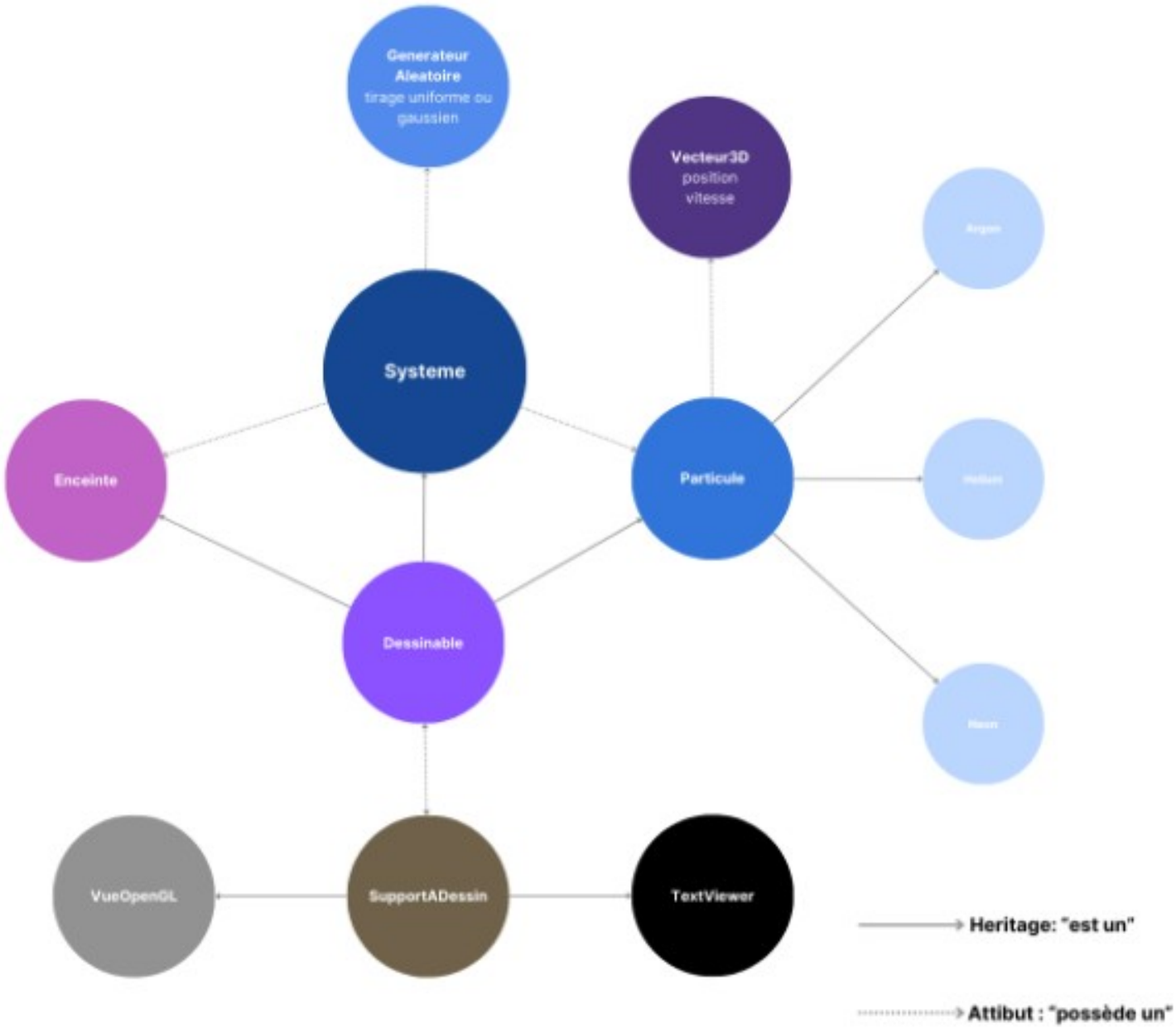
Emmanuel Ponti et Justin Pessia

Groupe 139

CONCEPTION

Liens entre classes :



Description et contenu des classes

Vecteur3D :

Attributs : On a mis des double x, y, z en private pour conserver l'encapsulation, vecteur3D n'a pas de sous-classes.

Méthodes : Un constructeur par défaut qui initialise le vecteur nul, un constructeur qui affecte 3 double en pris en parametres à ses attributs.

Les méthodes d'affichage, compare, addition, soustraction, oppose_n, oppose, mult, prod_scal, prod_vect, norme, norme2 et unitaire comme demandé lors des premières semaines, qui permettent aussi de surcharger les opérateurs.

On a par la suite ajouté 3 autres méthodes par nécessité :
2 accesseurs get_coord et compare2. Get_coord permet d'obtenir les coordonnées d'un vecteur dont on se sert dans les méthodes évolues de particules pour tester des postions, compare2 permet de faire un pavage cubique pour la methode evolue (pas la 2eme mais bien la première) et ainsi pouvoir vérifier si 2 particules se rencontrent.

On a aussi ajouté un setter : set_coord nécessaire pour replacer une particule qui sort de l'enceinte avec la methode choc_paroi.
Comme expliqué dans le fichier REPONSE nous avons implémenté un constructeur de copie car celui-ci peut s'avérer utile mais nous ne nous en servons pas dans le projet.

Particule :

Attributs : En protected : une position ainsi qu'une vitesse representé par des vecteurs 3D, une masse, en protected car on les manipule dans la méthode initialise_rd des sous classes.
En private : pour la partie « trace » qu'on visualise graphiquement on a ajouté une booléen trace qui si vaut false afficher la trace de la particule, cela en utilisant une double ended queue de vecteur3D (il faut donc include la bibliotheque), une queue classique suffisait mais j'ai choisi arbitrairement d'utiliser des deque, afin de stocker les positions au fil du temps.

Il y a 2 attributs de classe : R la constante des gazs parfaits utilisé pour définir les vitesses gaussienne dans initialise_rd des sous classes et precision qui est aussi dans Enceinte.h pour modifier la précision c'est-à-dire le nombre de cases dans l'enceinte lorsqu'on utilise la methode evolue2 de Système.
Attention si vous modifiez un des deux, il faut modifier l'autre, elles doivent toujours avoir la même valeur

Méthodes : Un constructeur qui initialise chaque attribut ainsi que la trace fixé par défaut à false et un autre qui fait appel au constructeur par défaut de vecteur3D pour position et vitesse, pour initialiser une particule il faut donc au moins lui donner une masse.

Le destructeur a été défini comme virtuel car nous travaillons avec des smart pointers sur particule dans la classe Système.

Dans la methode affiche nous utilisons la bibliotheque iomanip afin d'augmenter la précision d'affichage des données notamment de la masse de l'helium, on utilise aussi la surcharde de l'operateur << pour afficher les vecteur3D

La methode choc paroi appelle la méthode choc_paroi_n tant que la particule n'est pas dans l'enceinte.
La methode choc_paroi_n prend en parametre l'index de la particule dans le système, le chiffre 0 1 ou 2 correspondant respectivement a la largeur, la hauteur et la profondeur x, y, z comme pour la methode Vecteur3D::get_coord, on passe aussi en parametre la taille du cote de l'enceinte correspondant. Cette méthode prend le sysmétrique de la position par rapport à l'axe correspondant. Enfin on utilise la methode oppose_n qui permet de prendre l'opposé d'une des coordonnées de la vitesse suite au rebond

On a 2 methodes evolue surchargées:
La premiere correspond à la simple evolution d'une particule en actualisant sa position en fonction de sa vitesse et qui vérifie aussi si la particule doit afficher une trace. Si oui, alors on ajoute la position courante à la deque et on enleve la plus ancienne par un pop back pour avoir une trace plus esthétique.

La deuxieme méthode evolue qui prend trois parametres a savoir le pas de temps, l'index i de la particule et l'enceinte en référence car on la modifie.
Cette méthode est utilisé pour gérer les chocs de manière différente en regardant si deux particules sont dans une meme case. Ici on calcule la case dans laquelle devrait se trouver la particule courante en prenant l'arrondie inférieur de la position de la particule selon ses trois coordonnées.
Puis on l'efface de la case courante avec la méthode Enceinte ::efface_particule décrite plus loin.
On fait ensuite appel à la premiere méthode particule pour la déplacer puis on recalcule la méthode dans laquelle elle devrait se trouver et on la met avec la méthode Enceinte::ajoute_index.
A noter que nous rappelons la méthode choc paroi avant de remettre la particule dans sa case car ci celle-ci est sortie de l'enceinte lors de l'appel de l'autre methode evolue alors il y aura un segmentation fault.

La méthode test position verifie si deux paticules se rencontre selon un pavage cubique.

Les méthodes calcule_vg et calcule_L sont gérer le choc entre 2 particules et leur nouvelle vitesse

La méthode afficher choc affiche les données de deux particules qui se sont rentrés dedans ou vont se rentrer dedans.

Enfin on surcharge aussi l'operateur << à l'aide de la méthode affiche

Argon, Helium, Néon :
Attributs : Un constructeur qui appel celui de Particule car ces classes heritent de particules comme montré sur le schéma
Méthodes: Override de la méthode affiche de particule en écrivant le gaz dont il s'agit et de la méthode dessine_sur qui appelle dessine de support à dessin sur l'instance courante.

La méthode initialise_rd qui permet en prenant un nombre de particules et une masse commune pour chacune d'entre elles ainsi que le systeme par référence d'initialiser avec une position suivant un tirage uniforme et une vitesse suivant une loi gaussienne.

Système:
Attributs: tout est en private :

Un vector de unique ptr sur particule pour stocker l'ensemble des « particules » (pointeurs)
Une enceinte, un generateur aléatoire, un epsilon pour le pas d'espace, une temperature par défaut à 290 K, je ne la met pas dans le constructeur car je préfere la modifier au travers d'un setter. On a aussi un booléen forcer pour indiquer si il faut forcer le premier choc avec un azimut et zenith de pi/2 et pi/3

Méthodes : 2 constructeurs : un qui initialise le pas d'espace et les dimensions de l'enceinte, un autre qui fait appel au constructeur par défaut de l'enceinte.
Set_tirage qui permet d'enlever l'aléatoire en parametrant une graine pour debugger
Set forcage qui prend un booléen pour force le premier choc en mettant forcage sur true
Le destructeur de systeme est virtuel car il utilise du polymorphisme.
Une methode ajoute et supprime particule qui crée dynamiquement un unique ptr et l'ajoute l'autre vide le tableau.
La méthode affiche permet d'afficher le système textuellement en donnant les dimensions de l'enceinte et en affichant chaque particule.

Les méthodes evolue et evolue2 font globalement la meme chose mais d'une manière différente. Les 2 déplacent chaque particule et gerent les chocs contre les parois a l'aide des méthode de Particule. En revanche evolue teste sur chaque paire de particule si elles sont assez proche pour se rencontrés. Pour une meme particule on met alors toutes les particules qui sont candidates à un choc avec cette dernière dans un vector candidats puis si celui-ci contient plus d'une particule alors on en tire une au hasard qui rencontrera la première.
Alors que evolue2 regarde dans chaque case de l'enceinte si celle-ci contient une particule ou plus ce qui signiferait qu'elles sont assez proches pour se rentrer dedans suivant la precision voulue.
Il faut en plus enlever l'index de la particule dans la case courante, actualiser la postion de la particule puis remettre son index dans la nouvelle case ce que fait la deuxieme methode particule :: evolue décrite précedemment

La méthode evolue_choc est utilisée par les 2 méthodes evolue de système pour afficher le texte voulue lorsque 2 particules se rencontrent et calculer leur nouvelle vitesse à l'aide du tirage uniforme.

La methode affiche_systeme fait une simple boucle sur toutes les particules du système

La methode position_rd est appelée depuis les différentes sous classes de Particule afin de faire un tirage uniforme pour initialiser la position 0 : x, 1 : y ou 2 : z.
On doit utiliser un switch car la valeur dépend de la taille de l'enceinte selon le coté.
La méthode vitesse_rd elle renvoie juste un double d'un tirage gaussien.

Les méthodes initialise_rd_*gaz* permette l'initialisation de plusieurs particule d'un meme gaz dans le système. C'est la manière dont j'ai décidé de l'implémenter : on crée un système puis l'on y ajoute 20 particules de Neon de masse 4 en ecrivant initialise_rd_neon(20,4) cette méthode crée un particule inutile qui permet d'appeler la méthode initialise_rd qui permet de manipuler les vecteurs position et vitesse sans casser l'encapsulation et on rappelle les méthode position/vitesse_rd pour avoir accès au générateur aléatoire.

Enceinte:
Attributs: Une enceinte a : une largeur(x) une profondeur(y) et une hauteur(z), un tableau tridimensionnel (Tab_Cases) et une précision (comme dans Particule.h)

Methodes: Le constructeur d'enceinte initialise les 3 premiers attributs avec une valeur par défaut de 20 puis teste d'abord si toutes les longueurs sont bien positive et ensuite appelle la méthode redimensionne en private en lui passant des unsigned int (conversion faite grace a

static cast qui prend la valeur arrondi par défaut (comme floor en terme de conversion double -> unsigned int))

La méthode redimensionne prend donc 3 unsigned int en parametre et crée des vecteurs de dimension a b c, où a b et c on était multiplié par la precision que l'on souhaite donner.

La méthode affiche sort les dimensions de l'enceinte.
Ensuite les 2 méthodes restantes sont celles utilisées par la méthode evolue de Particule :
La méthode ajoute particule prend l'index de la particule dans particules de système ainsi que les numéros de la case correspondante.

La méthode efface particule prend les memes parametres mais c'est plus compliqué :
On aurait pu simplement faire un .clear() de la case mais j'ai préféré les effacer au fur et à mesure comme ça il peut y avoir des chocs entre une particule qui vient d'être déplacé et une particule qui n'a pas encore été déplacée.
Pour se faire on recherche l'index de la particule dans la case avec find(), si il est bien présent on l'efface, sinon on cout une erreur.

Enfin, nous avons surchargé l'operateur << en appelant la méthode affiche.

Generateur Aleatoire:
Il y a un attribut générateur, un de distribution uniforme et un de distribution gaussienne. Le constructeur par défaut est celui que nous appelons dans les main mais on peut aussi passer une graine en parametre pour debugger. Enfin il y a deux méthodes de tirage :
Le tirage uniforme qui prend une valeur min et une max et une gaussienne qui prend une moyenne et un ecart type

Dessinable

Il s'agit d'une classe abstraite qui contient une méthode virtuelle pure dessine_sur(). Cette méthode sera surchargée dans toutes ses sous-classes.

SupportADessin

Dans son fichier .h, nous déclarons les différentes classes dessinables sans les initialiser afin d'éviter une dépendance cyclique. Elles seront initialisées plus tard. La classe SupportADessin possède un destructeur virtuel et cinq méthodes virtuelles pures qui seront surchargées dans ses sous-classes. Chacune de ces méthodes sera appelée dans la méthode dessine_sur() des sous-classes de Dessinable respectives au paramètre de dessine().

Vue OpenGL

VueOpenGL est l'interface graphique sur laquelle apparaîtra la simulation. Elle hérite de SupportADessin. Les surcharges des méthodes dessine() ont les utilités suivantes :

- **Pour une Enceinte :** Grâce à sa méthode dessine_enceinte(), elle dessine les arêtes d'un rectangle de la taille de l'enceinte, permettant ainsi un effet de transparence.
- **Pour un Néon :** Grâce à sa méthode dessineCube(), elle dessine un cube opaque.
- **Pour un Argon :** Grâce à sa méthode dessine_pyramide(), elle dessine une pyramide.
- **Pour un Hélium :** Grâce à sa méthode dessine_sphere(), elle dessine une sphère.

Pour chaque particule, VueOpenGL peut dessiner des sphères dont la couleur dépend de la vitesse de la particule, si l'attribut booléen couleur de la classe Système est vrai.

Elle peut également dessiner ou non la trace d'une particule en fonction de l'attribut booléen trace de celle-ci.

Méthodes supplémentaires de VueOpenGL

VueOpenGL possède aussi les méthodes init() et initialise_position() pour initialiser la vue de la caméra et l'affichage.

Enfin, elle a l'attribut matrice_vue qui représente la vue affichée à l'écran.

TextViewer

Dans son fichier .h, on déclare la surcharge de l'opérateur << avec un ostream pour chacune des sous-classes de Dessinable, que l'on définira plus tard.

Attributs et Fonctionnalités

Cette classe a un ostream en attribut. Si cet attribut est par exemple cout, cela permettra un affichage textuel lors de l'exécution.

Méthodes de TextViewer

Les surcharges des méthodes dessine() utilisent les méthodes affiche() des sous-classes de Dessinable, ainsi que leur surcharge de l'opérateur << pour ajouter un élément au flot. Ce flot sera affiché lors de l'appel de l'une des méthodes dessine_sur() des sous-classes de Dessinable.

ChatGPT

GLWidget

GLWidget est une classe dérivée de QOpenGLWidget qui gère l'affichage et l'interaction avec une simulation graphique OpenGL. Elle réimplémente les méthodes clés initializeGL(), resizeGL(), et paintGL() pour configurer et dessiner la scène OpenGL. La classe possède un attribut représentant le système à afficher lors de la simulation, et elle permet d'influencer ce système via des événements clavier. Par exemple, des extensions ont été ajoutées pour changer la couleur, ajuster le pas de temps et afficher ou masquer les traces des particules. De plus, un signal switchWindow est émis lors d'un événement clavier et récupéré dans le main pour ouvrir un autre widget de type GLWidgetDeux.

GLWidgetDeux

GLWidgetDeux est une sous-classe de GLWidget. Cette approche permet de réutiliser les méthodes d'initialisation sans avoir à les redéfinir. En plus, GLWidgetDeux possède une référence à un GLWidget pour pouvoir recueillir des informations (énergie cinétique, nombre de chocs, pression, pas de temps) sur le système contenu dans ce widget. Grâce à sa surcharge de la méthode paintGL(), GLWidgetDeux peut afficher ces informations dans une autre fenêtre.

Discussion et améliorations possibles

La distribution aléatoire des particules dans notre simulation est en accord avec la physique statistique, où les propriétés macroscopiques comme la pression et la température émergent des comportements collectifs des particules. La simulation permet d'étudier des concepts comme l'entropie, la loi des gaz parfaits ou encore l'énergie libre. Cependant, l'absence de réactions chimiques et de transitions de phases ,qui devraient être engendrés par le comportement des particules ,permet pas une simulation toujours conforme à la réalité surtout au cas limites.