Nguyen Thanh-Mai Christine Section Math Group n°55
Sekarski Samuel Section Math Group n°55

RÉPONSES

[Question P1.1]

Comment représentez-vous ces vecteurs tridimensionnels ? Comment sont-ils organisés : quels attributs, quelles méthodes, quels droits d'accès ?

[Réponse P1.1]

Représentation : Précisons en premier lieu que nous allons considérer que les « vecteurs » sont des « vecteurs libres » au sens géométrique, déterminés par une direction, un sens, et une longueur, et non pas des « vecteur fixes » (qui ont, de plus, une origine fixe et sont moins pratique à employer). Les vecteurs tridimensionnels ont trois composantes, selon la base canonique de R^3 (qui est en particulier orthogonal, ce qui permet un calcule facile de norme et produit scalaire). Il y a deux façons évidentes pour les représenter : un tableau fixe (array) et juste des doubles x, y et z. Nous avons choisi les doubles x, y et z car ils sont plus simples à écrire. Il y a un avantage, mais également un inconvénient à cette écriture. L'avantage est que nous pouvons voir clairement quel coordonnée nous utilisons/manipulons ; le désavantage, c'est qu'il faut à chaque fois les réécrire.

Attributs : Nous avons mis ces trois composantes en attribut.

Méthode : Un vecteur, il faut tout d'abord le définir, c'est-à-dire qu'il lui faut des composantes (la méthode setVecteur), il faut également pouvoir l'afficher (afficheVecteur), le comparer avec un autre vecteur (compare). De plus, il faudra le manipuler. Il lui faut donc une méthode pour l'addition/la soustraction avec un autre vecteur, l'opposé, la multiplication par un scalaire, le produit scalaire, le produit extérieur, vectoriel et mixte.

Droits d'accès : les attributs, c'est-à-dire les composantes sont en private, tandis que tout le reste, c'est-à-dire toutes les méthodes sont en public.

[Question P4.1]

Avez-vous ajouté un constructeur de copie ? Pourquoi (justifiez votre choix) ?

[Réponse P4.1]

Nous n'avons pas ajouté de constructeur de copie étant donné qu'il y en a déjà par défaut qui fait ce que l'on souhaite lorsqu'on veut affecter v_1 à v_2 ($v_2 = v_1$;), donc recopier les coordonnées (x, y, z) de v_1 dans ceux de v_2.

[Question P4.2]

Pourquoi, contrairement à ce qui est trop souvent fait, l'écriture des deux premiers constructeurs avec une seule méthode en utilisant des valeurs par défaut aux arguments n'est-elle pas une très bonne idée ?

[Réponse P4.2]

Si l'on n'écrit qu'avec une seule méthode, on n'a pas une vision explicite de tout ce qu'il faut initialiser.

De plus, lors d'une telle écriture, on définit en fait trois constructeurs et non pas deux comme attendu, c'est-à-dire qu'on fait effectivement un constructeur par défaut ne prenant aucun argument, et le constructeur a deux arguments explicites, mais en plus, on crée sans le vouloir un constructeur à un argument.

En effet, lors d'une écriture *constructeur(param1 = truc, param2 = machin)*, l'appelle à *constructeur(qqch)* construira belle et bien une instance dont le 2ème attribut aura la valeur par défaut *machin.*

[Question P4.3]

Si l'on souhaitait ajouter un constructeur par coordonnées sphériques (deux angles et une longueur),

a) que cela impliquerait-il au niveau des attributs de la classe ?

b) quelle serait la difficulté majeure (voire l'impossibilité) de sa réalisation en C++ ? (C'est d'ailleurs pour cela qu'on ne vous demande pas de faire un tel constructeur !)

[Réponse P4.3]

D'après ce que j'ai compris :

a) Cela n'impliquerait rien du tout au niveau des attributs : l'encapsulation de la classe fait qu'on peut avoir les constructeurs qu'on veut indépendamment de l'implémentation choisie.

C'est-à-dire que le fait qu'on ait choisi de représenter nos vecteurs avec trois coordonnées cartésiennes à l'intérieur de la classe, n'importe pas à l'abruti qui veut créer un vecteur à l'extérieur de la classe avec des coordonnées sphériques (encapsulation). Concrètement, il suffit, dans le constructeur sphérique qu'on souhaite ajouter, convertir les coordonnées sphériques en cartésiennes (avec les formules vues en physiques) avant d'initialiser les attributs.

b) Question foireuse.

Il y a plusieurs inconvénients.

Premièrement, les constructeurs cartésiens et sphériques auraient tous les deux trois arguments de type double, ce qui fait qu'on (et le compilateur aussi) ne peut pas les différencier, mais if suffirait d'ajouter un 4ème paramètre « inutile » qui ne servirait qu'à faire la distinction entre les 2 constructeur. Par exemple *constructeursphérique(…, …, …, bool inutile)*.

Deuxièmement, le fait de fournir un constructeur sphérique implique qu'on offre la possibilité a l'utilisateur de la classe de faire ses raisonnement en coordonnées sphériques, par conséquent, il faut aussi fournir toute l'interface en version sphérique, c'est-à-dire des accesseurs getPhi(), getTeta(), getR() et l'affichage en coordonnées sphériques. Là encore, ce n'est qu'un inconvénient.

Quelle andouille voudrait passer en coordonnée sphérique !

[Question P7.1]

En termes de POO quel est donc la nature de la méthode dessine ?

[Réponse P7.1]

C'est une méthode virtuelle, on veut que l'instance se dessine comme sa nature réel

[Question P8.1]

A quoi faut-il faire attention pour les classes contenant des pointeurs ? Quelle(s) solution(s) est/sont envisageable(s) ?

[Réponse P8.1]

Toute zone mémoire allouée par un « new » doit impérativement être libérée par un « delete » correspondant (par exemple dans le destructeur). Il faut faire suivre tous les « delete » par l'instruction « pointeur = nullptr ». Lorsqu'on ajoute un pointeur a un tableau, il faut vérifier qu'il pointe bien sur quelque chose (ie : différente de nullptr).

Une solution aurait été d'utiliser les smartpointers, mais on se sent plus à l'aise avec les pointeurs « à la C ». De plus pour P14 il y aura plusieurs pointeurs qui pointeront sur un même objet (les particules en l'occurrence). Par conséquent, l'utilisation des « unique_ptr » n'aurait pas pu faire l'affaire, et nous n'avons pas vraiment vu les « shared_ptr ». C'est pourquoi l'utilisation des pointeurs « à la C » n'est pas une mauvaise idée (lorsqu'on est assez rigoureux).

Néanmoins, on peut quand même s'en sortir dans le P14 (tout en utilisant les smartpointers) en utilisant les indices de la collection hétérogène.

[Question P8.2]

Comment représentez-vous la classe Systeme ?

[Réponse P8.2]

Le système a comme attributs une collection (hétérogène) de particule (=> pointeurs), et une enceinte (à noter qu'à partir de P11 nous avons un pointeur sur une enceinte) ainsi qu'un double pour la température du système.

L'interface offre comme méthode public des constructeurs et un destructeur (cf. question P8.3), une méthode pour ajouter des particules au système, une méthode pour supprimer toutes les particules du système, une méthode « evolue » qui se charge de faire la simulation, et une méthode dessine qui appelle les méthodes dessine de toutes les particules du système et de l'enceinte. On imagine aussi qu'on puisse vouloir chauffer ou refroidir le système, donc on a mis une méthode pour modifier l'attribut température.

Le Système a aussi des méthodes « outils » (en private) qui ne sont pas offert à l'utilisateur de la classe, comme les méthodes pour gérer les chocs, mettre à jour les particules lorsqu'il y a un choc, et gérer les rebonds contre des parois de l'enceinte

[Question P8.3]

Pourquoi fait-on cela ? A quoi peuvent bien servir ces deux méthodes privées et «= delete» ?

[Réponse P8.3]

Pour empêcher les copies du système, on a supprimé la possibilité d'utiliser le constructeur de copie et l'opérateur d'affectation (les copies du système sont à éviter, car le système est potentiellement très grand, et en faire une copie serait une opération lourd en ressources).

[Question P8.4]

Avez-vous mis une méthode pour détruire toutes les particules ? Y avez-vous fait appel dans le destructeur ?

[Réponse P8.4]

Oui car c'est demandé (-_-'), cela évite a l'utilisateur de la classe de devoir gérer les « delete », il lui suffit d'appeler la méthode. Nous y avons fait appel dans le destructeur car cela empêche d'oublier de libérer la mémoire alloué par de « new » avant de détruire le système.

[Question P9.1]

Comment (et à quel(s) endroit(s)) intégrez vous ces deux aspects ?

[Réponse P9.1]

A ce stade du projet, on fait l'initialisation dans le fichier test, manuellement en spécifiant les positions et vitesses initiales de chaque particule de notre système. Pour le $2^{ème}$ aspect on a fait une méthode évolue système, qui fait évoluer tout le système (ie : on appelle les méthodes évolue des particules du système) d'un pas de temps dt.

[Question P9.2]

Quelle est la complexité de l'algorithme de simulation lorsqu'il utilise cette méthode pour savoir si deux particules se rencontrent ?

[Réponse P9.2]

2 boucles sur toutes les particules du système imbriquée => La complexité est quadratique ($O(n^2)$). Ce n'est pas top.

[Question P11.1]

Si vous souhaitez voir l'enceinte, i.e. la faire dessiner, comment devez vous modifier votre conception/votre code ?

[Réponse P11.1]

En gros, on a fait hériter la classe « Enceinte » de « Dessinable » (une enceinte est déssinable) ce qui la rend abstraite, on a créé une sous classe « Glenceinte » avec les instructions de dessin graphique (on a aussi créé la sous-classe « TXTenceinte » pour la simulation en mode texte, avec une méthode dessine qui ne fait pour l'instant rien, mais on imagine qu'on pourrait vouloir afficher l'enceinte dans le terminal, par exemple si on lui fait subir des changements). On a troqué l'attribut du système « enceinte » de type « Enceinte », pour un pointeur sur une « Enceinte » pour avoir un comportement polymorphique au niveau de la méthode dessine de l'enceinte.

[Question P13.1]

Comment représentez-vous cette/ces nouvelle(s) particule(s) ? Où s'incri(ven)t-elle(s) dans votre conception ?

[Réponse P13.1]

Tout d'abord on s'était dit qu'on allait faire des classes spécialisée des particules graphiques, mais cela impliquait de faire une sous classe par particule graphique différent… notre flemmardise a pris le dessus et on a fait autrement. On fait une superclasse « Traçable » et nous avons faire de l'héritage multiple pour les particules graphique : la particule hérite de Traçable (au sens « elle a la possibilité d'être tracée ») et Dessinable (via la classe intermédiaire « Particule »). On a modifié les constructeurs des particules graphiques pour prend tout ceci en compte, si on veut que la particule soit tracée il faut passer la longueur de la trace au constructeur.

[Question P14.1]

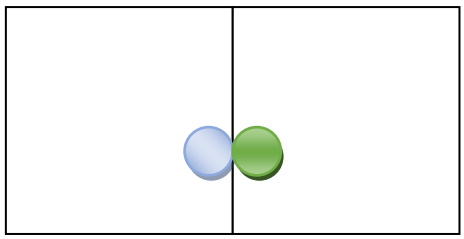
Avant de préciser les détails d'implémentation, quelle est la complexité temporelle pire cas de cette solution en fonction du nombre de particules ? [Attention : on suppose, par hypothèse même du modèle de gaz parfait, que les particules sont équiréparties dans l'enceinte. On suppose de plus que la taille d'une case est petite par rapport à la taille de l'enceinte. Ainsi on peut raisonnablement faire l'hypothèse que le nombre de particules par case est négligeable (O(1)) devant le nombre total de particules (i.e. toutes les particules ne se retrouvent pas en même temps dans la même case).

Quel(s) inconvénient(s) présente cependant cette solution ?

[Réponse P14.1]

Complexité temporelle pire cas : O(n) (complexité linéaire), puisqu'il y a deux boucle imbriquée, mais que la seconde (qui parcours la cases) est négligeable.

Inconvénients : Il y aura un problème concernant les collisions. Par exemple, s'il y a collision entre deux particules, mais elles ne sont pas dans la même case, cette collision ne sera pas pris en considération :



[Question P14.2]

Comment et où avez-vous implémenté cette nouvelle façon de calculer les collisions ?

[Réponse P14.2]

On a implémenté dans système où il y avait l'ancienne façon. On a choisi comme convention que chaque particule accède à sa case via les valeurs entières de ses coordonnées de position. Au lieu de parcourir toutes les cases pour voir si elles contiennent une particule ou pas et ensuite gérer tous les chocs a l'intérieure de la case, on a décidé de parcourir la collection de particule pour voir dans quelle case elle est, parce qu'il y a beaucoup moins de particules (~10^3 particules) que de case dans une enceinte 100x100x100 (10^6 cases).