Projet C++ 2021
Tower Control

Compte rendu

Projet de C++ - Master 1
Damien LACOMBE
25 Avril 2021

Table des matières

Présentation

Le projet est un simulateur d'aéroport.

On va pouvoir simuler la création d'avions, leurs atterrissages, leurs nivaux de carburant etc.

Nécessité

Le projet est prévu pour une architecture Linux.

Les logiciels suivants sont nécessaire pour compiler et exécuter le programme :

- Compilateur capable de compiler du C++17 ;

- freeglut ;

- OpenGL version 1,1 ou supérieur ;

- Cmake version 3 ou supérieur (pour générer le projet.

Utilisation

Pour utiliser le projet, il faut le générer avec Cmake.

Une fois le simulateur lancé il suffit d'utiliser les commandes suivantes (touches clavier) pour pouvoir utiliser le simulateur :

- touche x ou q : Quitte le simulateur ;

- touche p : Met en pause la simulation ;

- touche f : Met le simulateur en pleine écran (ou enlève le mode plein écran) ;

- touche + : Zoom sur l'aéroport ;

- touche - : Dé-zoom sur l'aéroport ;

- touche c : Créer un avion (aléatoire) ;

- touche m : Affiche le nombre d'avion qui se sont écrasé ;

- touche a : Augmente la framrate de la simulation ;

- touche z : Diminue la framrate de la simulation ;

- touche numérique (de 0 à 7) : Indique le nombre d'avion selon la ligne donnée.

Architecture du projet :

Displayable - c'est une classe abstraite qui forme la base pour toutes les choses qui peuvent être dessinée sur l'écran. La classe contient une coordonné z qui permet de trier les objets de cette classe.

DynamicObject - une autre classe abstraite qui forme la base pour tout les choses qui peuvent "bouger".

opengl-interface - pas de classe ici, juste quelques fonctions nécessaires pour interagir avec **OpenGL**; on remarque, par contre, la fonction *timer* dans laquelle tout les objets dans la *move_queue* ont leur *move()* appelé.

Texture2D - une texture qui contient un pointeur vers un *img::Image* (qui contient les octets en vrac de l'image); la texture peut être affichée avec *Texture2D::draw*.

Image - une classe qui gère des octets d'un image dans la mémoire pour être utilisé avec **Texture2D**

MediaPath - une classe qui gère l'acces aux PNGs qui vont avec le code.
stb_image - pas de classe ici, c'est en fait une bibliothèque C qui sait lire les PNGs correctement.

Point2D / **Point3D** - classes qui gèrent des maths entre points dans l'espace 2D et 3D.
C'est dernière classes seront par la suite généralisées dans une classe **Point.**

Waypoint - un point sur le chemin d'un avion, c'est juste un **Point3D** avec l'information si ce point se trouve au sol, chez un terminal ou dans l'air; ici, on voit aussi qu'un "chemin" (*WaypointQueue*) est un *deque* de **Waypoints**.

Runway - stocke le début et le fin d'une piste de décollage.

Terminal - classe qui gère le débarquement d'un avion; chaque Terminaux peuvent débarquer qu'un seul avion à la fois.

Aircraft - un avion qui peut :
- être dessiner (*Displayable*)
- bouger (*DynamicObject*);
Chaque avions peuvent retourner leur "*flight number*" ainsi que leur distances à un point donné.

AircraftType - le type d'un avion stocke des limites de vitesse et accélération ainsi que la texture; il y a 3 types prédéfinit.

Airport - gère l'aéroport, contient les terminaux et le *Tower* (seulment son *friend class*)**.**

Tower peut réserver des terminaux et demander un chemin de décollage.

AirportType - contient les coordonnés importantes (relatives au centre de chaque aéroport) comme le début/fin des *runways* (il peut en avoir plusieurs); chaque **AirportType** peut générer des chemins

pour atterrir et pour décoller.

Tower - classe qui gère la fonctionnalité de la tour de contrôle; des avions peuvent demander des nouvelles instructions ainsi qu'indiquer qu'ils sont arrivés à leur Terminal.
Chaque Tower contient une affectation des avions aux terminaux. Si un avion X demande d'atterrir à un moment quand tout les Terminaux de l'aéroport sont affectés, alors le Tower retourne un "cercle" autour de l'aéroport pour que X re-demande quand il a fini son cercle.

TowerSimulation - une classe pour la gestion de la simulation: création de l'aéroport, affichage de l'usage sur la ligne de commande, création des avions, etc.

config - pas de classe ici, mais des constantes qui déterminent quelques comportements de la simulation, par exemple le nombre intervalles nécessaire pour débarquer un avion.

Task 0

- Les vitesses et accélération des avions sont déterminée dans **AircrafType**, il faut changer la valeur du Concorde dans cette classe (par la suite ces valeurs seront dans la *factory*).
- Il faut faire attention à ce que la valeur du framerate ne passe pas en dessous de 0 car cela pourrait faire buguer entièrement le programme. La vérification se fait au moment où l'on veut diminuer la valeur de celle-ci.
- Pour mettre le jeu en pause sans passer par le framerate, j'ai ajouter un booléen qui de base est à false. Lorsque ce booléen passe à true, on empêche les mouvements des avions.

  Afin de mettre en pause la simulation, il faut appuyer sur la touche p (et appuyer une nouvelle fois pour relancer la simulation).
- On sait qu'unn avion doit être détruit via **Tower::get_instructions.**

  La destruction doit être faites après les appels sur l'Aicraft afin de ne pas avoir un pointeur null. En revanche, on n'est pas obligé de la faire pour **DynamicObject** car les avoions ne peuvent plus bouger lorsqu'ils sont supprimés (un chemin sans *Waypoints).*

Conclusion

Cette task nous a permis de prendre en main le projet, d'appréhender sons fonctionnement et son architecture afin de pouvoir plus facilement continuer l'implémentation par la suite.

Task 1

Aircraft Manager

Une classe **AircraftManager** est créée afin de respecter le principe d'OwnerShip.

Cette classe stocke tous les avions afin de pouvoir les créer et les supprimés plus facilement.

C’est également cette classe qui va mettre à jour les avions :

- Met à jour leurs dépllacement ;
- Met à jour leur fuel (avec la task 2) ;

On fait ainsi au début de l’update un trie sur les avions (pour prioriser les avions en manque de carburant et minimiser les crashs).

Puis on les supprime s’ils doivent-être supprimés.

Aircraft Factory

Afin de simplifier la création d’avion, une factory est créé. Ce qui évite également l’usage de variables globales comme les numéros de vols ou les types.

Cette classe veillera également au faites que deux avions ne peuvent avoir le même numéro de vol.

Conclusion

Cette task nous à permis de faire respecter le principe d’OwnerShip et de simplifier par conséquent l’utilisation / la création d’avions.

Task 2

Cette task est celle qui m’a pris le plus de temps.

Elle n’est pas complète, il manque la partie **Réaprovisionnement**, le problème étant que l’**Airport** doit avoir accès à l’**AircraftManager**, or l’**AircraftManager** implémente l’**AircraftFactory** qui elle-même implémente l’**Airport**.

Donc l’**Airport** ne peut implémenter l’**AircraftManager** car sinon on aurais une boucle d’implémentation (ou boucle de linkage).

Conclusion

Cette task a été la plus compliquer à réaliser, bien savoir comment implémenter les différentes choses sans corrompre se qui est déjà implémenter.

Task 3 et 4

La Task 3 permet (entre autre) de limiter le crashs d’avions en priorisant les avions en manque de carburant dans la liste de l’**AircraftManager**.

La Task 4 nous à permis d’appliquer les notions de **templates**, en faisant par exemple des Points générique (au lieu d’avoir une classe par type de points qu’ils soit 3D ou 2D ou autre).

Cette généralisation n’a pas été des plus intuitives.

Si on essaye d'instancier un Point2D avec trop d'argument, une erreur de compilation se produit.

En revanche si on implémente un Point3D avec pas assez d'arguments, rien ne se produit.

Cette erreur est due au fait que maintenant le nombre d'argument n'est pas spécifié clairement contrairement à avant où l'on avais un type par argument.

Conclusion

Ces deux dernières task étaient plus faciles à implémenter que la deuxième, bien que la notion de **templates** étant nouvelle ce n'était pas non plus intuitifs sur le début surtout pour la généralisation des points.

Git

Le projet est disponible via un dépôt Github.

Les différents commit ne sont pas à prendre dans l'ordre chronologique des implémentation des tasks.

En effet, suite a un problème d'ordinateur le projet à été réimplémenté ce qui fait que certaines task (notamment la deuxième) implémentais des choses qui n'était demande que bien plus tard.

Voici le lien du dépôt :

https://github.com/dlacombe4/CPP_Learning_Project.git

Conclusion

Ce projet était intéressent car il nous à permis de mettre en pratique les notions directement vue en cours.

Le coté progressiste du projet (sous formes de TP filés) est un grands plus car il nous permet d'implémenter au fur et à mesure les notions de cours sans s' éparpiller comme pour un grands projet final.

Le faite que la partie graphiques nous soit fournies dès le début est un grands plus car ça nous à permis de se concentrer sur les autres notions en pouvant tester directement le résultat.

Les moins du projet :

Le faites que le C++ est nouveau pour moi a fait que j'ai eu du mal à appréhender le projet et à bien comprendre au début toutes l'architecture et son fonctionnement.

C'est également ce qui a compliquer l'implémentation de nouvelles choses car on peut facilement s'y perdre (surtout pour la deuxième task).

Pour finir c'était un projet intéressent et enrichissent bien que pas facile à réaliser (à mon goût).