Rapport projet C++

Table des matières

TASK 0 :

A- Exécution

Après l'exécution de notre programme on a le choix entre différent commande.

Avec les touches 'x' et 'q' on peut quitter le programme.

La touche 'c' nous permet d'ajouter un avion.

Les touches '+' ou '-' nous permettent respectivement de zoomer et dézoomer.

La touche 'f' elle sert à mettre notre écran en pleine écran.

Chaque avion dans notre application va atterrir sur la piste si il y a de la place (3 avions maximum en même temps sur la piste d'atterrissage). Il va ensuite rouler sur la piste jusqu'à atteindre l'endroit ou il doit exécuter son services. A la fin de son service il repart et quitte l'aéroport.

Plusieurs messages vont apparaître dans le terminal de l'application :

- Un message quand un avion atterrit ('avion' is now landing…).

- Un message quand il commence à faire son service (now servicing 'avion'…) .

- Un message quand l'avion à fini son service (done servicing 'avion').

- Et un dernier message quand l'avion quitte l'aéroport ('avion' lift off).

B- Analyse du code

Classe

Classes	Rôle
aircraft.cpp	Représente un avion, contient tout les fonctions pour déplacer notre avion et pour l'afficher.
aircraft_types.hpp	Structure de représentation d'un avion et définie trois types d'avions différents.
airport.hpp	Notre classe nous permet de réserver un terminal pour un avion, et de lui donner ces waypoints.
airport_types.hpp	Structure de représentation d'un aéroport, créé le terminal, gère la queue des avions dans l'aéroport et leurs waypoint et construit un aéroport.

config.hpp	Définie tout nos variables globales et constantes dont on a besoin dans le projet.
geometry.hpp	Définie tout les structures d'objet géométrique nécessaire et tout les fonctions de calcules nécessaire à notre projet.
main.cpp	Notre programme principale.
runway.hpp	Définie un runaway par rapport à la position de l'aéroport.
terminal.hpp	Cette classe représente un terminal, elle possède tout les fonctions pour la représenter et pour récupérer les informations nécessaire.
tower.cpp	Structure qui représente une tour de contrôle, elle gère les avions, leurs états leurs actions sur l'aéroport et dans les airs.
tower_sim.hpp	Va créer tout les fonctions principales de notre programme telle que , les touches, la création d'un avion , l'exécution total du programme, l'initialisation d'un aéroport,…
waypoint.hpp	Permet de définir nos différents queue dans notre programme.

Package GL

displayable.hpp	Représente un objet de type 'displayable' qui peut être afficher et contient une valeur 'z' pour connaître sa profondeur lors de l'affichage.
dynamic_object.hpp	Représente un objet de type'dynamic object' il peut être affiché et déplacer.
opengl_interface.hpp	Contient tout les fonctions pour gérer la partie principale, de notre programme telle que les touches , le zoom, ...
texture.hpp	Permet de dessiner nos images.

Package img

image.hpp	Permet de représenter une image et de récupérer ces informations.
media_path.hpp	
stb_image.h	

Classe Aircraft

- Constructeur -> Le constructeur d’ un avion.

- const std::string& get_flight_num() const { return flight_number; } → Renvoie le numéro de vol d’un avion.

- float distance_to(const Point3D& p) const { return pos.distance_to(p); } → Renvoie la distance entre deux points.

- void display() const override; → Affiche les textures de l’avion.

- void move() override; → Permet de déplacer notre avion avec les différents waypoint et d’afficher sur la console ces actions.

Classe Airport

- Twoer& get_tower() {return tower } → Renvoie notre objet ‘tower’.

- void display() const override { texture.draw(project_2D(pos), { 2.0f, 2.0f }); } → Dessine notre aéroport.

- void move() override → Lance le déplacement de nos avions.

Classe Terminal

- bool in_use() const → Nous dit si il y a un avion en action dans le terminal.

- bool is_servicing() const → Nous indique dans quelle service se trouve l’avion.

- void assign_craft(const Aircraft& aircraft) → Assigne un avion au terminal.

- void start_service(const Aircraft& aircraft) → Exécute le lancement des services de l’avion actuellement assigné au terminal.

- void finish_service() → Termine le service notre avion actuelle.

- void move() override → Fais avancer un avion dans la terminal en passant au service suivant.

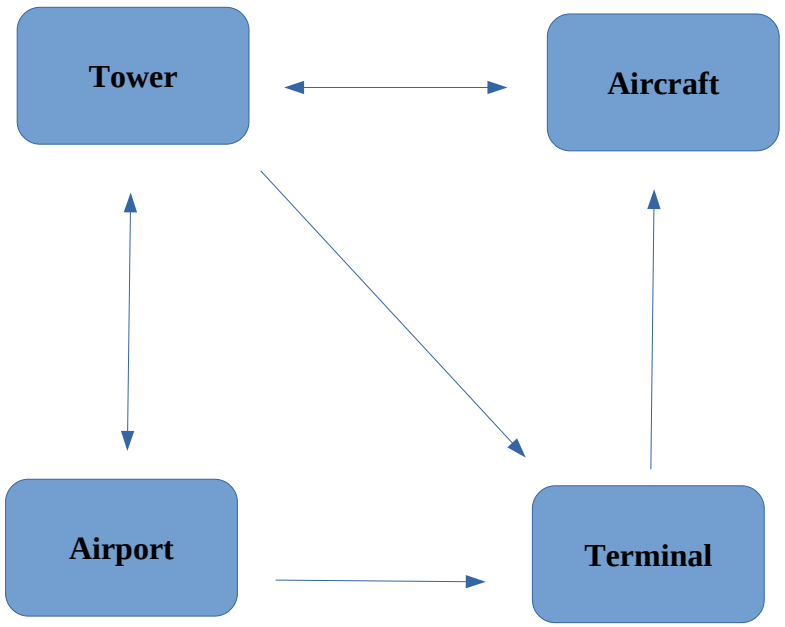
Classe Tower

- WaypointQueue Tower::get_circle() const → Renvoie la suite des positions du voyage de l'avions dans les air.

- WaypointQueue Tower::get_instructions(Aircraft& aircraft) → Renvoie la prochaine instruction pour un avion.

- void Tower::arrived_at_terminal(const Aircraft& aircraft) → Lance le début de l'action de l'avion sur le terminal.

Schéma d'interaction



Questions

La classe ‘aicraft.cpp’ est impliquées dans la génération d’un chemin d’un avion avec ça fonction ‘waypoints = control.get_instructions(*this);’ qui demande a ‘Tower’ des instructions.

La classe ‘tower.cpp’ elle va ensuite demander à l’aéroport de réserver un terminal et retourner le chemin vers cette terminal avec la fonction ‘const auto vp = airport.reserve_terminal(aircraft);’.

Pour représenter un chemin le conteneur ‘std::deque’ de la librairie standard a été choisi, on aurait pu utiliser ‘std::queue’ mais on a choisie ‘std::deque ‘parce qu’on veut rajouter des Waypoints à la fin et au début de la queue.

C- Bidouillons !

1) Les vitesses maximales et accélération de chaque avion sont définies dans la classe ‘aircraft_types’.

2) La variable qui contrôle le framerate de la simulation est ‘ticks_per_sec’ dans la classe ‘opengl_interface’. Si on essaie de réduire au maximum le framerate pour arrêter les avions notre programme crash.

On deux nouvelle fonctionnalité la touche ‘j’ qui permet d’augmenter le framrate et ‘k’ pour le diminuer.

On vérifie que le framerate ne descend pas en dessous de 0 pour éviter d’avoir des valeurs négatives.

3) La variable qui contrôle le temps de débarquement des avions est ‘service_cycle’ dans la classe ‘config.hpp’.

4) On peut savoir si un avion va décoller dans notre classe ‘Aircraft’. On évite de le supprimé l’avion dans cette endroit car il peut exister dans dautre partie de notre code comme dans la ‘move_queu’ ou l’on stocke nos avions. On peut le faire dans notre fonction ‘timer’ de notre classe ‘opengl_interface’ ou l’on s’occupe de déplacer nos avions.

5) On ne peut faire la même chose avec un ‘dynamic object’ car ils ne sont pas const comme les ‘displayable’.

6) On va utiliser une ‘std::map<const Aircraft*, size_t>’, avec pour chaque clé un aircraft qui est associé à une valeur l’indice de son terminal.

D- Théorie

1) Seul la classe ‘Tower’ inclue la classe ‘Terminal’ elle est donc la seul à pouvoir réserver un terminal.

2) On ne veut pas passer par une référence pour pouvoir modifier la valeur.

TASK 1 :

Objectif 1 - Référencement des avions

A - Choisir l'architecture

• créer une nouvelle classe, AircraftManager :
On évite de surcharger une classe déjà existante.
Plus simple à manipuler.

• donner ce rôle à une classe existante :

B - Déterminer le propriétaire de chaque avion

1.Qui est responsable de détruire les avions du programme ? (si vous ne trouvez pas, faites/continuez la question 4 dans TASK0)

La fonction ‘void timer(const int step)’ est responsable de la destruction de nos avions. Dans la classe ‘opengl_interface.cpp’.

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

On a ‘display_queue’, ‘move_queue’ , ‘reserved_terminals’ (Tower) et ‘const Aircraft* current_aircraft’ (Terminal).

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

On va enlever notre avion de la ‘display_queue’ mais il existe toujours en mémoire. Il faut lors le détruire avec un ‘delete’. On supprime donc sa valeur en mémoire. Dans ‘Tower’ et ‘Terminal’ on ne peut plus accéder à l’avion car il n’existe plus. L’owner de chaque avion est donc la ‘move_queue’.

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

Si l’on veut appliquer le même chose pour notre ‘AircrfatManager’ il faut que notre aircraft ait accès à son aicraftManager ce qui n’est pas logique. L’aicraftManager doit bien s’occuper de la destruction des aicrafts.

C - C'est parti !

On va stocker nos avions dans un ‘vector’ de ‘unique_ptr’. On a plus besoin d’utiliser notre ‘move_queue’ car nos on a accès à nos avions directement dans notre vector d’avion.

On peut réutiliser notre fonctions ‘move’ de ‘timer’ il suffit juste de changer l’utilisation de la ‘move_queue’ par notre nouveau conteneur d’aircrafts de notre ‘Aircraft manager’.

Objectif 2 – Usine à avions

A – Création d’une factory

A quelle ligne faut-il définir context_initializer dans TowerSimulation pour s'assurer que le constructeur de context_initializer est appelé avant celui de factory ?

On va définir notre ‘context_initializer’ dans le constructeur de ‘TowerSimulation’ comme cela nous sommes sur qu’il est appelé avant de celui de ‘factory’.

Notre ‘aircraft_factory’ va donc contenir une fonction ‘create_aircraft’ qui va avec un type d’airline et un aéroport nous créer un avion et une fonction ‘create_random_aircraft’ qui va créer un avion aléatoirement.

B – Conflits

On a créé un un ‘set’ pour contenir tout les numéro d’avion pour éviter les doublons, ainsi que deux fonctions ‘add_aircraft_number’ et ‘get_aircraft_number’ pour ajouter des numéro de vol et pour les récupérer.

```
std::set<std::string> aircraftsNumbers;
```

TASK 2 :

Objectif 1 – Réfactorisation de l’existant

A – Structure Bindings

```
for (const auto& [key : const char ,value : const function<void ()> ] : GL::keystrokes)
{
    std::cout << key << ' ';
}
```

On va utiliser une couple de valeur avec la ‘key’ étant un char et notre ‘value’ une fonction qui ne renvoie rien.

B – Algorithmes divers

1) On créer un prédicat ‘is_delete’ qui nous renvoie un booléen si l’avion doit être détruit ou non.
On va ensuite utilisé la cette fonction dans notre ‘remove_if’ pour effacer un avion ou non .

```
std::function<bool(std::unique_ptr<Aircraft>&)> is_delete = [this](std::unique_ptr<Aircraft>& aircraft) -> bool {
    try {
        aircraft->move();
    }catch(const AircraftCrash& crash){
        std::cerr << crash.what() << std::endl;
        ++total_crashed_aircraft;
    }
     return aircraft->del_object();
};

aircrafts.erase( first: std::remove_if( first: aircrafts.begin(), last: aircrafts.end(), pred: is_delete), last: aircrafts.end());
```

2) On créer une fonction ‘aircrafts_by_airlines’ dans ‘aircrafts_manager’ qui nous renvoie le nombre d’avion par airline.

On utilise la fonction ‘std::count_if’ qui avec un prédicat et un numéro d’airline va compter le nombre total d’avion appartenant à celui-ci.

C – Relooking de Point3D

1) On va utiliser la fonction ‘std::transform’ qui prend un lambda et renvoie son résultat ici le résultat de la multiplication d’un scalaire avec les valeur de notre array.

2) Pour cette question on utilise la même fonction que la question précédente on va juste pas utiliser un lambda mais la fonction ‘std::plus<>’ qui existe déjà.

3) Pour notre fonction ‘length’ on ne va pas multiplier chacune des valeurs à la main on va utiliser la fonction ‘std::accumulate’.

Objectif 2 – Rupture de kérosène

A – Consommation d’essence

J’ajoute une nouvelle variable ‘fuel’ qui est une initialisé à la création d’un avion.

```
int fuel = (rand()%2850)+150;
```

Si notre avion n’a plus d’essence on renvoie une erreur :

```
if(fuel <= 0)
{
    throw AircraftCrash { arg: flight_number + " out of fuel then crashed " };
}
```

B – Un terminal s’il vous plaît

1) Notre fonction ‘has_terminal’ renvoie un booléen qui nous indique si l’avion appelant cette méthode possède déjà un terminal réservé.

2) La fonction ‘is_circling’ renvoie un booléen qui renvoie l’inverse de ‘has_terminal’ donc si l’avion n’a pas de terminal réservé et tourne donc en boucle autour de notre aéroport.

3) ‘reserve_terminal’ , va renvoyer un chemin vers un terminal si il y en a un de disponible, sinon renvoie un chemin vide.

4) On simplement rajouter une condition dans la fonction ‘move’ d’un aircraft si il n’est pas atterrit et qu’il est ‘is_circling’ donc qu’il n’a pas de terminal réservé on essaie de lui en assigner un.

C – Minimiser les crashs

Au début de notre fonction ‘move’ dans aircraft on va utiliser ‘std::sort()’ pour trier notre liste d’avions. Pour pouvoir trier les différents aircraft j’implémente l’opérateur ‘<’ dans ma classe ‘aircraft’. On les tries pour minimiser le nombre de crash et priorisé la reservation de terminal aux avions avec le moins de fuel.

D – Réapprovisionnement

1) Notre fonction ‘is_low_on_fuel’ va renvoyer true si l’avion qui appelle cette méthode possède moins de 200 de fuel.

2) La fonction ‘get_required_fuel()’ va renvoyer l’ensemble du fuel manquant pour chaque avion

3) On rajoute chacun de nos nouveaux attributs ainsi qu’un attribut ‘aicraftManager’ afin que notre aéroport puisse y accéder.

4) Un créer notre fonction ‘refill ‘ dans la classe ‘Aicraft’, pour éviter que notre variable ‘fuel_stock’ ne soit négatif on la définit que un une unsigned int.

5) On définit notre fonction ‘refill_aircraft_if_needed’ , on vérifie que notre aircraft ait bien été initialisé, si il est dans un terminal et qu’il lui manque de l’essence on le lui en donne.

6) On modifie notre fonction ‘move ‘ de ‘Airport’ afin que l’on puisse réapprovisionner un avion d’essence à chaque fois que notre variable ‘next_refill_time’ vaut 0. Chaque terminal pourra remplir d’essence son avion s’il en a besoin.

TASK 3 :

Objectif 1 – Crash des avions

1) On va remonter l’erreur du crash d’un avion jusqu ‘aux ‘move’ de ‘AircraftManager’ et l’on av utiliser un try catch pour éviter que le programme crash et simplement renvoyer un message d’erreur.

```
try {
    aircraft->move();
}catch(const AircraftCrash& crash){
    std::cerr << crash.what() << std::endl;
    ++total_crashed_aircraft;
}
```

2) La touche ‘m’ de notre clavier va afficher le nombre total d’avion qui se sont écrasés, on va incrémenter cette valeur dans notre try catch du ‘move’ de ‘AicraftManager’ lorsqu’un avion s’écrase.

3) Il suffit de rajouter une nouvelle erreur quand le fuel d’un avion atteint 0. On rajouter cette erreur dans la fonction ‘move’ de la classe ‘Aicraft’.

```
if(fuel <= 0)
{
    throw AircraftCrash { arg: flight_number + " out of fuel then crashed " };
}
```

Objectif 2 – Détecter les erreurs de programmation

On vérifie que le fuel d’un avion ne soit pas en dessous de 0.

Lorsque l’on compte le nombre d’avion par airline on vérifie que l’airline soit bien compris entre 0 et 7, donc qu’il existe.

Dans notre factory d’avion et dans la fonction ‘create_random_aircraft’ de ‘TwoerSimulation’ on vérifie que l’aéroport ait bien été initialisé avant de créer des avions.

Lorsque l’on réapprovisionne un avion on vérifie qu’il existe avant.

TASK 4 :

Objectif 1 – Devant ou derrière ?

1) On ajoute bien l’utilisation de la fonction ‘add_waypoint’ dans la fonction ‘move’ d’ ‘Aircraft’.

2) On modifie la fonction ‘add_waypoint’ en une fonction template qui prend en paramètre un booléen, on vérifie la condition avec ‘constexpr’ afin que l’évaluation du flag ait lieu à la compilation.

Objectif 2 – Points génériques

1) On créer une classe template ‘Point’ avec un paramètres, ‘dimension’ qui est le nombre de total valeur et le ‘typename T’ avec T qui représente le type des valeurs données.

2) Le code de la fonction libre ‘test_generic_points’ ne renvoie pas d’erreur le compilateur est donc capable de générer des classes à partir de notre template.

3) On va créer deux alias ‘Point3D’ et ‘Point2D’ avec notre classe ‘Point’ , avec leurs nombre d’argument respectif.

4) Un ‘Point2D’ ne peut contenir grand max que deux paramètres, l’erreur ne se produit que maintenant car l’on avait pas définit à combien de paramètres un ‘Point2D’ était représenté.

5) Pour éviter qu’un l’on instancie un ‘Point3D’ avec seulement 2 paramètres on va utiliser des ‘static_assert’ et pour éviter d’appeler des fonctions qui ne peuvent être appelé seulement sur certain type de point.

6) On créer un constructeur template qui utilise le type de point données et une multitude d'argument. 'std::forward' va nous permettre d'assurer le passage de nos paramètre et leurs bonnes conversions. On n'oublie pas de laisser le 'static_assert' qui vérifie que le nombre de paramètres donnée correspond bien à la dimension énoncée. Dans notre constructeur 'arg1' représente le premier argument données et 'K&&… args' le reste des arguments.

Le Projet :

Une des difficultés du projet à été de comprendre le code données, il est plus simple de commencer un projet plutôt que de continuer un projet existant.

Le fait de déplacer l'appartenance de certain objet à d'autre classe, de supprimer des objets référencées dans plusieurs partie du programme, l'on peut se perdre rapidement.

Plusieurs choix se sont proposées à nous durant ce projet comme notre façon d'implémenter la classe 'AircraftManager' , mais le sujet du projet nous aident plutôt pas mal à savoir ce qu'il faut faire et quoi modifier précisément.

Le projet était très instructif et intéressant, on découvre et apprend beaucoup de nouvelle façon de développer.

Ce projet ma permis d'en savoir beaucoup plus sur le langage c++ ainsi que comment le compilateur agit, les différents erreurs et problèmes sur lesquels on peut tomber.