Introduction

Le but de cette session est de se familiariser avec les notions de classe, de pointeur, de référence du `C++`.

Il s'agit également de mettre en place les premiers éléments permettant la simulation à grandes échelles de systèmes particulaires complexes.

Vous êtes libres d'utiliser l'environnement de développement de votre choix. Je recommande d'utiliser `Visual Studio Code`.

1 Analyse de performance

Vous serez amenés à analyser les performances de votre implémentation à différentes étapes.

Il existe plusieurs applications permettant de mesurer les performances d'une implémentation. L'application la plus utilisée est `gprof`.

Afin d'utiliser `gprof`, le programme doit être compilé avec l'option `-pg`. L'exécution du programme générera un fichier de trace `gmon.out`.

En exécutant `gprof` sur le programme, vous obtiendrez un rapport d'analyse détaillé, mais verbeux, de l'exécution.

Le rapport est composé de deux parties :

- le profil plat, i.e., le détail des appels de chacune des fonctions : le temps, le nombre d'appels ;
- le profil d'appel, i.e, la hiérarchie d'appel des fonctions.

Pour rendre le rapport plus digeste, en enlevant les explications, il faut exécuter `gprof` avec l'option `-b`.

2 Mise en place des structures de données

Dans ce premier exercice, nous allons décrire la structure de donnée élémentaire utilisée.

Nous allons étudier le comportement de plusieurs dizaines, plusieurs centaines, voire plusieurs milliers de particules.

Dans un premier temps, on définira une particule comme un objet tri-dimensionnel caractérisé par une position, une vitesse, une masse, un identifiant, une catégorie (un type) et une force. Cette catégorie n'est pas utilisée dans cette première séance ; elle permettra de distinguer plusieurs types de particules dans les séances suivantes.

Définition 2.1 : Encapsulation

L'encapsulation est un des principes fondamentaux de la programmation orientée objet. Elle consiste à :

regrouper au sein d'une classe les données (attributs) et les fonctions qui les manipulent (méthodes) ;

cacher les détails d'implémentation en rendant les attributs **`private`**, et n'exposer que ce qui est nécessaire via une interface **`public`** (constructeur, accesseurs, mutateurs).

Cela permet de protéger les données contre des modifications incontrôlées et de faire évoluer l'implémentation interne sans casser le code qui l'utilise.

```cpp
class Particule {
  private:
    double masse; // attribut caché, inaccessible depuis l'extérieur
```

```
4
5   public:
6     Particule(double masse);   // constructeur
7
8     double getMasse() const;   // accesseur ("getter")
9     void   setMasse(double m); // mutateur ("setter")
10  };
```

Question 1.
Créer une classe `Particule` qui correspond à ces caractéristiques. Les attributs devront être **`private`** ; vous fournirez un constructeur ainsi que les accesseurs et mutateurs nécessaires pour y accéder depuis l'extérieur de la classe.

La simulation se compose d'une collection de particules. Il va donc être nécessaire de les garder en mémoire.

On pourrait créer une liste chaînée écrite à la main. Cependant, la bibliothèque standard offre une grande variété de collections. Elles ont toutes des propriétés différentes. Vous pouvez vous référer à la documentation pour les détails. On reviendra sur le sujet plus tard.

Pour utiliser une collection spécifique, vous devez inclure l'entête correspondante :

```
1  #include<set>
2  #include<list>
3  #include<deque>
4  #include<vector>
```

La collection est **`template`**, c'est-à-dire qu'elle s'adapte au type pour lequel on l'initialise (la notion de **`template`** sera abordée plus tard).

On instancie une collection, par exemple, de la manière suivante

```
1  std::list<Particule> listeParticule;
```

Toutes les collections ont des méthodes communes :

- insérer : `insert(val)` ;
- effacer : `erase(pos)` ;
- compter : `size()` ;

  et bien d'autres.

Une fois construite, la collection se parcourt à l'aide d'un itérateur :

```
1  auto it = listeParticule.begin();
```

`it` se comporte comme un pointeur. On peut l'utiliser dans les boucles pour parcourir la collection entre ses bornes :

```
1  for (it = listeParticule.begin(); it != listeParticule.end(); ++it){};
2  for (auto it : listeParticule){} ;
```

Étant donné que nous souhaitons créer une grande variété de particules, nous allons les initialiser de manière aléatoire :

```cpp
#include <random>
#include <iostream>

int main() {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    for (int i=0; i<16; ++i)
        std::cout « dist(mt) « "\n";
}
```

Ce générateur peut être utilisé pour tirer aléatoirement la masse, la position et la vitesse de chaque particule au moment de sa construction, avant de l'insérer dans la collection.

Question 2.
Créer une collection de particules initialisées aléatoirement.

Question 3.
Comparer les performances de chacune des collections pour $64, 128, \ldots, 2048, \ldots$ particules. A partir de quelle taille est-ce que les différences s'observent ?

Pour mesurer les performances, nous allons utiliser la bibliothèque `chrono`.
Il suffit ensuite d'entourer les éléments de code à mesurer avec des compteurs

```cpp
auto start = std::chrono::steady_clock::now();
/* Code à évaluer */
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double> elapsed_seconds = end-start;
std::cout « "elapsed time: " « elapsed_seconds.count() « "s\n";
```

3 Méthode de Störmer-Verlet

Il s'agit ensuite de déplacer les particules en les soumettant à différentes forces d'interaction.

Les forces d'interaction entre les particules sont définies explicitement. Dans un premier temps, nous allons utiliser la force d'interaction gravitationnelle :

$$\mathbf{F}_{ij} = \frac{m_i m_j}{r_{ij}^3}\mathbf{r}_{ij} \tag{1}$$

La force qui s'applique à une particule est la somme des forces d'interactions.

$$\mathbf{F}_i = \sum_{i \neq j} \mathbf{F}_{ij} \tag{2}$$

L'évolution du système particulaire se calcule à l'aide de l'algorithme de Störmer-Verlet :

Algorithme 1 : algorithme de Störmer-Verlet

Data : particules initiales
Data : vecteur de force $\mathbf{F}^{old}$
Initialisation : calcul des forces $\mathbf{F}$;
while $t < t_{end}$ **do**
  $t = t + \delta_t$;
  for *toutes les particules* i **do**
    $\mathbf{x}_i = \mathbf{x}_i + \delta_t * (\mathbf{v}_i + 0.5/m_i * \mathbf{F}_i * \delta_t)$;
    $\mathbf{F}_i^{old} = \mathbf{F}_i$;
  end
  Calculer les forces $\mathbf{F}$;
  for *toutes les particules* i **do**
    $\mathbf{v}_i = \mathbf{v}_i + \delta_t * 0.5/m_i * (\mathbf{F}_i + \mathbf{F}_i^{old})$;
  end
  Calculer les quantités dérivées ;
  Afficher les quantités t, x, y ;
end

Dans un premier temps, nous allons nous intéresser au système gravitationnel composé du Soleil, de la Terre, de Jupiter et de la comète de Halley.

Corps	Masse	Position	Vitesse
Soleil	1	(0,0)	(0,0)
Terre	3.0×10^{-6}	(0,1)	(-1,0)
Jupiter	9.55×10^{-4}	(0, 5.36)	(-0.425,0)
Halley	1.0×10^{-14}	(34.75,0)	(0, 0.0296)

On choisira $\delta_t = 0.015$ et $t_{end} = 468.5$.

Question 4.
Implémenter l'algorithme de Störmer-Verlet.

Question 5.
Vérifier votre implémentation pour le système gravitationnel proposé. Quels arguments vous permettent de conclure que la simulation est correcte ?

Question 6.
Sauvegardez les résultats dans un fichier texte, en utilisant les accesseurs mis en place précédemment plutôt qu'un accès direct aux attributs. Ce fichier devra contenir la position de chaque astre à chaque itération sur la même ligne.

4 La suite

Les sessions suivantes vont continuer d'utiliser des particules et sont conçues de manière incrémentale : vous allez étoffer petit à petit votre bibliothèque de modélisation de particules avec des nouvelles fonctionnalités.

Pensez bien à vérifier que le travail effectué précédemment continue de fonctionner au fur et à mesure des ajouts, notamment la simulation planétaire de cette séance.