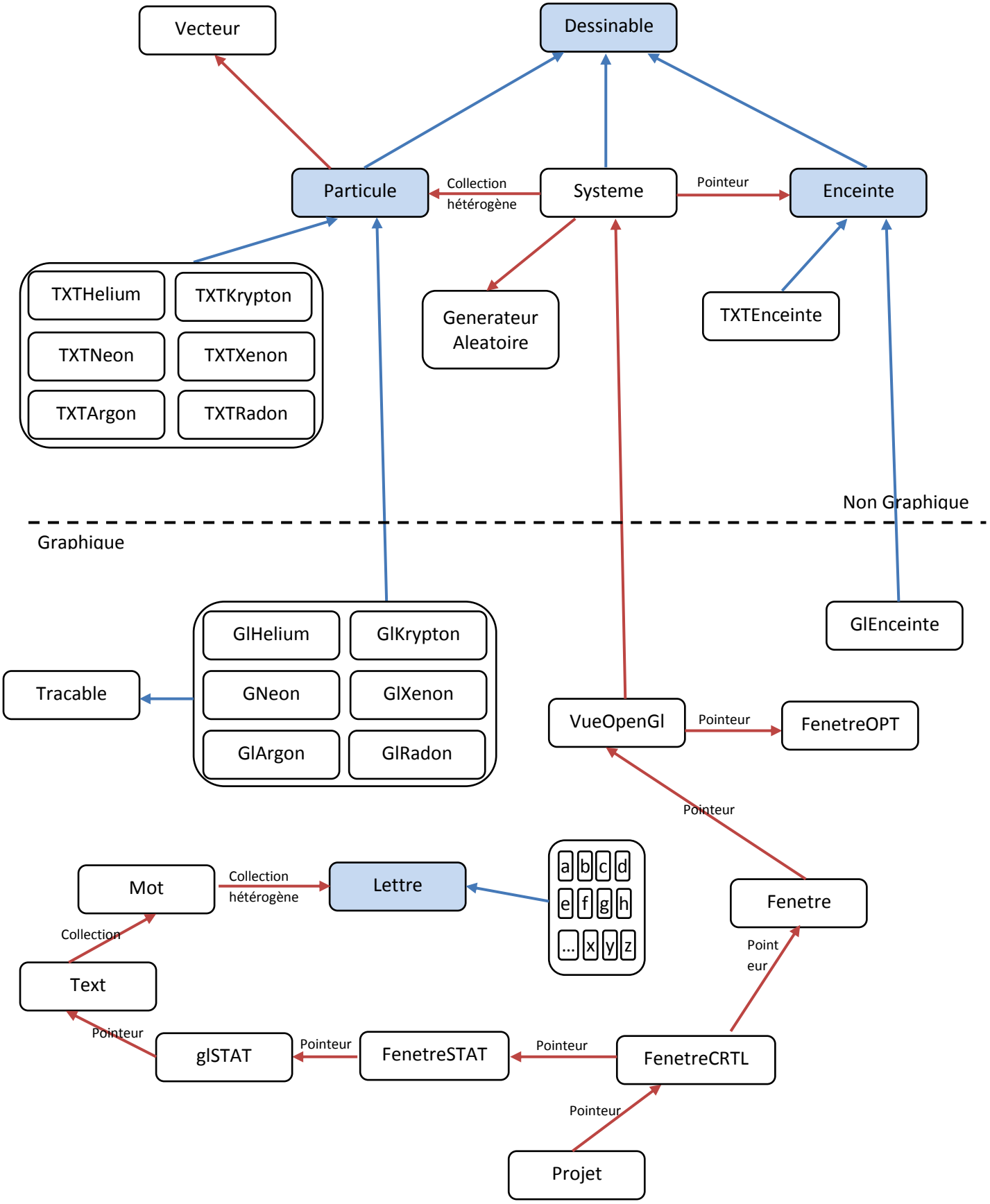
Nguyen Thanh-Mai Christine Section Math Groupe n°55
Sekarski Samuel Section Math Groupe n°55

CONCEPTION



Ci-dessus est la HIÉRARCHIE DE CLASSES du Projet (avec les bonus). Lorsque l'encapsulation ne se fait pas de manière basique, on a ajouté la précision au-dessus de la flèche (par exemple « pointeur »). Voici la légende (qui ne tient plus sur la même page que le diagramme)

Légende :

Relations d'héritage EST UN:
Sous-Classe --->Super-Classe
« La sousclasse hérite de la superclasse »

Relation d'encapsulation A UN/DES :
Classe X --->Classe Y
« La classe x possède un/des (pointeurs sur) y »

Lettre Signale que la classe est abstraite

POINTS IMPORTANTS, EXPLICATIONS ET JUSTIFICATION DE NOS CHOIX :

Vecteur :

La Classe **vecteur** a trois attributs **double** représentant les coordonnées cartésiennes d'un vecteur (libre), car ça nous simplifie la tâche dans le reste de notre programme. On n'offre pas la possibilité à l'utilisateur d'utiliser les coordonnées sphériques.

Etant donné que c'est la classe qui va être utilisée le plus dans notre programme, nous avons cherché à l'optimiser au mieux.

Dessinable :

Classe abstraite avec seulement une méthode virtuelle pure « dessine ».

Particule :

Cette classe possède deux **vecteurs**, respectivement pour la position et la vitesse de la particule, ainsi qu'un **double** pour la masse de cette particule. A noter que le constructeur par défaut initialise une particule d'hélium immobile à l'origine, c'est-à-dire qu'il ne devrait pas être possible d'initialiser une particule de masse nul (photon ?) => masse de l'hélium par défaut. Par contre, la position peut très bien être nulle par rapport à un certain repère (coin de l'enceinte), idem pour la vitesse.

Cette classe hérite de **Dessinable** (une particule est dessinable). Par contre, on ne redéfinit pas la méthode dessine (pour forcer la redéfinition dans les sous classes de **Particule**s), donc **Particule** reste une classe abstraite (non instanciable). On veut aussi un comportement polymorphique sur les particules. Par conséquent, beaucoup des méthodes sont virtuelles.

Nous avons aussi hésité à mettre l'attribut « masse » en **const**, car on ne simule pas de fusion/fission nucléaire, et même si telle était le cas on pourrait considérer que ce sont de nouvelles particules qui sont formées. Mais au final, on a laissé sans, au cas où on voudrait utiliser la classe a d'autre fins. (En plus, le gaz noble « Radon » qu'on utilise est radioactif, donc sa masse devrait décroître en fonction du temps… (Avec une demi-vie d'une heure à trois jours, dépendant les

isotopes)[1]. On n'a pas pris ceci en compte dans la simulation, car on s'est dit que c'était trop ambitieux.

Nous avons mis les méthodes de rencontres de particules (les anciennes, pavage cubique et sphérique) dans cette classe, car tester si une particule est epsilon proche d'une autre, nous semblait être une propriété de la classe **Particule**, et pas seulement du **Système**.

TXTGaz :

Ces sous classes sont des particules spécialisées (héritant de **Particule**) sans attribut supplémentaire, mais avec une redéfinition de « dessine » pour l'affichage dans un flux de sortie (par défaut « cout » sur le terminal). Nous avons mis, dans les constructeurs, en paramètre par défaut, leur masse atomique standard[2].(On a mis les masses en paramètre par défaut, plutôt que d'initialiser automatiquement leur masse atomique standard, car on imagine qu'on puisse vouloir initialiser d'autres isotopes du même gaz (qui ont évidemment des masses différentes)).

On a réfléchit pour la « constante spécifique » (R/M*1000) de ces gaz, et au début, on voulait en faire un attribut constant de classe, **static const**, car chaque gaz (hélium, neon, argon etc..) a exactement la même constante (vu qu'elle a la même masse a priori) et il serait inutile de la copier dans chaque instance des sous-classes, mais on a opté pour une « variable de préprocesseur » (#define), car on utilise cette constante qu'une fois pour l'initialisation du système, puis elle prend juste de la place en mémoire pour rien. De plus, avec la remarque ci-dessus (la possibilité d'avoir d'autres isotopes), il pourrait y avoir deux particules d'un même gaz avec des masses différente, donc des constantes différentes.

Enceinte :

La classe **Enceinte** est une classe abstraite héritant de **Dessinable** (une enceinte est dessinable). On veut un comportement polymorphique de cette classe. Sinon, rien de spécial à en dire, le constructeur par défaut initialise une enceinte carrée de dimension 20x20x20 ou une unité est égal à 0,1 nm (dit dans la consigne). Ses attributs sont trois **double** et on offre trois getters et trois setters (il y a la possibilité d'avoir des parois mobiles) à l'utilisateur de la classe.

TXTEnceinte :

Classe spécialisée d'**Enceinte**, dédiée à l'affiche en « mode text », sans attribut supplémentaire, et avec une redéfinition de la méthode dessine, qui ne fait rien (On sait, On sait ! ce n'est pas bien, mais on imaginait vouloir afficher quelque chose sur le terminal si on modifie les dimensions de l'enceinte en cours de route. D'ailleurs, si on a encore du temps quand on aura fini d'écrire ceci on le fera, comme ça il y aura bien une redéfinition. (En plus ça nous arrange parce qu'on veut que **TXTEnceinte** soit instanciable…)).

Système :

La Classe **Système**, la dernière classe qui n'est pas étroitement liée au graphisme (mis a par **GenerateurAléatoire**). Préparez-vous pour un pavé. C'est la classe centrale du projet, le cerveau de la simulation si vous voulez. Cette classe possède en attributs une collection hétérogène de **Particule**, un pointeur sur une **Enceinte** (on veut du polymorphisme sur **Enceinte**), une instance de

[1]http://en.wikipedia.org/wiki/Radon
[2]http://en.wikipedia.org/wiki/Noble_gas_%28data_page%29

GenerateurAléatoire (pour faire des tirages aléatoire de nombres), un **double** pour la température du système, et finalement un « tableau tridimensionnel de list » (simplifié avec des **typdefs**) pour les « cases » de l'espace. (à noter aussi que **Système** hérite de **Dessinable**, avec une redéfinition de la méthode dessine).

Conceptuellement parlant, on a choisi de faire des cases un attribut du **Système** et non pas un attribut d'**Enceinte**, car les cases sont un échantillonnage de l'espace, qui ne dépendent pas forcément de l'enceinte. C'est-à-dire imaginons qu'on veuille faire une enceinte percée, pour que les particules puissent s'échapper, les cases devront englober plus que juste l'enceinte, car il pourrait y avoir des chocs a l'extérieure de l'enceinte. (Aussi nous n'avons pas jugé nécessaire de créer une classe « cases »).

Comme le système peut comporter un grand nombre d'élément, on ne veut pas qu'il soit copiable, on a donc enlevé l'opérateur d'affectation et le constructeur de copie.

Au niveau implémentation on a un certain nombre de méthode privée (dite « outils ») qu'on n'offre pas à l'utilisateur et qui simplifie l'écriture des autres méthodes de la classe. Nous avons une méthode pour gérer les rebonds contre les parois de l'enceinte, deux méthodes pour gérer les chocs entre particules (une ancienne et une nouvelle plus performante (avec les cases)), toutes surchargée pour permettre un affichage dans le terminal par exemple. Puis il y a encore une méthode qui met à jour 2 **Particule** lorsqu'elle se rencontre, et une version truquée de celle-ci pour débugger (à cause de l'aléatoire). Et finalement 1 méthode pour « remplir les cases » (c'est-à-dire associer une case aux **Particule** qu'elle contient). Le vidage de cases se fait par contre directement dans la méthode qui gère les chocs.

Il y a ensuite les méthodes publics : ajouter/supprimer des particules, un certain nombre de getters, constructeurs/destructeur, et évidemment les méthodes « évolue » (ancienne et nouvelle, surchargée pour l'affichage TXT). De plus il y a une méthode pour tirer aléatoirement une position et une vitesse de particule. (On fait l'initialisation de la collection de **Particule** en dehors de la classe)

A noter qu'on passe des pointeurs dans la méthode « ajoute » des particules et dans le constructeur pour l'enceinte. Ceci implique quelques précautions au niveau de l'utilisateur de la classe ☹ : il faut allouer dynamiquement l'enceinte dans le constructeur, et dynamiquement les particules dans la méthode ajouter_particules, sinon on pourrait avoir des effets non voulu (si on passe, par exemple, l'adresse d'un objet cité ci-dessus qui existe déjà, et qu'on supprime cet objet, alors le pointeur à l'intérieure du **Système** ne pointerais plus sur rien….). Une solution aurait été de faire une copie profonde et transmettre l'appartenance de ses objets au **Système**. (Mais faute de temps, nous ne l'avons pas fait).

Sinon sans trop entrer dans les détails, la méthode « dessine » redéfini appelle les méthodes « dessine » propres à chaque élément du système, l'ancienne méthode « évolue » appelle d'abord les méthodes « evolue » de chaque particules (les déplace d'un MRUA quoi), puis gère les rebonds contre les parois de l'enceinte (effectivement en faisant une symétrie de toutes les **Particule** qui serait allé au-delà de l'enceinte au point 1), et finalement fait les chocs inter-particules, de toute particules en situation de choc.

La nouvelle version d'évolue fait globalement les mêmes choses, sauf qu'en plus elle remplit les cases au début de chaque appelle, et la gestion des chocs est différente : on regarde la cases de chaque particule, on fait le choc avec toutes les autres particules a l'intérieure de la case et on la supprime de sa case.

Tracable :

Superclasse contenant un tableau de **Vecteur**, pour garder en mémoire, les positions passée d'une particule, une méthode pour ajouter une position aux tableaux, et une méthode pour tracer la trace.

GlGaz :

Ces classes sont les équivalents des **TXTgaz** mais pour l'affichage graphique des particules. Ces classes héritent multiplement de **Particule** et de **Tracable**. Mêmes remarques que pour **TXTgaz**.
A noter qu'on dessine les particules en « fil de fer » ce qui permet de mieux visualiser la 3D, l'autre chose et qu'on a divisé la taille des particules par 4.

En effet en cherchant[3] les rayons approximatifs des gaz nobles on arrive à des rayons entre 100-200pm (10^{-12} => $1x10^{-10}$) or on nous dit (consigne pour **Enceinte**) qu'une unité de **double** valait 0.1 nm (10^{-9}=> $1x10^{-10}$), du coup sur 20unités on ne pourrait en mettre qu'une dizaine cote a cote et le fait qu'on utilise le centre de masse pour faire toute la simulation est très voyant : on a des particules qui sortent a moitié de l'enceinte (pareil pour les choc entre particules). D'où le fait d'avoir rapetissé un peu les rayons.

GlEnceinte :

Rien de particulier à dire là-dessus, cette classe hérite d'**Enceinte** et redéfini la méthode « dessine » pour l'affichage graphique

Projet :

Cette classe est l'application en tant que tel, avec la boucle infinie. Cette classe ne contient qu'une méthode qui initialise une fenêtre de menu (appelée **fenetreCTRL**), allouée dynamiquement comme pour toutes les instances de classes de wxWidgets. Elle hérite de **wxApp**.

FenetreCTRL :

Cette classe est la fenêtre de Menu, elle permet de choisir quel modèle de choc utiliser, combiens de particules de chaque initialiser, puis de lancer la simulation par un bouton. Elle offre aussi un bouton pour lancer les crédits. Cette fenêtre hérite de wxWindow. Elle possède tous les attributs nécessaires pour interagir avec l'utilisateur comme nous le souhaitons.

C'est à dire, il y a des **Sliders** pour choisir les nombres de particules, qui sont géré par des méthodes (Handlers) lorsqu'un événement se produit (par exemple on déplace le slider), pour ça on a connecté les évènements avec les méthodes qui les gèrent. Bon euh, on ne va pas refaire tout le tutoriel sur le graphisme et la programmation évènementielle (vu que ce serait très long, et vous savez déjà tout ce qu'il y a dedans).

On utilise des **Sliders**, car leurs « valeur » sont déjà des **ints**, donc pas besoin de faire trop de conversion (comme si on utilisait des champs de texte) de plus on n'a pas besoin de faire plein de test pour savoir si ce que l'utilisateur a entré est valable (bornes, pas de lettres, etc…). On a fait en sorte que si un nombre maximum de particule prédéfini était atteint, les **Sliders** se bloquent. On s'est aussi arrangé pour que quand on modifie les **Sliders** ça affiche (avec un **statictext**)à coté la valeur du

[3] http://en.wikipedia.org/wiki/Noble_gas_%28data_page%29

slider (sinon on aurait absolument aucune idée de ce que représente une position x sur le slider. On a fait pour que chaque modèle de collisions ait un nombre maximum différent de particules (puisqu'on peut en avoir beaucoup plus avec le nouveau modèle).

Bref, et on a mis tout ceci dans plein de tableau imbriqué (des **Sizers**) pour la mise en page

On passe après tous les choix de l'utilisateur en paramètre au constructeur de **Fenêtre**, qui les passe au constructeur de **Vue_OpenGL** (effet cascade). Cf. dernière page pour la mise en page de la fenêtre de Menu avec tous ses attributs.

Fenetre :

Cette classe a pour seul but d'accueillir un contexte de dessin OpenGl, qu'elle alloue dans son constructeur.

A noter qu'on gère aussi nous-même la fermeture de la fenêtre, car si le timer de **VueOpenGl** tourne encore et qu'on ferme la fenêtre il y a un « segmentation fault ». On a donc déclaré une table d'évènement, qui connecte l'événement de fermeture (quand on clique sur la croix par exemple) a son Handler (~= manager ~=gestionnaire (ça se dit ?) enfin la méthode qui s'occupe de l'évènement quoi). Dans cette méthode on regarde en premier si le chronomètre tourne encore, et si c'est le cas, on l'arête, et la fenêtre se ferme normalement.

Vue_OpenGl :

Cette classe va s'occuper de dessiner le système, tous les X secondes (en l'occurrence 20ms), ou quand on demande un rafraichissement de l'écran (comme par exemple lorsqu'on appuie sur un bouton pour faire tourner l'enceinte.).

On a des méthodes correspondant : à l'appui de certaines touches qui font subir des transformations au système ; au timer ; et a la redimension de la fenêtre.

La Classe possède une sous fenêtre à option, dont nous décrierons la conception di-dessous, mais il faut savoir que dans les méthodes de **Vue_OpenGl**, il y a des modifications qui se font en fonction de ce que l'utilisateur fait dans la fenêtre de contrôle.

La classe fait l'initialisation du **Système** en fonction de ce qu'a choisi l'utilisateur dans la fenêtre de Menu. L'extension qui place la camera à la place d'une particule est codée dans la méthode dessine. (Bon on pourrait dire encore énormément de choses à propos de **Vu_OpenGl**, mais on va s'arrêter ici… « Conception » c'est un peu vague quand même…)

FenetreOPT :

Cette classe de fenêtre s'occupe d'interagir avec l'utilisateur pendant la simulation (au lieu d'avant, comme la fenêtre de menu). La mise en page, et les interactions sont plus ou moins du même principe que la fenêtre de Menu.

Conceptuellement ce qui change, c'est que **fenetreOPT** est une instance « enfant » (Child) de **Vue_OpenGL**, au lieu d'une instance « parent » comme l'était la fenêtre de menu. Dès lors, c'est la classe parente **Vue_OpenGl** qui va « venir chercher » ce dont elle a besoin de **FenetreOPT**. (Au lieu que ça soit la classe **Fenetre** qui « envoie » (par le constructeur) les données nécessaire a **Vue_OpenGL**)

Lettre :

Tout le reste concerne les crédits (un bonus/extension au projet), donc on va aller assez vite, et dire que les grandes lignes de la conception.

C'est une classe abstraite, avec une méthode virtuelle pure « dessine » qui sera redéfini dans les sous-classes de **Lettre**. Cette classe a été faite pour pouvoir faire une collection hétérogène de lettres. En y réfléchissant maintenant, on aurait pu faire hériter **Lettre** de **Désinable**, (mais ça rendrait le diagramme héritage de la première page un peu moche ^^, et on n'a pas envie de changer trop de code maintenant que notre projet est fini.)

A,b,c,d,…y,z :

Ces sont des sous classes de **Lettre**, chacune avec sa propre redéfinition de la méthode « dessine ». Ces classes correspondent aux lettres de l'alphabet, sans faire de différence entre les majuscules et les minuscules.

Mot :

La classe **Mot** possède une collection hétérogène de **Lettre** (car on veut du polymorphisme pour la méthode dessine), et une chaine de caractère qui lui correspond. Le constructeur initialise la chaine de caractère et appelle une méthode qui va remplir la collection avec les **Lettres** « graphique » (en claire faire le lien entre la chaine et sa représentation graphique).

Comme on utilise des pointeurs, il y a évidemment un destructeur qui libère la mémoire. Une méthode dessine est aussi présente, elle fait appel aux méthodes dessine de chaque **Lettre**, en faisant une translation entre chaque lettre (pour qu'elles s'écrivent de gauche à droite comme attendu et non pas empilée les unes sur les autre)

Text :

Un texte est une collection de mots séparé par des espaces. Nous avons donc un tableau de **Mot** (nous avons mis des pointeurs sur des **Mot**), et la méthode dessine fait appelle à la méthode dessine de chaque **Mot**, séparé par une translation.

Il y a en fait 2 méthodes dessine, une qui écrit le texte graphiquement justifié (comme pour le « text crawl » original de Star Wars), et une méthode qui écrit le texte graphiquement centré (ce que nous avons utilisé pour nos crédits)

On a procédé de la même façon que dans la classe **Mot** : on passe une String au constructeur, et on lie celle-ci à un tableau de **Mot**.

glSTAT :

Classe de type opengl, qui s'occupe de faire défiler les crédits. Ses attributs sont des **Text**, initialisé dans le constructeur, voir des variables sans noms pour certain ajouts. (Dommage qu'on n'ait pas eu le temps de trouver comment mettre de la musique). Un Timer s'occupe de « faire défiler » les crédits (concrètement à chaque pas de temps du chronomètre, on fait une translation du repère et on redessine.

FenetreSTAT :

Cette classe sert uniquement à accueillir **glSTAT**. Il y a en plus, de nouveau la gestion manuelle de la fermeture de la fenêtre pour éviter un « segmentation fault » dans le cas où le timer tourne toujours. **FenetreSTAT** est un attribut de **FenetreCTRL**.

Ci-dessous est la mise en page de la classe FenetreCRTL, avec le nom des attributs a leurs place, le nom des sizers, et leurs taille pour mieux permettre de comprendre le code de FenetreCTRL