CPP_Learning_Project

KHELIFI Allan

Master 1 Informatique - Groupe 1

Table des matières

TASK 0 - Se familiariser avec l'existant

1.1 A - Exécution

Allez dans le fichier tower_sim.cpp et recherchez la fonction responsable de gérer les inputs du programme. Sur quelle touche faut-il appuyer pour ajouter un avion ? Comment faire pour quitter le programme ? A quoi sert la touche 'F' ?

La fonction responsable de la gestion des inputs est TowerSimulation : :create_keystrokes(). Quitter le programme se fait à l'aide des touches 'x' et 'q'. La touche 'f' passe le programme en plein écran.

Ajoutez un avion à la simulation et attendez. Que est le comportement de l'avion ? Quelles informations s'affichent dans la console ?

Le comportement de l'avion est, lors de son entrée, d'aller vers l'aéroport et une fois atteri, il se pose sur un terminal avant de repartir de la simulation pour éventuellement revenir et répéter le même comportement.

Ajoutez maintenant quatre avions d'un coup dans la simulation. Que fait chacun des avions ?

Les trois premiers avions suivent le comportement décrit précedemment, le quatrième volera autour de l'aéroport tant qu'aucun terminal ne sera libre.

1.2 B - Analyse du code

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.

Les différentes classes sont **Aircraft**, **Airport**, **AirportType**, **Terminal**, **Tower**, **TowerSimulation**, **Waypoint**.

— Aircraft Objet représentant un avion
— Airport Objet représentant l'aéroport (ici, il n'y en a qu'un)
— AirportType Objet représentant le type d'aéroport
— Terminal Objet représentant un des trois terminaux où les avions peuvent atterrir
— Tower Objet représentant la tour de contrôle, elle guide les avions en leur donnant des instructions à suivre
— TowerSimulation Classe principale qui gère le programme en son intégralité
— Waypoint Objet représentant une direction qui l'avion doit suivre : décoller, aller à un terminal, autre

Pour les classes Tower, Aircaft, Airport et Terminal, listez leurs fonctions-membre publiques et expliquez précisément à quoi elles servent. Réalisez ensuite un schéma présentant comment ces différentes classes intéragissent ensemble.

Aircraft

```
// aircraft.hpp
const std::string& get_flight_num() const;
float distance_to(const Point3D& p) const;
void display() const override;
void move() override;
```

- get_flight_num() renvoie l'identifiant du vol
- distance_to(p) renvoie la distance de l'avion au point p
- display() permet de dessiner l'avion
- move() permet le déplacement de l'avion

Tower

```
// tower.hpp
WaypointQueue get_instructions(Aircraft& aircraft);
void arrived_at_terminal(const Aircraft& aircraft);
```

- get_instructions() défini le fonctionnement d'un avion. S'il n'est pas dans un terminal, s'il y a de la place dans un terminal : lui assigner un, faire un cercle autour de l'aéroport sinon. S'il est dans un terminal, lui assigner un chemin pour repartir s'il n'est pas en service.
- arrived_at_terminal() vérifie que l'avion est bien arrivé au terminal (qu'il l'occupe)

Airport

```
// airport.hpp
Tower& get_tower();
void display() const override;
void move() override;
```

- get_tower() getter pour un objet Tower
- display() permet de dessiner l'aéroport
- move() permet le "déplacement" d'un aéroport. Ici le déplacement correspond à "l'actualisation" de l'objet à chaque frame

Terminal

```
// terminal.hpp
bool in_use() const;
bool is_servicing() const;
void assign_craft(const Aircraft& aircraft);

void start_service(const Aircraft& aircraft);

void finish_service();

void move() override;
```

— in_use() renvoie est-ce que le terminal contient un avion ou non

— is_servicing() renvoie est-ce que le terminal effectue l'entretien d'un avion ou non

— assign_craft() assigne un avion au terminal et rend le terminal occupé.

— start_service(aircraft) commence le service (= entretien) d'un avion.

— finish_service() libère le terminal une fois le service d'un aéroport terminé

— move() permet le "déplacement" d'un terminal. Ici le déplacement correspond à "l'actualisation" de l'objet à chaque frame

Quelles classes et fonctions sont impliquées dans la génération du chemin d'un avion ? Quel conteneur de la librairie standard a été choisi pour représenter le chemin ? Expliquez les intérêts de ce choix.

Les classes impliquées dans la génération du chemin d'un avion sont Aircraft (pour l'avion), Tower (pour guider l'avion) et Waypoint (son chemin). Toute la résolution est effectuée dans Tower : :get_instructions(). Le conteneur utilisé ici est **std : :deque** qui, pour les opérations recherchées ici, opère en complexité constante.

1.3 C - Bidouillons !

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.

C'est dans le fichier **aircraft_types.hpp**, dans la méthode init_aircraft_types() que sont définies les propriétés des avions. Dans celle-ci on peut modifier, pour chaque type d'avion, sa vitesse au sol, en l'air et son accélération maximale.

Identifiez quelle variable contrôle le framerate de la simulation. Ajoutez deux nouveaux inputs au programme permettant d'augmenter ou de diminuer cette valeur. Essayez maintenant de mettre en pause le programme en manipulant ce framerate. Que se passe-t-il ? Fixez le problème.

La variable DEFAULT_TICKS_PER_SEC contenue dans le fichier **config.hpp** permet le contrôle du framerate de la simulation. Si l'on essaie de rendre le framerate inférieur ou égal à 0 le programme plante.

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

La variable service_progress utilisée avec la constante SERVICE_CYCLES (dans **config.hpp**) contrôle le temps de débarquement des avions.

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

C'est moins pertinent puisque si l'objet est aussi un Displayable il faut pouvoir le retirer aussi, ce qui n'est pas le role du destructeur d'un DynamicObject.

1.4 D - Théorie

Comment a-t-on fait pour que seule la classe Tower puisse réserver un terminal de l'aéroport ?

En gardant l'information des terminaux réservés dans la classe **Tower**, avec son champ Aircraft-ToTerminal.

En regardant le contenu de la fonction void Aircraft : :turn(Point3D direction), pourquoi selon-vous ne sommes-nous pas passer par une réference ? Pensez-vous qu'il soit possible d'éviter la copie du Point3D passé en paramètre ?

On est pas passés par référence puisque les méthodes appelés dans la méthode turn() modifient la direction passée en argument.

TASK 1 - Gestion des ressources

2.1 Analyse de la gestion des avions

La création des avions est aujourd'hui gérée par les fonctions TowerSimulation : :create_aircraft et TowerSimulation : :create_random_aircraft. Chaque avion créé est ensuite placé dans les files GL : :display_queue et GL : :move_queue.

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

Si l'on souhaite accéder à un tel avion il faut parcourir l'intégralité de la move_queue (ou bien de la display_queue) pour y trouver l'avion.

2.2 B - Déterminer le propriétaire de chaque avion

Qui est responsable de détruire les avions du programme ? (si vous ne trouvez pas, faites/continuez la question 4 dans TASK_0)

Le responsable doit être l'objet lui-même, il détient la responsabilité de se détruire.

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

La move_queue et la display_queue contiennent une référence sur cet avion lorsqu'il va être détruit.

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

Il nous faut parcourir ces structures et utiliser les méthodes std : :remove et move_queue.erase() (respectivement display_queue.erase()).

```
GL::display_queue.erase(std::remove(GL::display_queue.begin(),
    GL::display_queue.end(), item));
GL::move_queue.erase(std::remove(GL::move_queue.begin(), GL::move_queue.end(), item));
```

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

Il vaut mieux éviter les fuites en cas d'oubli de retrait de référence dans une structure en passant l'ownership à la classe **AircraftManager**.

TASK 4 - Templates

3.1 Objectif 1 - Devant ou derrière ?

Modifiez Aircraft : :add_waypoint afin que l'évaluation du flag ait lieu à la compilation et non à l'exécution. Que devez-vous changer dans l'appel de la fonction pour que le programme compile ?

TODO

BONUS En utilisant GodBolt, comparez le code-assembleur généré par les fonctions suivantes :

```
int minmax(const int x, const int y, const bool min) {
    return x < y ? (min ? x : y) : (min ? y : x);
}

//////////////////////////////////////////////////

template<bool min>
int minmax(const int x, const int y){
    return x < y ? (min ? x : y) : (min ? y : x);
}
```

On observe que pour la fonction templatée aucun code assembleur n'est généré, le code est donc généré pour chaque fonction templatée basée sur cette template.

Architecture

4.1 Choix d'architecture

4.1.1 AircraftManager et AircraftFactory

La réalisation de l'AircraftManager pendant la TASK1 était à des fins d'ownership avant tout. En réalisant cette classe on possède un simple point de contrôle des avions : on sait où les trouver, on sait où ils sont gérés.

L'AircraftFactory elle permet la création des avions qui seront envoyés à l'AircraftManager ensuite. La réalisation du design pattern Manager permet d'éviter les dangling references et toutes fuites quelconques en sachant où manipuler les avions et le pattern Factory permet d'avoir un point simple de création des avions.

4.1.2 AircraftCrash

Cette classe a été réalisée selon le bonus de la TASK3, pour remplacer les exceptions existantes par une personnallisée selon la définition de la classe **AircraftCrash**. Elle permet d'avoir des exceptions plus détaillées pour chaque cas de crash d'un avion, un exemple d'appel de celle-ci :

```
// dans aircraft.hpp
throw AircraftCrash { flight_number, pos, speed, " has no fuel remaining"s };
```

4.1.3 Architecture globale

L'architecture finale du projet est basée sur les deux design patterns implémentés. La gestion du programme à toujours lieu dans la classe **TowerSimulation** qui contient un objet **AircraftManager** et un objet **AircraftFactory**. La Factory permet efficacement d'instancier de nouveaux avions tandis que le Manager en permet le contrôle, la gestion de ceux-ci. Par exemple : lors de l'approvisionnement en carburant, l'information sur le caburant manquant pour satisfaire tout les avions, le déplacement de ceux-ci, c'est un lien pratique vers les avions.

4.2 Apprentissage

Le projet m'a permi d'acquérir une bien meilleure compréhension du langage, comment mieux identifier les bugs, appliquer les différentes parties du cours. C'était à mes yeux plutôt complet et bien qu'être coincé sur certaines parties était plus difficile par moments, j'en garde quand même un très bon ressenti. Récupérer mes connaissances en C et les mélanger et approfondir à travers le C++ c'était pour le coup très enrichissant !