Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# RAPPORT DE PROJET

Tower Control

JAAFRI Amyr

Février 2022 à Avril 2022

**Responsable d'enseignement** : Céline NOËL
**Chargé de TD :** Mathias WELLER

**Établissement** : Université Gustave Eiffel – Institut Gaspard-Monge
**Formation** : Master 1 Informatique
**TD** : Groupe 2
**Matière**: Programmation en C++

## Avant-propos

Dans le document qui suit, les mots en italique désigne des identificateurs ou des anglicismes traduit ou en brut (e.g., fonctions, variables, nom de fichiers, classe amie, etc.), les mots en italique et en gras désigne des mot-clés (e.g., ***class***, ***private***, etc.) ou des types (e.g., types primitifs, objets, etc.).
Qu'il en soit fait mention ou pas, le lecteur peut se référer au code avec l'historique des modifications en faisant une recherche par occurrence dans le code du projet, tel que : *« **TASK**_[**1**, …,**3**] – [**A**,…,**D**].[**1**,**2**,…] ».* Cette traçabilité de l'évolution du projet m'a aidé, et je l'espère aidera le lecteur à mieux percevoir la direction qui a été prise par moment.
Afin d'éviter la redondance et de faciliter la lecture, j'ai préféré détailler mes choix d'implémentations uniquement lorsque cela me semblait pertinent.
Par manque de temps

## Sommaire

# I. Task 0

*Note : git clone … + git checkout to … win_mingw*

## A) Exécution

1) Allez dans le fichier *tower_sim.cpp* et recherchez la fonction responsable de gérer les inputs du programme.

***TowerSimulation****::create_keystrokes()*

2) Sur quelle touche faut-il appuyer pour ajouter un avion ?

La touche *C* .

3) Comment faire pour quitter le programme ?

Appuyer sur la touche *X/Q* .

4) A quoi sert la touche *F* ?

A activer/désactiver le mode plein écran.

5) Ajoutez un avion à la simulation et attendez. Que est le comportement de l'avion ?

L’avion vol, atterrit sur le tarmac, s’arrête au terminal et enfin il repart.

6) Quelles informations s'affichent dans la console ?

Land, lift off, servicing .

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

Il vol jusqu’à ce qu’au moins 1 terminal soit disponible.

## B) Analyse du code

1) 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 classes présentent à la racine du dossier *src/* sont :

- ***Aircraft*** : Un avion ;
- ***Airport*** : Un aéroport avec ses terminales et sa tour de contrôle;
- ***AirporType*** : La position des éléments de l’aéroport;
- ***Terminal*** : Un terminal assigné à des avions ;
- ***Tower*** : Une tour de contrôle associer à un aéroport qui donne des instructions à des avions ;
- ***TowerSimulation*** : Mettre en place la simulation ;
- ***Waypoint*** : Contrôler les marques de parcours de l’avion (au sol, au terminal, etc.).

2) 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 interagissent ensemble.

- ***Aircraft***

| Name | Parameters | Return type | Description |
|---|---|---|---|
| *Aircraft* | ***const AircraftType****& type_* | ***void*** | (Constructeur) |

| | | | |
|---|---|---|---|
| | ***const std***::***string_view***& *flight_number_*<br>***const Point3D***& *pos_*<br>***const Point3D***& *speed_*<br>***Tower***& *control_* | | |
| *distance_to* | ***const Point3D***& *p* | ***float*** | Renvoyer la distance du membre *pos* avec *p* |
| *get_flight_num* | - | ***std::string***& | Renvoyer le numéro de vol de cet avion |
| *display* | - | ***void*** | Dessiner l'avion |
| *move* | - | ***void*** | Mettre à jour le déplacement de l'avion |

- ***Tower***

| Name | Parameters | Return type | Description |
|---|---|---|---|
| *Tower* | ***const Airport***& *airport_* | ***void*** | (Constructeur) |
| *get_instructions* | ***Aircraft***& *aircraft_* | ***WaypointQueue*** | Fournir les instructions à l'avion |
| *arrived_at_terminal* | - | ***void*** | Récupèrer le terminal sur lequel doit se poser l'avion et le fournir |

- ***Airport***

| Name | Parameters | Return type | Description |
|---|---|---|---|
| *Aircraft* | ***const AirportType***& *type_*<br>***const Point3D***& *pos_*<br>***const img::Image**** *image*<br>***const float*** *z_* | ***void*** | (Constructeur) |
| *get_tower* | - | ***Tower***& | Renvoyer la tour de contrôle |
| *move* | - | ***void*** | Déplacer tout les terminaux |

- ***Terminal***

| Name | Parameters | Return type | Description |
|---|---|---|---|
| *Terminal* | ***const Point3D***& *pos_* | ***void*** | (Constructeur) |
| *in_use* | - | ***bool*** | Vérifier si ce terminal est occupé par un avion |
| *in_servicing* | - | ***bool*** | Vérifier si ce terminal a terminé de s'occuper d'un avion |
| *assign_craft* | ***const Aircraft***& *aircraft_* | ***void*** | Assigner un avion |
| *start_service* | ***const Aircraft***& *aircraft_* | ***void*** | Commencer à gérer un avion |
| *finish_service* | - | ***void*** | Vérifier si le terminal gère toujours un avion et supprimer la référence sur cet avion si c'est le cas |
| *move* | - | ***void*** | Incrémenter la progression du traitement de l'avion si le terminal le gère et qu'il n'a pas finit de s'occuper de lui |

3) 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.

***Aircraft****::move* vérifie s'il n'existe aucun chemin pour cet avion, si c'est le cas on affecte à *waypoints* le résultat de l'appel à ***Tower****::get_instructions*.
***Tower****::get_instructions* fait un appel à ***Airport****::reserve_terminal* qui appel ***Terminal****::assign_craft* s'il existe un terminal qui n'est pas occupé.
***Terminal****::assign_craft* assigne à ce terminal un avion.

Une ***std****::****deque*** (i.e. double-ended queue) a été choisi pour représenter le chemin, car il permet d'insérer et de supprimer des éléments à la fin, comme les ***std****::****vector***, mais aussi d'insérer et de supprimer des éléments au début du conteneur. Ainsi, on est en mesure d'ajouter/supprimer des points du chemin tant au début qu'à la fin de la ***WaypointQueue***.

## C) **Bidouillons !**

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.

Dans *aircraft_types.hpp* :

```
30  inline void init_aircraft_types()
31  {
32      aircraft_types[0] = new AircraftType { .02f, .05f, .02f, MediaPath { "l1011_48px.png" } };
33      aircraft_types[1] = new AircraftType { .02f, .05f, .02f, MediaPath { "b707_jat.png" } };
34      aircraft_types[2] = new AircraftType { .02f, .05f, .02f, MediaPath { "concorde_af.png" } };
35  }
36
```

De plus, sachant que le deuxième argument du constructeur représente *max_air_speed*, il nous suffit de faire passer *.05f* par *.06f* à la ligne 34.

2) 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 ? Ajoutez une nouvelle fonctionnalité au programme pour mettre le programme en pause, et qui ne passe pas par le framerate.

Dans *config.hpp*, on trouve la constante suivante :

```
23  // default number of ticks per second
24  constexpr unsigned int DEFAULT_TICKS_PER_SEC = 16u;
```

On retrouve cette dernière affecter à une variable dans *opengl_interface.hpp* :

```
21  inline unsigned int ticks_per_sec = DEFAULT_TICKS_PER_SEC;
```

Note : la spécification du mot clé ***in-line*** nous permet de définir une variable globale. On s'assure ici que les adresses seront les mêmes.

Au bout d'un certain nombre de diminution de *ticks_per_sec*, cette dernière risque d'être nul et donc d'engendrer une division par zéro. Ainsi, il nous faut trouver un mécanisme afin que cette variable ne soit jamais égale à zéro (cf., *opengl_interface.*cpp, l.84).
On rajoute trois inputs, respectivement, augmenter, diminuer ou mettre en pause le framerate. On obtient dans *tower_sim.cpp* les lignes suivantes :

```
68      GL::keystrokes.emplace('a', []()
69                             { GL::ticks_per_sec += 1u; });
70      GL::keystrokes.emplace('d', []()
71                             { GL::ticks_per_sec =
72                                   std::max(GL::ticks_per_sec - 1u, 1u); });
73      GL::keystrokes.emplace('p', []()
74                             { GL::is_paused = !GL::is_paused; });
75  }
```

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

```
11  // number of cycles needed to service an aircraft at a terminal
12  constexpr unsigned int SERVICE_CYCLES = 40u;
```

4) Lorsqu'un avion a décollé, il réatterrit peu de temps après. Faites en sorte qu'à la place, il soit retiré du programme.

*Indices :*
*- A quel endroit pouvez-vous savoir que l'avion doit être supprimé ?*
*- Pourquoi n'est-il pas sûr de procéder au retrait de l'avion dans cette fonction ?*
*- A quel endroit de la callstack pourriez-vous le faire à la place ?*
*- Que devez-vous modifier pour transmettre l'information de la première à la seconde fonction ?*

On peut savoir quand l'avion doit être supprimer dans ***Aircraft****::move*, car c'est ici où on contrôle la distance entre l'avion et le point de cheminement à l'aide de *DISTANCE_THRESHOLD* (i.e., la distance en dessous de laquelle on considère que le point a été atteint).
Cependant, les avions ne peuvent être généré/supprimé que par ***GL****::move_queue*.
Ainsi, on pourrait gérer cela dans ***TowerSim****::timer* puisque c'est ici que l'on fait appel à la fonction *move*.
Pour régler le problème d'asymétrie d'informations entre ces deux fonctions, on peut en même temps parcourir *move_queue* et en modifier le contenu. On doit utiliser un itérateur, comme suit :

```
90          // TASK_0 - C.4)
91          for (auto il = move_queue.begin(); il != move_queue.end();)
92          {
93              auto *object = *il;
94              if (object->move())
95              {
96                  il++;
97              }
98              else
99              {
100                 il = move_queue.erase(il);
101                 delete object;
102             }
103         }
```

Au préalable, on a changé la signature de ***Airport****::move*, ***Terminal****::move* et ***Aircraft****::move* afin que ces dernières renvoient un booléen, et ainsi indiquer lorsque les opérations sont terminées.

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

Pour que l'ajout et la suppression de *display_queue* soit "automatiquement gérée", une des solutions est d'effectuer ces opérations, respectivement, dans le constructeur et le destructeur de ***Displayable***. Conséquemment, on devra intégrer *display_queue* comme membre *static* de ***Displayable*** afin que le conteneur soit le même pour toutes les instances. De plus, on devra préfixer, si nécessaire, *display_queue* par le nom de la classe englobante, c'est-à-dire ***Displayable***.
Ici, il n'est pas intéressant de procéder de même pour ***DynamicObject***, car a priori nous ne cherchons qu'à faire disparaître des avions. Dans *displayable.hpp*, on obtient le bout de code suivant :

```
17      public:
18          Displayable(const float z_) : z{z_}
19          {
20              // TASK 0 - C.5)
21              display_queue.emplace_back(this);
22          }
23          virtual ~Displayable()
24          {
25              // TASK 0 - C.5)
26              display_queue.erase(
27                  std::find(display_queue.begin(),
28                            display_queue.end(),
29                            this));
30          }
31
32          virtual void display() const = 0;
33
34          float get_z() const { return z; }
35
36          // TASK 0 - C.5)
37          static inline std::vector<const Displayable *> display_queue;
38      };
```

De plus, afin de savoir si l'avion a déjà été servi, on ajouté un champ privé à ***Aircraft*** tel que :

```
24          // TASK 0 - C.5)
25          bool has_been_served = false;
```

Ainsi, on peut mémoriser si un appel à ***Tower****::get_instructions* par cette avion a déjà eu lieu. S'il a effectivement déjà fait appel à cette fonction, ***Aircraft****::move()* renverra ***false*** puisqu'on ne souhaite plus fournir d'instructions à cette avion.

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 Aircraft****&)* ne devrait plus être nécessaire.

On peut utiliser *map* ou *unordered_map*. Ici, nous avons privilégié le choix de *unordered_map*, car la recherche se fait en moyenne en temps constant ou en temps linéaire au pire des cas, au détriment de la préservation de l'ordre qui n'est guère nécessaire pour son utilisation (cf., TASK_0 – C.6) )

D) **Théorie**

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

***Airport****::reserve_terminal* est déclaré en tant que fonction membre privée. Or, ***Tower*** est déclarée en tant que *classe amie* dans ***Airport***, c'est-à-dire que cette dernière autorise ***Tower*** à accéder à ses membres privés.

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

On ne souhaite pas modifier la valeur de l'argument *direction*, car on lui applique des opérations qui sont dans le seul but de calculer la vitesse de l'avion. Il n'est pas question d'engendrer un effet de bord sur *direction*.
On pourrait éviter la copie de l'argument en opérant à une transmission par référence et y ajouter ***const*** afin d'empêcher tout effet de bord. Cependant, il faudra procéder d'une autre manière afin de faire les calcules nécessaires sur *distance*.

E) **Bonus**

Le temps qui s'écoule dans la simulation dépend du framerate du programme.
La fonction *move()* n'utilise pas le vrai temps. Faites en sorte que si.
Par conséquent, lorsque vous augmentez le framerate, la simulation s'exécute plus rapidement, et si vous le diminuez, celle-ci s'exécute plus lentement.

# II. Task 1

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

Au point où nous sommes, on peut chercher l'avion indistinctement dans *move_queue* ou *display_queue*, car la recherche dans les deux conteneurs se fait en complexité de temps égale (cf., ***unordered_set***, ***vector***).

## 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.*

1) Réfléchissez aux pour et contre de chacune de ces options.

En ce qui concerne le choix de créer la classe ***AircraftManager*** : C'est une solution qui respecte le principe de *délégabilité* mais qui n'est pas efficace. En effet, on devra instancier une nouvelle classe pour chaque avion, ce qui peut ralentir notre programme.
A contrario, désigner une classe qui aura ce rôle n'est pas la meilleure d'un point de vue *clean code* mais la meilleure en terme d'efficacité.

**Pour le restant de l'exercice, vous partirez sur le premier choix.**

## B) Déterminer le propriétaire de chaque avion

*Vous allez introduire une nouvelle liste de références sur les avions du programme.*
*Il serait donc bon de savoir qui est censé détruire les avions du programme, afin de déterminer comment vous allez pouvoir mettre à jour votre gestionnaire d'avions lorsque l'un d'entre eux disparaît.*

1) Répondez aux questions suivantes :

(1) Qui est responsable de détruire les avions du programme ? (si vous ne trouvez pas, faites/continuez la question 4 dans TASK 0)

C'est dans ***GL****::timer* que l'on parcours et que l'on modifie le contenu de *move_queue* (cf., TASK 0 – C)).

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

***GL****::****Displayable****::display_queue*, ***GL****::move_queue* et ***Tower****::reserved_terminals*.

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

Afin de propager la suppression à laquelle nous allons opérer, nous n'avons pas d'autres

solutions que de procéder à la suppression de toutes les références en parcourant les conteneurs susmentionnés.

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

Conformément au suffixe du nom de la classe (i.e., *Manager*), notre classe ***AircraftManager*** ne doit procéder à aucune modification puisque c'est elle qui déterminera la durée de vie d'un avion et déléguera la suppression aux autres classes.

2) Pour simplifier le problème, vous allez déplacer l'ownership des avions dans la classe ***AircraftManager***.
Vous allez également faire en sorte que ce soit cette classe qui s'occupe de déplacer les avions, et non plus la fonction *timer*.

## C) C'est parti !

1) Ajoutez un attribut *aircrafts* dans le gestionnaire d'avions. Choisissez un type qui met bien en avant le fait que ***AircraftManager*** est propriétaire des avions.

```
14    // TASK_1 - C.1)
15    std::vector<std::unique_ptr<Aircraft>> aircrafts;
```

*Note : Utiliser un **std::unique_ptr** nous permet de nous assurer que la propriété de l'emplacement mémoire des aircrafts appartient uniquement à **AircraftManager** (i.e., il ne sont référencés que par le pointeur issue de **AircraftManager**).*

2) Ajoutez un nouvel attribut *aircraft_manager* dans la classe ***TowerSimulation***.

(cf., ***TASK_1 – C.2)***).

3) Modifiez ensuite le code afin que *timer* passe forcément par le gestionnaire d'avions pour déplacer les avions.
Faites le nécessaire pour que le gestionnaire supprime les avions après qu'ils aient décollé.

(cf., ***TASK_1 – C.3)***).

4) Enfin, faites ce qu'il faut pour que *create_aircraft* donne l'avion qu'elle crée au gestionnaire. Testez que le programme fonctionne toujours.

(cf., ***TASK_1 – C.4)***).

## D) Création d'une factory

(cf., ***TASK_1 – D***).

```
10  // TASK_1 - D
11  inline std::vector<std::string> airlines{"AF", "LH", "EY", "DL", "KL", "BA", "AY", "EY"};
```

*Note : On souhaite, pour plus de souplesse, que airlines puisse être définit plus d'une fois. Ainsi, on spécifie le mot-clé **inline**.*

```
31      // TASK_1 - D
32      std::unique_ptr<Aircraft> create_aircraft(Tower &tower);
```

*Note : Dans l'optique de fournir le résultat de la fonction à **AircraftManager**::add_aircraft, il est nécessaire de renvoyer un résultat de type **std**::unique_ptr<**Aircraft**> (cf., **TowerSimulation**::create_random_aircraft).*

E) **Conflits**

1) Il est rare, mais possible, que deux avions soient créés avec le même numéro de vol. Ajoutez un conteneur dans votre classe ***AircraftFactory*** contenant tous les numéros de vol déjà utilisés. Faites maintenant en sorte qu'il ne soit plus possible de créer deux fois un avion avec le même numéro de vol.

```
12  // TASK_1 - E
13  std::string AircraftFactory::get_flight_number()
14  {
15      std::string flight_number;
16      do
17      {
18          flight_number = airlines[std::rand() % airlines.size()] + std::to_string(1000 + (rand() % 9000));
19      } while (flight_numbers.find(flight_number) != flight_numbers.end());
20      flight_numbers.emplace(flight_number);
21      return flight_number;
22  }
```

Le champ flight_number (un ***std***::*unordered_set*<***std***::*string*>) nous servira de référence pour éviter les doublons, d'où le choix d'un ensemble

# III. Task_2

## A) Structured Bindings

1) ***TowerSimulation****::display_help()* est chargé de l'affichage des touches disponibles. Dans sa boucle, remplacez ***const auto****& ks_pair* par un structured binding adapté.

```
100     // TASK_2 - A
101     for (const auto &[key, function] : GL::keystrokes)
102     {
103         std::cout << key << ' ';
104     }
105     std::cout << std::endl;
```

On peut directement spécifier le couple dans la boucle *foreach*.

## B) Algorithmes divers

1) ***AircraftManager****::move()* (ou bien *update()*) supprime les avions de la *move_queue* dès qu'ils sont "hors jeux". En pratique, il y a des opportunités pour des pièges ici. Pour les éviter, *<algorithm>* met à disposition la fonction ***std****::remove_if*. Remplacez votre boucle avec un appel à ***std****::remove_if*.

```
26      // TASK_2 - B.1)
27      aircrafts.erase(std::remove_if(aircrafts.begin(), aircrafts.end(),
28                                     [this](const std::unique_ptr<Aircraft> &aircraft)
29                                     { return !aircraft->move(); }),
30                      aircrafts.end());
```

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.

```
 98     // TASK_2 - B.2)
 99     for (auto i = 0; i < (int)airlines.size(); i++)
100     {
101         GL::keystrokes.emplace('0' + i, [this, i]()
102                                { show_airline(i); });
103     }
```

On peut économiser des lignes de codes en parcourant les index de *airlines* et en les sommant avec le code ASCII de *'0'*. Ainsi, on obtiendra une touche pour chaque ligne. ***TowerSimulation****::show_airline* permettra d'adapter l'affichage sur la console en fonction de l'index de la ligne dans *airlines* puis par un appel à ***AircraftManager****::count*, tel que :

```cpp
// TASK_2 - B.2)
int AircraftManager::count(const std::string_view &line)
{
    return std::count_if(aircrafts.begin(), aircrafts.end(),
                         [line](const std::unique_ptr<Aircraft> &a)
                         { return (a->get_flight_num().rfind(line, 0) == 0); });
}
```

C) **Relooking de Point3D**

1) 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)*

```cpp
        // TASK_2 - C.1).1)
        std::transform(values.begin(), values.end(), values.begin(),
                       [scalar](auto k)
                       { return k * scalar; });
```

(2) ***Point3D****::operator+=(const Point3D& other)*

```cpp
    // TASK_2 - C.1).2)
    std::transform(other.values.begin(), other.values.end(), values.begin(), values.begin(),
                   [](auto k, auto p)
                   {
                       return k + p;
                   });
```

(3) ***Point3D****::length() const*

```cpp
    // TASK_2 C.1).3)
    return std::sqrt(
        std::accumulate(values.begin(), values.end(), 0.f, [](float sum, float next)
                        { return sum + next * next; }));
```

*Note : L'utilisation de **std**::minus<**float**>, **std**::plus<**float**>, produit des erreurs que je n'ai pas réussi à déboguer. Ainsi, je propose une version un peu moins élégante mais fonctionnelle.*

Indication : Vous allez introduire la gestion de l'essence dans votre simulation. Comme le but de ce TP est de vous apprendre à manipuler les algorithmes de la STL, avant d'écrire une boucle, demandez-vous du coup s'il n'existe pas une fonction d'<algorithm> ou de <numeric> qui permet de faire la même chose. La notation tiendra compte de votre utilisation judicieuse de la librairie standard.

D) **Consommation d'essence**

(cf., ***TASK_2 – D***)

*Note : J'ai choisi de décrémenter fuel de 1. Ce choix est arbitraire.*

E) **Un terminal s'il vous plaît**

Afin de minimiser les crashs, il va falloir changer la stratégie d'assignation des terminaux aux avions.

Actuellement, chaque avion interroge la tour de contrôle pour réserver un terminal dès qu'il atteint son dernier ***Waypoint***. Si un terminal est libre, la tour lui donne le chemin pour l'atteindre, sinon, elle lui demande de tourner autour de l'aéroport.

Pour pouvoir prioriser les avions avec moins d'essence, il faudrait déjà que les avions tentent de réserver un terminal tant qu'ils n'en n'ont pas (au lieu de ne demander que lorsqu'ils ont terminé leur petit tour).

1) Introduisez une fonction ***bool Aircraft****::has_terminal()* ***const*** qui indique si un terminal a déjà été réservé pour l'avion (vous pouvez vous servir du type de *waypoints.back()*).

   (cf., ***TASK_2 – E.1)***)

2) Ajoutez une fonction ***bool Aircraft****::is_circling()* ***const*** qui indique si l'avion attend qu'on lui assigne un terminal pour pouvoir attérir.

   (cf., ***TASK_2 – E.2)***)

3) Introduisez une fonction ***WaypointQueue Tower****::reserve_terminal(****Aircraft****& aircraft)* qui essaye de réserver un ***Terminal***. Si c'est possible, alors elle retourne un chemin vers ce ***Terminal***, et un chemin vide autrement (vous pouvez vous inspirer / réutiliser le code de ***Tower****::get_instructions*).

   (cf., ***TASK_2 – E.3)***)

4) Modifiez la fonction *move()* de ***Aircraft*** afin qu'elle appelle ***Tower****::reserve_terminal* si l'avion est en attente. Si vous ne voyez pas comment faire, vous pouvez essayer d'implémenter ces instructions :

   (1) Si l'avion a terminé son service et sa course, alors on le supprime de l'aéroport (comme avant)

   (cf., ***if*** *(waypoints.empty())*)

   (2) Si l'avion attend qu'on lui assigne un terminal, on appelle ***Tower****::reserve_terminal* et on modifie ses *waypoints* si le terminal a effectivement pu être réservé

```
135     // TASK_2 - E.4).2)
136     if (is_circling())
137     {
138
139         auto waypoint_queue = control.reserve_terminal(*this); // On essaye de reserver un terminal
140         if (!waypoint_queue.empty())
141         {
142             waypoints = waypoint_queue;
143         }
144     }
```

   (3) Si l'avion a terminé sa course actuelle, on appelle ***Tower****::get_instructions* (comme avant).

(cf., ***if*** *(!has_been_served)*)

F) **Minimiser les crashs**
(cf., ***TASK_2 - F***)

G) **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.

(cf., ***TASK_2 – G.1)***)

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.

(cf., ***TASK_2 – G.2)***)

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.

(cf., ***TASK_2 – G.3)***)

4) Ajoutez une fonction *refill* à la classe ***Aircraft***, prenant un paramètre `fuel_stock` par référence non-constante.
Cette fonction remplira le réservoir de l'avion en soustrayant ce dont il a besoin de *fuel_stock*.
Bien entendu, *fuel_stock* ne peut pas devenir négatif.
Indiquez dans la console quel avion a été réapprovisionné ainsi que la quantité d'essence utilisée.

(cf., ***TASK_2 – G.4)***)

5) Définissez maintenant une fonction *refill_aircraft_if_needed* dans la classe ***Terminal***, prenant un paramètre *fuel_stock* par référence non-constante.
Elle devra appeler la fonction *refill* sur l'avion actuellement au terminal, si celui-ci a vraiment besoin d'essence.

(cf., ***TASK_2 – G.5)***)

6) Modifiez la fonction ***Airport****::update*, afin de mettre-en-oeuvre les étapes suivantes.\
- Si *next_refill_time* vaut 0 :
 - *fuel_stock* est incrémenté de la valeur de *ordered_fuel*.
 - *ordered_fuel* est recalculé en utilisant le minimum entre ***AircraftManager****::get_required_fuel()* et 5'000 (il s'agit du volume du camion citerne qui livre le kérosène)
 - *next_refill_time* est réinitialisé à 100
 - La quantité d'essence reçue, la quantité d'essence en stock et la nouvelle quantité d'essence commandée sont affichées dans la console.

- Sinon *next_refill_time* est décrémenté.
- Chaque terminal réapprovisionne son avion s'il doit l'être.

```
90          // TASK_2 - G.6)
91          if (next_refill_time <= 0)
92          {
93              fuel_stock += ordered_fuel;
94              int temp = ordered_fuel;
95              ordered_fuel = std::min(aircraft_manager.get_required_fuel(), 5000);
96              next_refill_time = 100;
97              std::cout << "Received fuel: " << temp << ", Stocked fuel: " << fuel_stock << ", Ordered fuel: " << ordered_fuel << std::endl;
98          }
99          else
100         {
101             next_refill_time -= 1;
102         }
```

```
104         // TASK_2 - G.6)
105         for (auto &t : terminals)
106         {
107             t.refill_aircraft_if_needed(fuel_stock);
108         }
```

# IV. Task_3

1) 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*.

Dans ***AircraftManager::****move.cpp*, nous obtenons le code suivant :

```
33      aircrafts.erase(std::remove_if(aircrafts.begin(), aircrafts.end(),
34                                     [this](const std::unique_ptr<Aircraft> &aircraft)
35                                     {
36                                         // TASK_3 - 1.1)
37                                         try
38                                         {
39                                             return !aircraft->move();
40                                         }
41                                         catch (const AircraftCrash &aircraft_crash)
42                                         {
43                                             std::cerr << aircraft_crash.what() << std::endl;
44                                             return true;
45                                         }
46                                     }),
47                      aircrafts.end());
```

Étant donné que nous lançons ***AircraftCrash*** dans ***Aircraft****::move* nous souhaitons attraper l'erreur là où nous devons supprimer l'avion en question. Ainsi, c'est au moment où nous parcourons *aircrafts* que nous rattrapons ***AircraftCrash*** afin de poursuivre l'exécution du programme.

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.

(cf. ***TASK_3 - 2)***)

3) Si vous avez fini d'implémenter la gestion du kérosène (*Task_2 - Objectif 2 - A*), lancez une exception de type ***AircraftCrash*** lorsqu'un avion tombe à court d'esssence. Normalement, cette exception devrait être traitée de la même manière que lorsqu'un avion s'écrase parce qu'il a attérit trop vite.

(cf., question 1))

# V. Conclusion

## A) Difficultés rencontrées

Compte-tenu de l'architecture de départ du projet qui était assez conséquente, j'ai accordé un peu trop de temps à la *TASK_0/1*, puisque la structure générale ne m'était pas tout à fait familière. De ce fait, cela ne m'a laissé que peu de temps vis-à-vis du travail plus intuitif et moins guidé des dernières *TASK*. Ainsi, il a fallut passé un peu moins de temps a préciser les détails de mes implémentations dans le rapport.

De plus, les modifications opérées à un endroit se propagent et provoquent des incohérences qui génèrent des messages d'erreurs qui ne sont que trop souvent peu explicites. Ces étapes sont énormément chronophage et m'ont causé beaucoup de frustration.

## B) Ce que j'ai aimé/détesté

Ayant apprécié mon expérience en C, le C++ a été l'occasion pour moi de pratiquer un langage qui préserve les avantages du C élagués de ses quelques inconvénients (i.e., absence de structures dynamiques, allocation manuelle, désallocation manuelle, etc.). Comparé à JAVA, la permissivité du C++, tant vis-à-vis du paradigme de programmation que du code, est agréable une fois la prise en main dépassée.

Cependant, le trop peu de temps accordé à un vrais cours et la surcharge de l'emploi du temps en M1, rendent le tout assez ardu malgré l'intérêt que l'on peut y porter. L'agencement assez rébarbatif du sujet (i.e., implémenter, puis corriger, puis corriger, etc.) n'aide pas vraiment à être constant dans l'avancement du projet, ce qui engendre à chaque reprise du sujet un temps conséquent de re-compréhension qui s'alourdit au fur et à mesure. Il me semble primordiale d'allouer plus de temps en présentiel, afin de pallier à ces problématiques.

L'interface graphique ne permet pas toujours de savoir si nous avons bien implémenté ce qui a été demandé. Peut-être que des tests unitaires ponctuels pourraient aider.

Enfin, l'effort mis dans le workflow de la page du cours pour nous améliorer avec git est louable mais ne suffit pas toujours. De manière générale, il faudrait une séance, au moins, dédié à l'utilisation

## C) Ce que j'ai appris

L'utilisation des lambda et des fonctions de la STL qui facilitent énormément l'implémentation d'algorithmes complexes. De plus, les subtilités du C++ qui permettent un gain de temps à l'exécution (e.g., ***inline***, ***static***, etc.).