D::operator**=`, `*Point3D::operator+=*` et `*Point3D::operator-=*` et la fonction `*std::inner_product*` pour `*Point3D::length()*`.

Objectif 2:

C - Minimiser les crashs

Assurez-vous déjà que le conteneur `AircraftManager::aircrafts` soit ordonnable (`vector`, `list`, etc).

Au début de la fonction `AircraftManager::move` (ou `update`), ajoutez les instructions permettant de réordonner les `aircrafts` dans l'ordre défini ci-dessus.

Le conteneur '*AircraftManager::aircrafts'* est un *vector.*

On utilise la fonction '*std::sort'* pour trier les avions.

D - Réapprovisionnement

1. **Ajoutez une fonction `bool Aircraft::is_low_on_fuel() const`, qui renvoie `true` si l'avion dispose de moins de `200` unités d'essence.**

Modifiez le code de `Terminal` afin que les avions qui n'ont pas suffisamment d'essence restent bloqués.

Testez votre programme pour vérifier que certains avions attendent bien indéfiniment au terminal.

Si ce n'est pas le cas, essayez de faire varier la constante `200`.

On ajoute la condition que l'avion n'est pas à un niveau de carburant faible dans la fonction '*bool Terminal*::*move'.*

2. **Dans `AircraftManager`, implémentez une fonction `get_required_fuel`, qui renvoie la somme de l'essence manquante (le plein, soit `3'000`, moins la quantité courante d'essence) pour les avions vérifiant les conditions suivantes :**

\- l'avion est bientôt à court d'essence

\- l'avion n'est pas déjà reparti de l'aéroport.

On utilise la fonction *std::for_each* pour parcourir le conteneur *aircrafts* et faire la somme de l'essence manquante de chaque avion presque à court d'essence et qui est dans un terminal.

3. **Ajoutez deux attributs `fuel_stock` et `ordered_fuel` dans la classe `Airport`, que**

vous initialiserez à 0.

Ajoutez également un attribut `next_refill_time`, aussi initialisé à 0.

Enfin, faites en sorte que la classe `Airport` ait accès à votre `AircraftManager` de manière à pouvoir l'interroger.

On ajoute un attribut '*AircraftManager*' dans '*Airport'* qu'on initialise dans le constructeur de '*Airport'*.

Task 3

Objectif 1 :

Actuellement, quand un avion s'écrase, une exception de type `AircraftCrash` (qui est un alias de `std::runtime_error` déclaré dans `config.hpp`) est lancée.

1. **Faites en sorte que le programme puisse continuer de s'exécuter après le crash d'un avion. Pour cela, remontez l'erreur jusqu'à un endroit approprié pour procéder à la suppression de cet avion (assurez-vous bien que plus personne ne référence l'avion une fois l'exception traitée). Vous afficherez également le message d'erreur de l'exception dans `cerr`.**

On catch dans *AircraftManager::move* le fait qu'un avion se crash, on affiche le message d'erreur et renvoie *true* pour que l'avion soit supprimé.

2. Introduisez un compteur qui est incrémenté chaque fois qu'un avion s'écrase. Choisissez une touche du clavier qui n'a pas encore été utilisée (`m` par exemple ?) et affichez ce nombre dans la console lorsque l'utilisateur appuie dessus.

Avant de renvoyer true dans la fonction citée précédemment on incrémente le nombre d'avion crashé.

Task 4

Objectif 1 :

2. Modifiez `Aircraft::add_waypoint` afin que l'évaluation du flag ait lieu à la compilation et non à l'exécution.

Il faut passer le flag en template plutôt qu'en paramètre de la fonction.

Que devez-vous changer dans l'appel de la fonction pour que le programme compile ?

On doit mettre "front" entre chevrons tel que `add_waypoint<front>(wp)`.

Objectif 2 :

4. **Dans la fonction `test_generic_points`, essayez d'instancier un `Point2D` avec 3 arguments.**

Que se passe-t-il ? Une erreur s'affiche indiquant que le nombre de paramètres est supérieur au nombre qu'il devrait être (ici 2).

Comment pourriez-vous expliquer que cette erreur ne se produise que maintenant ?

Cela peut s'expliquer par le fait qu'avant le nombre d'arguments n'était pas spécifié contrairement à maintenant.

5. **Que se passe-t-il maintenant si vous essayez d'instancier un `Point3D` avec 2 arguments ?** Le programme ne plante pas car il utilise le constructeur à 2 paramètres (celui de Point2D).

Points Positifs:

- Projet dirigiste ce qui permet d'avancer étape par étape.
- Architecture préexistante ce qui nous oblige à nous approprier le code avant de pouvoir l'améliorer/modifier.
- Qu'on ait pas à s'occuper de la partie graphique.
- Le fait qu'on puisse avancer dans le projet lors des TD.

Points Négatifs:

- Le fait que ce soit en solo et non en groupe.
- Un peu difficile par moment pour quelqu'un qui découvre le C++.

Ce que j'ai appris:

- Les bases du langage C++.
- Comprendre du code C++ et l'améliorer/modifier.