C++ PoP – Sections Electricité et Microtechnique

Printemps 2023 : *Mission Propre-En-Ordre*

Combien de temps vous faut-il pour nettoyer la planète ?

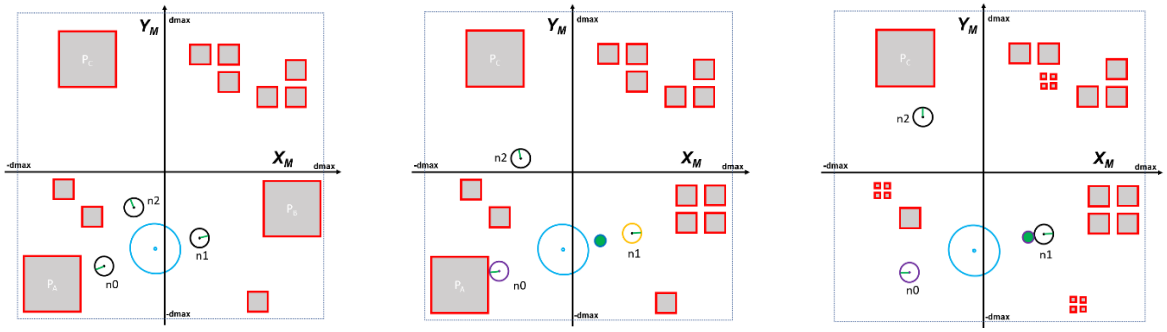


Fig 1 : de gauche à doite, le robot spatial bleu a pour mission la neutralisation des particules radioactives (bord rouge), ici avec 3 robots neutraliseurs n0, n1 et n2 et un réparateur (vert) en cas de panne (n1).

1. Introduction

Ce projet est indépendant de celui du semestre dernier. Le lien reste néanmoins la mise en œuvre des grands principes (*abstraction, ré-utilisation*), les conventions de présentation du code et les connaissances accumulées jusqu'à maintenant dans ce cours. Le but du projet est surtout de se familiariser avec deux autres grands principes, celui de *séparation des fonctionalités* (*separation of concerns*) et celui *d'encapsulation* qui deviennent nécessaires pour structurer un projet important en *modules* indépendants.

Nous mettrons l'accent sur le lien entre *module* et *structure de données*, et sur la robustesse des modules aux erreurs. Par ailleurs *l'ordre de complexité* des algorithmes sera testé avec des fichiers tests plus exigeants que les autres.

Vous pouvez faire plus que ce qui est demandé dans la donnée mais n'obtiendrez aucun bonus ; notre but est d'éviter que vous passiez plus de temps que nécessaire pour faire ce projet au détriment d'autres matières. Dans tous les cas, vous êtes obligé de faire ce qui est demandé *selon les indications de la donnée et des documents des rendus*. Vos éventuelles touches personnelles ne doivent pas interférer avec les présentes instructions.

Le projet étant réalisé par groupes de deux personnes, il comporte un oral final individuel noté pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de l'ensemble du projet. Une performance faible à l'oral peut conduire à un second oral approfondi et une possible baisse individuelle des notes des rendus.

La suite de la donnée indique les ***variables*** en ***italique gras*** et les **constantes** globales en **gras** (la valeur des constantes est visible dans les annexes de ce document ; des fichiers .h seront fournis. On utilisera la *double précision* pour des calculs en virgule flottante.

Pitch du sujet *: un robot spatial arrive sur une planète inhabitée mais qui a été exploitée dans le passé pour ses ressources. Sa mission est de neutraliser toutes les particules radioactives instables qui y sont restés. Pour cela, à partir de son lieu d'atterrissage (fixe) il peut envoyer des robots neutraliseurs non-holonomes qui doivent se positionner très précisément vis-à-vis des particules radioactives pour pouvoir les faire disparaitre. L'objectif est de décontaminer la planète le plus rapidement possible étant donné les risques présentés par les particules instables. En effet la matière radioactive peut changer d'état de manière aléatoire et supprimer la protection de tout robot neutraliseur au voisinage de celle-ci ; il ne peut alors plus bouger. Il faut dans ce cas envoyer un robot réparateur au plus vite pour le remettre en état de marche sinon il est détruit après un temps donné. La mission est finie lorsqu'il n'y a plus de particule radioactive à traiter et que tous les robots sont revenus à bord*

2. Modélisation des composantes de la Simulation

But : Le but de ce projet est de mettre au point une simulation responsable de la mise à jour de l'état de l'ensemble des particules et des robots. Cela est effectué sous la forme d'une boucle infinie de mise à jour de l'état de ses composantes. Chaque mise à jour correspond à l'écoulement d'***une*** *unité de temps* **delta_t**. Un compteur de mises à jour ***nbUpdate*** doit permettre de mesurer la durée de la simulation. Nous demandons de structurer la mise à jour selon les étapes suivantes (détails dans les sections suivantes):

```
// UNE mise à jour de la simulation
// entrée modifiée : ensemble des robots et des particules

   Incrémentation du compteur de mise à jour

   Désintégration éventuelle de chaque particule                  // section 2.1.4
      Y compris l'effet sur l'état des robots neutraliseur

   Destruction des robots neutraliseurs trop longtemps en panne   // section 2.1.4

   Prise de décision du robot spatial                             // section 2.5
      Décision de création d'un nouveau robot neutraliseur ou réparateur
      Décision de mise à jour du but des robots réparateurs
      Décision de mise à jour de la particule cible des robots neutraliseurs

   Pour chaque robot neutraliseur en service                      // section 2.1.3
      Gérer son état (panne, déplacement, neutralisation)
      Si nécessaire
         Mettre à jour l'ensemble des particules ou des neutraliseurs

   Pour chaque robot réparateur en service                        // section 2.1.2
      Gérer son état (déplacement et éventuel dépannage)
```

Ebauche de pseudocode 1 : ordre des opérations à effectuer pour chaque mise à jour de la simulation. La lecture de fichier et l'affichage graphique sont des tâches indépendantes de celle-ci (cf section 4 et 5).

La figure 1 montre le système de coordonnée ***Monde*** de la simulation avec **X_M comme axe horizontal, orienté positivement vers la droite**, **et Y_M comme axe vertical, orienté positivement vers le haut**. Le site contaminé est limité à un domaine **[-dmax, dmax]** ; il est dessiné à l'aide d'un carré indiquant sa frontière. Les particules doivent être entièrement comprise dans ce domaine. Le déplacement des robots est autorisé en dehors de cet espace. Les composantes de la simulation sont décrites dans les sections suivantes.

2.1 Les éléments de la simulation

Trois types de robots existent dans cette simulation ; ils sont tous modélisés par une forme circulaire.

2.1.1 Robot spatial

Le robot **Spatial**, de rayon **r_spatial** ; sa position (x,y) est fixe dans le repère Monde. Il n'a pas d'orientation. Il mémorise les informations définissant l'avancement de la simulation :

- Compteur de mises à jour (entier) ***nbUpdate*** qui est incrémenté d'une unité pour chaque mise à jour
- Nombres des robots neutraliseurs ***nbN*** = ***nbNr*** (en réserve) + ***nbNs*** (en service) + ***nbNd*** (détruits).
- Nombres des robots réparateurs : ***nbR*** = ***nbRr*** (en réserve) + ***nbRs*** (en service)

Les robots en service peuvent entrer et sortir du robot spatial dans n'importe quelle direction. Leur création s'effectue au centre du robot spatial. Pour le retour, il suffit que leur **centre** soit considéré comme en collision avec le robot spatial pour valider leur retour à l'intérieur du robot spatial.

Les algorithmes de prises de décision du robot spatial (cf Ebauche pseudocode) sont précisées en section 2.5.

La simulation stoppe lorsqu'il n'y a plus de particule radioactive à traiter et que *tous les robots sont revenus à bord du robot spatial*. La dernière valeur du compteur ***nbUpdate*** sera l'équivalent d'un score.

2.1.2 Robot réparateur

Un robot **Reparateur**, de rayon **r_reparateur**, est omni-directionnel, c'est-à-dire qu'il peut bouger dans n'importe quelle direction sans avoir besoin de contrôler une orientation. C'est pourquoi il est représenté par une position (x,y) et un but (xb, yb) décidé par le robot spatial.

Il se déplace vers ce but en translation avec la vitesse **vtran_max** jusqu'à atteindre le contact de son but. Une fois le contact établi, la réparation est considérée comme effectuée. Dès la mise à jour suivante, son but est d'aller vers une autre réparation ou de revenir au robot spatial s'il n'y a aucune réparation à effectuer. Lors de ses déplacements il est aussi soumis aux règles de gestion des collision (section 2.2) comme le robot Neutraliseur.

2.1.2.1 Déplacement du robot Réparateur pour une mise à jour sur delta_t

Comme indiqué plus haut ce type de robot se déplace en ligne droite vers son but (xb, yb) avec une vitesse **vtran_max**. Sa position à l'instant t étant (x,y), sa mise à jour pour un intervalle **delta_t** est obtenue de la manière suivante :

- Normaliser le vecteur reliant sa position à son but et le multiplier par **vtran_max.delta_t**. On obtient ainsi le vecteur de déplacement courant ***u***.
- La nouvelle position est l'addition vectorielle de la position courante et du vecteur ***u***.

2.1.3 Robot neutraliseur

Un robot **Neutraliseur**, de rayon **r_neutraliseur**, est non-holonome, c'est-à-dire qu'il bouge comme une chaise roulante (section 2.1.3.1). Son but est d'aller au contact d'une particule radioactive de telle manière que l'outil de décontamination soit orthogonal à la surface de la particule pour pouvoir la faire disparaitre (section 2.3).

Le robot tombe en panne en cas de changement d'état de la particule quand celui-ci est dans son voisinage (section 2.1.4). Il doit être réparé avant un nombre de **max_update** mises à jours sinon il est détruit. Il faut qu'un robot réparateur vienne à son contact pour qu'il puisse reprendre sa mission à la mise à jour suivante.

La figure 2a précise comment bouge un robot non-holonome et montre les deux degrés de mobilité dans la position par défaut : on peut seulement *avancer/reculer selon l'axe* X_r et *tourner autour de l'origine (point rouge)*. Comme pour une chaise roulante, il n'est pas possible de bouger latéralement selon Y_r . L'état courant d'un robot est représenté par la position (x,y) de son centre dans le repère Monde et par l'angle α entre l'axe X_r du robot et l'axe X du repère Monde comme illustré sur le dessin simplifié de la Fig 2c. L'angle α doit être mesuré en radians et être compris dans l'intervalle $[-\pi, \pi]$.

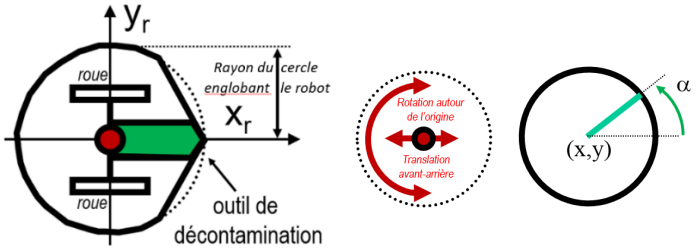


Fig 2 : Vues de dessus du robot ; (a) système de coordonnées (X_r, Y_r) du robot et orientation du robot par défaut ; on remarque les deux roues parallèles à l'axe X_r et l'outil de décontamination aussi orienté selon X_r, (b) les 2 degrés de mobilité possible (translation avant/arrière et rotation autour de l'origine) (c) Représentation simplifiée d'un robot montrant la position (x,y) de son centre et son orientation α.

2.1.3.1 Simplification du déplacement du robot Neutraliseur pour une mise à jour sur delta_t

Le déplacement du robot est contrôlé en définissant la vitesse selon les deux degrés de mobilité (Fig2b) :

- vitesse en translation ***vtran*** comprise dans l'intervalle [-**vtran_max, vtran_max**]
- vitesse en rotation ***vrot*** comprise dans l'intervalle [-**vrot_max, vrot_max**]

En théorie, lorsque ***vrot*** est différent de zéro, la combinaison des deux mouvements de translation et de rotation du robot correspond à un mouvement sur un arc de cercle de rayon **vtran/vrot**. Cependant, pour ce projet nous demandons d'adopter une mise à jour de la position et de l'orientation qui repose sur une approximation de la trajectoire par un petit segment de droite car l'intervalle de temps **delta_t** (Δt sur le dessin) est considéré comme suffisamment petit pour la justifier (Fig 3a) :

- **d'abord** une translation Δd selon l'axe X_r du robot: $\Delta d = vtran \cdot delta\_t$ **(Equ. 1)**
- puis une rotation $\Delta\alpha$ autour du centre du robot: $\Delta\alpha = vrot \cdot delta\_t$ **(Equ. 2)**

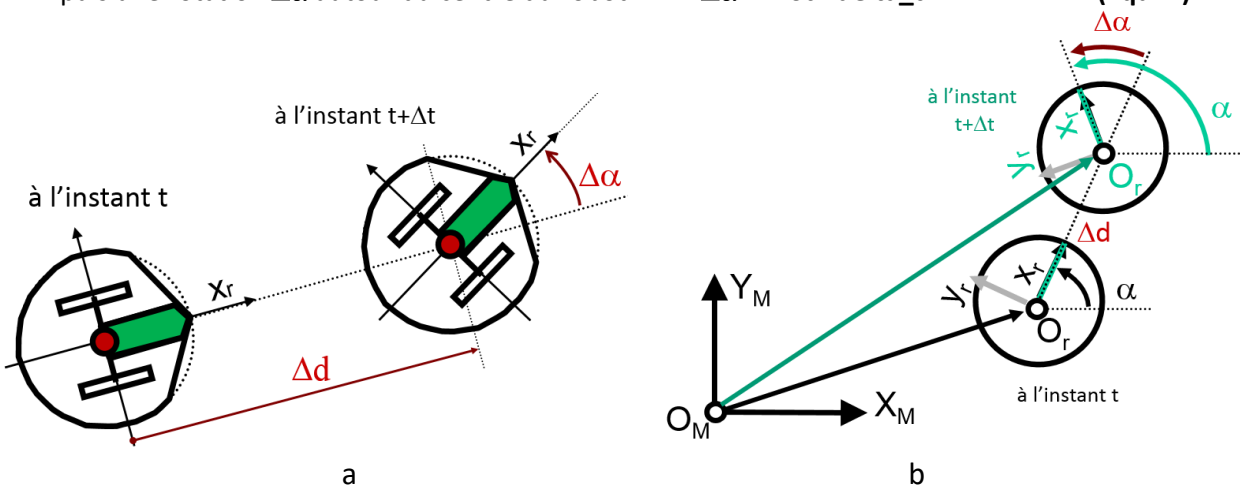


Fig 3 : (a) simplification du calcul du déplacement pour un petit **delta_t**, (b) mise à jour de la position du robot dans le repère Monde pour l'instant t + **delta_t** connaissant sa position et son orientation à l'instant t.

Pour trouver la nouvelle position et orientation du robot dans le repère Monde il suffit d'exploiter l'état courant du robot défini par sa position (x,y) correspondant au vecteur $\mathbf{O_M O_r}$ et son orientation α à l'instant t (Fig 3b) :

- Le vecteur de translation dans le repère Monde est donné par : $\boldsymbol{u} = \Delta d\ .(\cos\alpha, \sin\alpha)$
- La nouvelle position est l'addition vectorielle de la position courante et du vecteur de translation ***u***.
- La nouvelle orientation est donnée par $\alpha + \Delta\alpha$; celle-ci doit être renormalisée dans $[-\pi, \pi]$.

2.1.4 Particule contaminée, probabilité et fin de décomposition

La simulation contient ***nbP*** particules contaminées. Une Particule est représentée par un carré plein avec un contour de couleur distincte (Fig 1 & 4). Son coté a une longueur ***d_particule***. Les particules sont alignées sur les axes X et Y. Elles doivent être entièrement comprises dans le domaine [**-dmax, dmax**] selon X et Y.

Chaque particule à une probabilité constante de se décomposer en 4 particules dont les centres sont situés au quart et trois-quarts de la particule d'origine et dont le coté a pour longueur *: **d_particule**/2 - **2.epsil_zero***. La Figure 4a,b,c montre deux étapes de décomposition avec, à chaque fois, une soustraction de **2.epsil_zero** de la demi-longueur de la particule parente. Le but est d'éviter que les nouvelles particules soient considérées comme en collision.

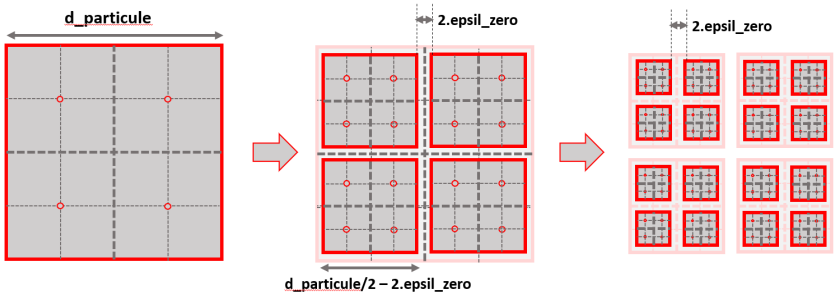


Fig 4 : (a,b,c) de gauche à droite, deux étapes de décomposition d'une particule. Les nouvelles particules ont leur centre au quart et trois quarts des cotés de leur particule parente. Leur longueur est la moitié de la particule parente de laquelle on soustrait **2.epsil_zero**

La décomposition d'une particule est liée à la probabilité **desintegration_rate**, c'est-à-dire qu'il n'est pas certain qu'elle se produise pendant l'intervalle **delta_t** d'une mise à jour (cf section 3.1 pour les aspects liés au langage C++).

Zone à risque et conséquence de la désintégration sur les robots : la zone à risque d'une particule est définie par le carré dont le coté est multiplié par **risk_factor** (Fig 6). Si une partie d'un robot neutraliseur se trouve dans cette zone pendant la mise à jour où se produit une désintégration, celui-ci tombe en panne et ne peut plus bouger ni tourner. Le robot mémorise alors ***nb_update*** dans sa variable ***k_update*** (sauf s'il est déjà en panne). Il doit être réparé avant un nombre de **max_update** mises à jour sinon il est détruit si :

$$nb\_update - k\_update \geq \text{max\_update} \quad \textbf{(Equ. 3)}$$

La réparation est obtenue par le contact d'un robot réparateur avec le robot en panne (c'est-à-dire une détection de collision); ce dernier pourra bouger à nouveau à la mise à jour suivante. Le robot spatial et le robot réparateur sont immunisés contre la désintégration.

Fin de la désintégration : La désintégration d'une particule ne continue pas si la future longueur de coté est plus petite que **d_particule_min + epsil_zero.**

2.2 Gestion des collisions entre éléments de la simulation

Les deux contextes de tests de collision sont :

- **Situation initiale (lecture de fichier)** : les collisions sont interdites entre les entités de la simulation dans leur configuration initiale (particule-particule, robot-robot, robot-particule). La collision est autorisée entre le robot spatial et les 2 autres types de robot. Le projet doit détecter les configurations initiales incorrectes.
- **Simulation :** toute détection de collision sur la *future* position calculée pour la mise à jour courante annule le déplacement en translation (la rotation est exécutée); seul le robot qui voulait se déplacer (pas l'autre élément de la collision) change de couleur pour nous informer de cet état pour la mise à jour courante.

Cependant la forme des robots est approchée par un cercle tandis que les particules sont carrées. Il faut donc gérer les collisions entre cercles, entre carrés et entre cercles et carrés. Nous utiliserons la même tolérance pour définir la collision entre ces différentes formes.

2.2.1 Détection de collision entre cercles

Nous posons qu'il y a ***collision*** entre deux cercles de centres **C1** et **C2** et de rayon respectifs **r1** et **r2** lorsque :

$$D < (r1 + r2) + epsil\_zero \quad \textbf{(Equ. 4)}$$ où **D** est la distance entre les centres **C1** et **C2**

2.2.2 Détection de collision entre carrés

Soit deux carrés de centres **C1(x1,y1)** et **C2(x2,y2)** et de longueurs de coté **d1** et **d2**. Le vecteur C1C2 reliant les deux centres a pour coordonnées (x2-x1, y2-y1). Nous posons qu'il y a collision si :

$$|x2-x1| < d1/2 + d2/2 + epsil\_zero \quad AND \quad |y2-y1| < d1/2 + d2/2 + epsil\_zero \quad \textbf{(Equ. 5)}$$

2.2.3 Détection de collision entre un cercle et un carré

Soit le carré de centre **C1(x1,y1)** de longueur de coté **d1** et le cercle de centre **C2(x2,y2)** de rayon **r2**. Le vecteur **C1C2** reliant les deux centres a pour coordonnées (x2-x1, y2-y1). Nous posons qu'il y a collision :

$$|x2-x1| < d1/2 + r2 + epsil\_zero \quad AND \quad |y2-y1| < d1/2 + r2 + epsil\_zero \quad \textbf{(Equ. 6)}$$

Sauf si :

$$(|x2-x1| > d1/2 \quad AND \quad |y2-y1| > d1/2) \; AND \quad (L > r2 + epsil\_zero) \quad \textbf{(Equ. 7)}$$

Avec **L** la norme du vecteur **(|x2-x1|-d1/2, |y2-y1| -d1/2)**

ATTENTION : quand ces tests sont effectués dans le contexte de lecture d'un fichier, la quantité **epsil_zero** doit être considérée comme nulle ; c'est nécessaire à cause des arrondis produits par la sauvegarde formatée.

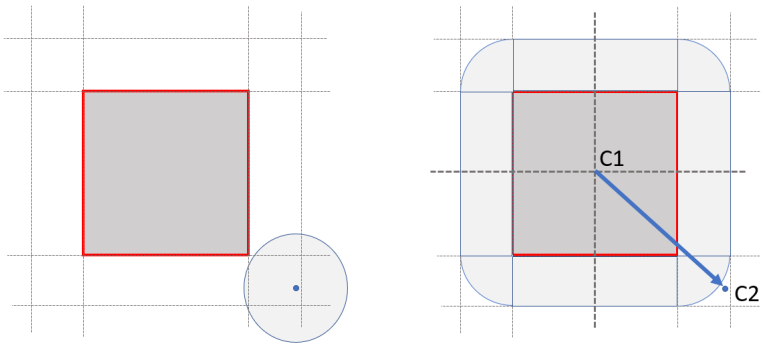


Fig 5 : on augmente le carré de centre C1 (x1,y1) et de coté d1 (à gauche) avec le rayon r2 du cercle de centre C2 (x2,y2) pour ramener la détection de collision à l'appartenance du point C2 à la nouvelle forme (à droite)

2.3 Conditions à remplir pour la décontamination

Il faut remplir deux conditions pour qu'un robot neutraliseur puisse faire disparaître une particule (Fig 6) :

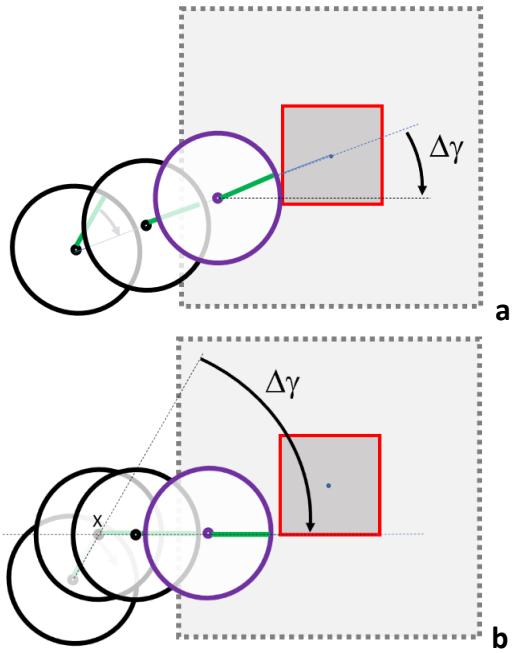
- Une collision a été détectée pendant la mise à jour courante (signalée avec une couleur différente pour le robot sur la Fig 6)
- **ensuite** il y a vérification de l'***alignement*** de l'outil de contamination avec la direction perpendiculaire à la surface de la particule. Nous posons qu'il y a ***alignement*** si la valeur absolue de l'angle $\Delta\gamma$ entre Xr et cette direction est telle que :

$$|\Delta\gamma| < \textbf{epsil\_alignement} \qquad \textbf{(Equ. 8)}$$

2.4 Coordination de la translation et de la rotation d'un robot non-holonome

La simulation devra comparer trois types de coordination de la translation et de la rotation d'un robot neutraliseur non-holonome. Pour cela le robot spatial devra affecter d'une manière systématique le type de coordination que devra suivre un robot neutraliseur.

Chaque nouveau robot neutraliseur créé par le robot spatial est associé à la valeur ***i_n*** qui vaut ***nbNs+nbNd***. Le type de coordination d'un robot est alors ***c_n = i_n* % 3 (Equ. 9)**. Ces types sont décrits ci-dessous. La valeur ***c_n*** devra être mémorisée dans le fichier de sauvegarde de manière à garantir l'exécution de la même coordination après la lecture du fichier sauvegardé (section 4).



2.4.1 Type 0 : viser la cible, avancer et s'aligner pendant la phase de collision (Fig 6a)

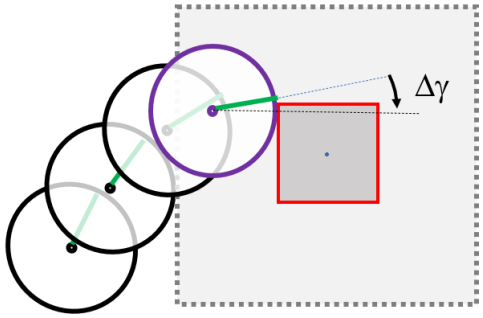
Tout d'abord le robot tourne sur place pour s'aligner sur le centre[1] de la particule cible puis avance en ligne droite jusqu'à la collision, et enfin, en gardant cet état de collision, il tourne sur place pour l'alignement.

Le risque avec cette approche est d'être présent dans la zone à risque pendant une durée plus importante qu'avec les autres types et donc de subir une panne.

2.4.2 Type 1 : viser un point en dehors de la zone à risque, s'aligner puis avancer (Fig 6b)

Il s'agit d'abord de tourner pour viser un point (à calculer) en dehors de la zone à risque, de l'atteindre, puis de s'aligner et ensuite seulement d'avancer en ligne droite jusqu'à la particule. Le temps de présence dans la zone à risque est réduit mais le temps de rotation reste éventuellement plus long que pour les 2 autres types.

[1] Si le robot se trouve dans un angle de la particule (Fig 5 droite), on demande de s'aligner sur le vecteur C1C2.



c

2.4.3 Type 2 : avancer tout en s'alignant vers la cible, finir pendant la phase de collision (Fig 6c)

Pour que cette approche fonctionne bien, le robot ne peut avancer que si $|\Delta\gamma| < \pi/3$. Sinon il tourne sur place pour s'aligner avec la direction la plus proche de la cible. Quand la condition ci-dessus est remplie, l'alignement continue tout en avançant en direction de la cible. Le rapport final précisera la relation liant ***vtran*** et $|\Delta\gamma|$ pour obtenir une approche efficace vers la cible.

Fig 6 : après mise à jour de la position d'un robot avec correction de collision, la seconde condition à remplir pour faire disparaitre la particule est que l'écart angulaire $|\Delta\gamma|$ soit plus petit que **epsil_alignement**.

2.5 algorithmes de prise de décision par le robot spatial

L'objectif des robots est d'atteindre l'ensemble des particules le plus vite possible pour les décontaminer avant qu'elles se désintègrent encore plus. Leur position initiale peut être quelconque (fournie dans un fichier). On suppose que le robot spatial a accès à l'ensemble des informations des autres robots et des particules pour prendre ses décisions.

2.5.1 Décision de création d'un nouveau robot

Tout d'abord la création d'un robot ne peut avoir lieu que lorsque le compteur de mises à jour est un multiple de **modulo_update**. Ensuite, il faut trouver un juste milieu entre d'une part, créer autant de nouveaux robots que possible (jusqu'au maximum de ***nbNr***) pour décontaminer rapidement les particules et d'autre part, éviter que les robots se ralentissent les uns les autres car ils sont soumis aux règles de collision (section 2.2). Pendant la simulation nous demandons qu'il y ait au moins un robot neutralisateur par type de coordination translation/rotation (section 2.4) et qu'il y ait au moins un robot réparateur dès qu'un neutraliseur est en panne. Une fois ces conditions remplies, vous êtes libres d'adapter le rythme de création selon votre propre stratégie.

2.5.2 Décision de mise à jour du but des robots réparateurs

On demande de mettre au point l'algorithme suivant à chaque mise à jour : pour chaque robot neutraliseur en panne, trouver le robot réparateur le plus proche et lui donner ce robot comme but seulement si cette distance la plus proche est plus petite que ***(nb_update – k_update)*vtran_max.** Si cette condition n'est pas remplie le robot neutraliseur ne peut pas être sauvé ; on va donc l'ignorer et passer au suivant. On exécute autant de fois que nécessaire cette boucle des neutraliseurs en panne jusqu'à ce que tous les réparateurs aient un but *qu'ils peuvent réparer (sinon ils n'ont pas de but à réparer).* Par construction, ces robots peuvent donc changer de but d'une mise à jour à la suivante. Vous pouvez choisir de renvoyer un réparateur vers le robot spatial ou de le laisser sur place s'il n'y a pas de neutraliseur en panne *ou s'ils ne sont pas réparables*. Il faudra seulement tous les y ramener à la fin de la décontamination.

2.5.3 Décision de mise à jour de la particule cible des robots neutralisateurs

On demande de mettre au point l'algorithme suivant à chaque mise à jour : pour chaque particule d'une liste triée dans l'ordre décroissant de leur taille, trouver le robot neutraliseur le plus proche et lui donner cette particule comme but. S'il y a moins de particules que de robots il est autorisé de ré-exécuter cette boucle pour envoyer plus d'un robot vers cette particule. Par construction, ces robots peuvent donc changer de but d'une mise à jour à la suivante. En cas de collision avec une autre particule sur le chemin de la particule cible, on demande de détruire d'abord la particule qui bloque le passage.

Remarque importante : la vitesse de rotation maximum étant relativement faible le critère de «robot le plus proche » doit aussi prendre en compte l'écart angulaire à corriger pour effectuer la décontamination. Pour simplifier et homogénéiser les quantités comparées, on évalue seulement le *temps* que prendrait l'alignement d'une *coordination de type 0*. Ce temps s'ajoute au temps de parcours en translation.

3 Actions à réaliser par le programme de simulation

Le but du programme est de pouvoir réaliser les actions suivantes :

- **Exécution de la simulation en continu (boucle infinie) ou ponctuelle (une seule mise à jour à la fois)**
  - Mise en œuvre du hasard pour la désintégration
- **Lecture** d'un fichier pour initialiser l'état du monde (section 4).
- **Ecriture** d'un fichier décrivant l'état actuel du monde (section 4)

Après **chaque action de lecture** et **chaque mise à jour de la simulation**, l'affichage de l'état courant de la simulation doit être effectué dans une interface dédiée (section 5) et dans une fenêtre graphique (section 6). La section 7 précise l'architecture modulaire du projet et comment la simulation et l'affichage sont gérés à l'aide de la programmation par événements. La section 8 précise la répartition des tâches entre les 3 rendus pour structurer votre travail.

3.1 Mise en œuvre avec C++

On exige l'usage de la fonction **atan2** de **<cmath>** pour obtenir une orientation à partir d'un vecteur (x,y).

3.1.1 Génération d'un booléen true avec une probabilité p

Une tâche doit faire appel à une probabilité pour son exécution lors d'une mise à jour:

- Booléen de désintégration d'une particule avec la probabibilité de **desintegration_rate**.

Avec C++11, il faut créer un objet de type **bernoulli_distribution** comme suit :

```
default_random_engine  e;       //à recréer à chaque lecture de fichier
double p(desintegration_rate); // probabilité de la section 2.1.4, Annexe A
… à chaque nouvelle mise à jour on divise p par le nb de particules :
bernoulli_distribution b(p/nbP); //booléen true avec probabilité p/nbP
… puis pour chaque particule:
      if(b(e)) // désintégration de la particule…
```

4. Sauvegarde et lecture de fichiers tests : format du fichier

Votre programme doit être capable d'initialiser l'état de la simulation à partir d'un fichier texte. Il doit aussi pouvoir mémoriser la configuration actuelle dans un fichier texte également. Cela vous permettra de pouvoir créer vos propres scénarios de tests avec un éditeur de texte comme geany.

4.1 Caractéristiques des fichiers tests

L'opération de lecture doit être indépendante des aspects suivants qui peuvent être différents d'un fichier à l'autre : présence de lignes vides commençant par **\n** ou **\r**, les commentaires commençant par **#** précédé éventuellement d'espaces, et les espaces avant ou après les données. Les indentations visibles dans le format ci-dessous ne sont pas obligatoires non plus. Les fins de lignes peuvent contenir **\n** et/ou **\r** à cause du système d'exploitation sur lequel le fichier a été créé ; votre programme doit pouvoir traiter ces cas (cf série fichier).

format général du fichier

```
# Nom du scenario de test
#
# nombre de particules puis les données d'une particule par ligne
nbP
    x1 y1 d1

# données du robot spatial
x y nbUpdate nbNr nbNs nbNd nbRr nbRs

# données des nbRs robots réparateurs en service (un par ligne)
    x1 y1

# données des nbNs robots neutraliseurs en service (un par ligne)
    x1 y1 a1 c_n panne k_update_panne
```

Dans ce format **x** et **y** désignent les coordonnées de la position d'une entité ; **a** désigne l'orientation en rd dans l'intervalle [-π, π]. Le fichier contient d'abord les données des particules (**d** est la longueur du coté) puis du robot spatial suivi par les robots réparateurs les robots neutraliseurs. L'entier **c_n** mémorise le type de coordination. Le booleen **panne** est vrai si le robot est en panne et **k_update_panne** est la valeur de **nbUpdate** quand le robot est tombé en panne (s'il n'est pas en panne, une valeur est quand même présente dans le fichier et doit être lue). Des exemples de fichiers sont fournis.

4.2 Vérifications à effectuer pendant la lecture et conséquences d'une détection d'erreur

La **lecture** doit vérifier que le coté des particules est supérieur ou égal à **d_particule_min**, ainsi que les conditions d'inclusion complète des particules dans le domaine **[-dmax, dmax]** et de non-collision détaillées au début de la section 2.2 (pour les tests effectués à la lecture, **epsil_zero** est ignoré). Elle doit aussi vérifier que toutes les valeurs **k_update_panne** sont inférieures ou égales à la valeur **nbUpdate** lue dans le fichier. Si plus de données que nécessaire sont fournies sur la ligne de fichier elles sont simplement ignorées sans générer d'erreur de lecture. Un commentaire peut aussi suivre les données et ne doit pas poser de problèmes.

Les conséquences d'une détection d'erreur dépendent du rendu du projet (section 8):

- Rendu1 : Dès la première erreur détectés à la lecture, il faut afficher dans le terminal le message d'erreur fourni avec le module **message** ou **error_shape** (section 8) et quitter le programme.
- Rendus 2 et 3: Il ne faut pas quitter le programme en cas de détection d'erreur. Dès la première erreur détectée à la lecture, il faut afficher dans le terminal le message d'erreur fourni (section 8), interrompre la lecture, détruire la structure de donnée en cours de construction et attendre une nouvelle commande en provenance de l'interface graphique.

5. Interface utilisateur (GUI)

Nous utilisons une seule fenêtre graphique divisée en 2 parties:

- Les boutons des actions ou affichage de données (Fig 7a partie gauche, et Fig 7b)
- le dessin de l'état courant de la simulation dans un *canvas* (Fig 1 et Fig 7a partie droite). Il s'agit du dessin du monde dans **[-dmax, dmax]**.

L'interface utilisateur doit contenir (Fig 7b):

Commandes générales :

- **Exit :** quitte le programme de simulation
- **Open** : remplace la simulation par le contenu du fichier dont le nom est fourni. Les structures de données antérieures doivent être supprimées ; on obtient donc un écran blanc si une erreur est détectée à la lecture.
- **Save** : mémorise l'état actuel de la simulation dans le fichier dont le nom est fourni.

Simulation :

- **Start** : bouton pour commencer/stopper la simulation en continu
- **Step** (lorsque la simulation est stoppée) : calcule seulement un pas de mise à jour

Affichage de données générales :

- ***nombre de mises à jour***
- ***nombre de particules***
- ***nombre de Robots réparateurs en service***
- ***nombre de Robots réparateurs en réserve***
- ***nombre de robots neutraliseurs en service***
- ***nombre de robots neutraliseurs en panne***
- ***nombre de robots neutraliseurs détruits***
- ***nombre de robots neutraliseurs en réserve***

a

b

Fig 7 : GUI

6. Affichage et interaction dans la fenêtre graphique

A partir du rendu 2, l'exécution du programme ouvre une fenêtre GTKmm contenant l'interface graphique utilisateur (Fig 7) et le dessin de la simulation dans un *canvas*. Le dessin du monde complet doit couvrir l'espace **[-dmax,dmax]** selon X et Y (Fig 1). On demande de matérialiser le domaine par une fine bordure.

Taille de la fenêtre d'affichage en pixels : La taille initiale du *canvas* dédié au dessin du monde est de **taille_dessin** en largeur et en hauteur (annexe C). La taille de la fenêtre peut changer durant l'exécution du programme. Un changement de taille de fenêtre ne doit pas introduire de distorsion dans le dessin (un carré reste un carré quelle que soit la taille et la proportion de la fenêtre).

Formes et couleurs : Le module graphique de bas niveau (**graphic**) met à disposition une table de couleurs prédéfinie dont les index peuvent être indiqué en paramètre des fonctions de dessin (section 8). Les entités suivantes devront utiliser les formes et couleurs suivantes :

- On travaille en light mode = le fond du dessin est blanc.
- La bordure du carré 2D du monde est grise
- Une particule est un carré gris avec un bord rouge (Fig 1 et 4)**.**
- Le robot spatial a un bord bleu clair, le robot neutraliseur a un bord noir sauf quand il est en collision (violet) ou quand il est en panne (orange) ; son orientation est indiquée par un rayon vert. Le robot réparateur a un bord noir et est rempli de vert.

6.1 Interaction avec le clavier

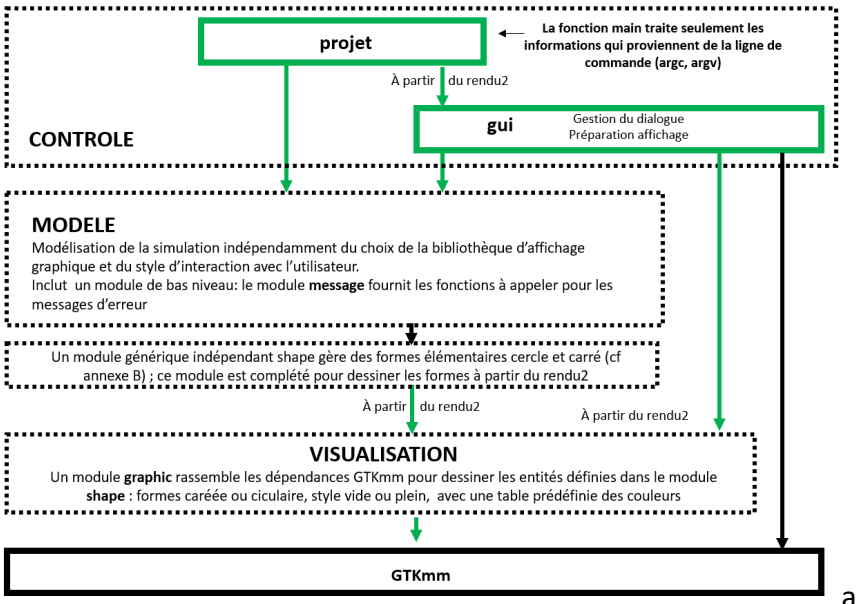
Utiliser la touche clavier '**s**' pour faire la même action que le bouton Start/Stop
Utiliser la touche clavier '**1**' pour faire la même action que le bouton Step

7. Architecture logicielle

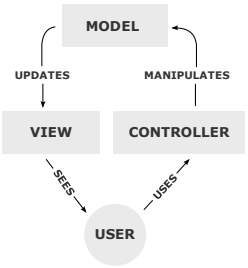
7.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 8 décrit l'organisation minimum du projet en sous-systèmes avec leur responsabilité (Principe de Séparation des Fonctionnalités) :

- **Sous-système de Contrôle : son but est de gérer le dialogue avec l'utilisateur (Fig 8b).** Si une action de l'utilisateur impose un changement de l'état de la simulation, ce sous-système doit appeler une fonction du **sous-système du Modèle** qui est le seul responsable de gérer les structures de données de la simulation (voir point suivant).
  Le sous-système de contrôle est mis en œuvre avec deux modules :
  - Le module **projet** qui contient la fonction **main** : son rôle est modeste car il est seulement responsable de traiter les éventuels arguments fournis sur la ligne de commande au lancement du programme. Pour le rendu1, le sous-système de **Contrôle** ne contient que le module **projet**.
  - le module **gui** est créé à partir du rendu2 pour gérer le dialogue utilisateur à l'aide de l'interface graphique mise en œuvre avec GTKmm.
- **Sous-système du Modèle**: est responsable de gérer les structures de données de la simulation. Il est mis en œuvre sur plusieurs niveaux d'abstractions selon les Principes d'Abstraction et de Ré-utilisation (section 7.2).
- **Utilitaire générique indépendant du Modèle :** un module **shape** gère des formes élémentaires circulaires ou carrées (cf annexe B) ; c'est l'équivalent d'une bibliothèque mathématique.
- **Sous-Système de Visualisation :** le module **graphic** dessine l'état courant de la simulation à l'aide des entités élémentaires gérées par le module **shape**. Le module **graphic** rassemble les dépendances vis-à-vis de GTKmm pour faire les dessins. On autorise l'inclusion de son interface dans l'interface **shape.h** pour que le Modèle puisse choisir le style (plein,vide) et les couleur prédéfinies dans **graphic.h** .



a



b : Diagramme conceptuel de l'approche Model-View-Controller [wikipedia]

Fig 8 : Architecture logicielle minimale à respecter (a), inspirée par l'approche MVC (b)

7.2 Décomposition du sous-système MODELE en plusieurs modules

Cette partie du présent document fait partie de l'étape *d'Analyse* dans la mise au point d'un projet. En bref, le Modèle gère la simulation ; ce Modèle est organisé en plusieurs modules pour maîtriser la complexité du problème et faciliter sa mise au point. La Figure 9 présente l'organisation minimale à adopter en termes d'organisation des modules :

- ***Au plus haut niveau***, le module **simulation** gère le déroulement de la simulation et les autres actions (lecture, écriture de fichier). Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi, en vertu du principe d'abstraction le module **simulation** *est le seul module* dont on peut appeler des fonctions en dehors du MODELE (Fig 9)[2].

- ***Niveau intermédiaire***: il faut au moins considérer un module distinct pour les particules et les robots Le module **robot** doit être conçu comme une hiérarchie de classes pour intégrer les 3 sortes de robots (nous accepterons une version simplifiée, sans hiérarchie de classe, pour le **rendu1** seulement).

- ***Au plus bas niveau*** **:** nous mettons à disposition un ensemble de fonctions dans **message.h** . Ces fonctions doivent être appelée pour faire afficher les messages d'erreurs liées au Modèle et détectées à la lecture d'un fichier. Une fonction supplémentaire est fournie pour afficher un message quand la lecture est effectuée avec succès. Il n'est pas autorisé de modifier le code source de ce module car il sera utilisé par notre autograder.

7.3 Module générique indépendant shape pour les calculs d'inclusion ou de collision

7.3.1 Indépendance de shape

Ce module est équivalent à une bibliothèque mathématique destinées à être utilisées par de nombreux autres modules de plus haut niveau (principe de Ré-utilisation). L'idée fondamentale est qu'il doit aussi être conçu pour être utilisable par d'autres programmes très différents de notre projet. Pour cette raison **AUCUN** des noms de types/concepts du niveau supérieur MODELE ne doit apparaître dans **shape.** On y trouvera des fonctions effectuant les tests d'inclusion et de collision utiles pour les niveaux supérieurs (cf Rendu1).

7.3.2 Relation « Possède-un » entre la hiérarchie de classes de Robot et les types de shape

Les classes du MODELE devront utiliser les types mis à disposition dans l'interface de **shape** MAIS SEULEMENT en utilisant la relation « ***possède-un*** » plutôt que la relation « ***est-un*** ». C'est-à-dire que, par exemple, un Robot

[2] si le sous-système de Contrôle veut modifier l'état de la simulation cela doit se faire par un appel d'une fonction de **simulation.h.** Par exemple la lecture du fichier doit se faire en appelant une fonction disponible dans simulation.h.

possède un attribut Cercle pour ses calculs géométriques et l'affichage mais la classe Robot n'est pas une classe dérivée de la classe Cercle car, en vertu du principe de *séparation des fonctionnalités* le module **shape** n'a pas vocation à servir de classe parente pour l'ensemble des applications qui utilisent ce module.

7.3.3 Type concret S2d et autres types de shape

Nous demandons d'implémenter le type **S2d** qui permet de modéliser une position et/ou un vecteur soit avec cette approche :

```
struct S2d {double x=0.; double y=0.;}; //plus robuste aux bugs
```

ou avec cette approche :

```
typedef array<double,2> S2d;
enum {X,Y}// quand on veut accéder explicitement à une coordonnée
```

Pour les 2 entités de cercle/carré, nous considérons qu'ils sont suffisamment simples et de bas-niveau pour vous autoriser à les créer à l'aide de **struct**. Alternativement, vous pouvez aussi les dériver d'une classe parente possédant un attribut **S2d** pour représenter la position d'un point.

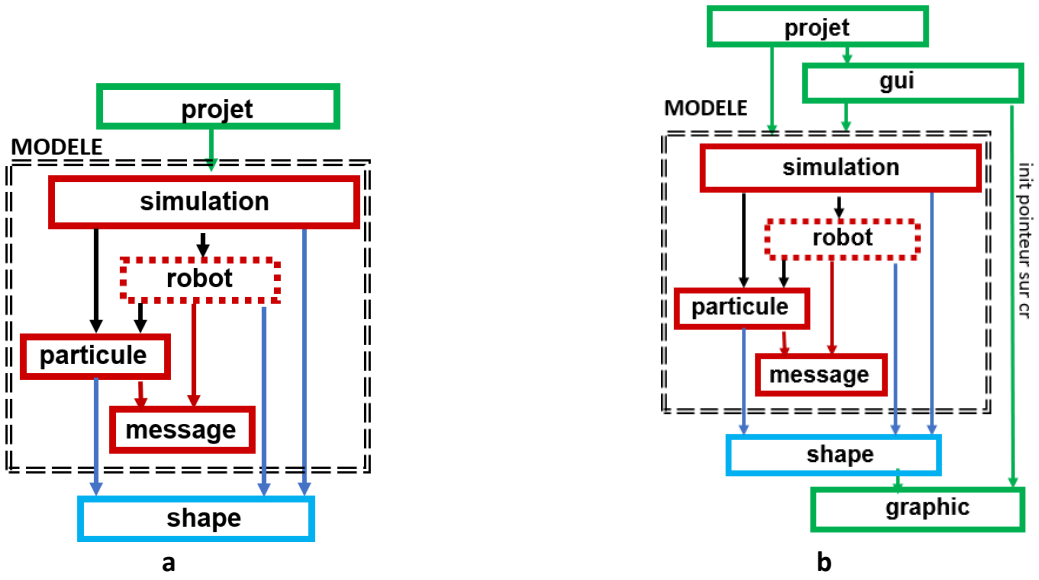


Figure 9 : (a) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (rendu1); (b) modules et dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendus 2 et 3)

7.4 Module graphique de bas-niveau (graphic)

A partir du rendu2 le module **shape** va aussi offrir des fonctions de dessin pour les types qui y sont définis dont **S2d**. Cependant le module **shape** doit rester indépendant d'une libraire graphique particulière (principe de regroupement des dépendances). C'est pourquoi les dépendances vis-à-vis de la bibliothèque **GTKmm** doivent être rassemblées dans le module **graphic**. C'est dans ce module qu'on définit une table de couleurs prédéfinies et les fonctions de tracé des formes géométriques ; son interface **graphic.h** met à disposition des symboles pour définir le style du dessin (plein/vide) et une table de couleurs prédéfinies. Comme indiqué en section 6, on autorise **shape.h** d'inclure **graphic.h** pour avoir accès à ces symboles de style/couleur dans le MODELE.

8. Syntaxe d'appel et répartition du travail en 3 rendus notés

Chaque rendu sera précisément détaillé dans un document indépendant. Votre exécutable doit s'appeler **projet**. Selon le rendu le programme doit pouvoir traiter un argument optionnel sur la ligne de commande.

8.1 Rendu1 : Son architecture est précisée par la Fig 9a.
Ce rendu sera toujours testé en indiquant un nom de fichier de test sur la ligne de commande selon la syntaxe suivante **: ./projet test1.txt**

Le programme cherche à initialiser l'état de la simulation en construisant une première version de vos structures de données. Le programme s'arrête dès la première erreur trouvée dans le fichier. Il sera possible de faire évoluer votre choix de structure de données entre le rendu1 et les suivants.

Le programme s'arrête aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, il y a affichage d'un message indiquant le succès de la lecture.

Le rendu1 ne doit PAS utiliser GTKmm.

8.2 Rendu2 : Son architecture est précisée par la Fig 9b.
Ce rendu avec GTKmm sera toujours testé comme pour le rendu1, en indiquant un nom de fichier de test sur la ligne de commande selon la syntaxe suivante : **`./projet test1.txt`**

Ce rendu construit les structures de données et affiche l'état initial avec GTKmm (section 6). Ce rendu sera testé en effectuant plusieurs lecture/écriture/relecture avec le GUI pour vérifier que l'affichage est bien correct, que le programme gère bien les erreurs détectées à la lecture et qu'il ré-initialise correctement les structures de données à chaque lecture de fichier. En effet, il est demandé de détruire les structures de données existantes avant de commencer toute lecture.

La simulation devra gérer seulement la désintégration des particules au cours du temps pour le rendu2.

Un rapport devra décrire les choix de structures de donnée en *anticipant* comment elles seront utilisées pour les algorithmes du rendu final (section 2.5).

8.3 Rendu3 : Ce rendu utilise toujours GTKmm (avec l'architecture de la Fig 9b). Si un nom de fichier est indiqué sur la ligne de commande il doit être ouvert pour initialiser l'interface graphique et le dessin, incluant l'affichage de la valeur initiale de l'état de la planète. Si aucun nom n'est fourni le programme initialise l'interface graphique et attend qu'on l'utilise pour demander l'ouverture d'un fichier.

Plusieurs scénarios de simulation seront testés pour illustrer les règles définies dans le présent document.

Un rapport final devra décrire votre approche pour les algorithmes de la section 2.5.

ANNEXE A : constantes globales du Modèle définies dans constantes.h

Ces constantes sont appelées « globales » car elles pourraient être nécessaires dans plus d'un module du Modèle. L'utilisation de **constexpr** crée automatiquement une instance *locale* dans chaque fichier où constantes.h est inclus ; il n'y a donc pas de problème de définition multiple de ces entités.

Ces constantes sont associées au Modèle ; elle reflète la nature du problème spécifique résolu dans le sous-système du Modèle. Pour cette raison *il n'est pas autorisé d'inclure ce fichier de constantes dans le module utilitaire* qui ne doit rester très général/générique et donc n'avoir aucune dépendance vis-à-vis de concepts et de constantes de plus haut niveau. Si vous désirez mettre en œuvre vos propres constantes, les bonnes pratiques sont les suivantes :
- utilisez **constexpr** pour les définir
- définissez-les *le plus localement possible* ; inutile de les mettre dans l'interface d'un module (.h) si elles ne sont utilisées que dans son implémentation (.cc)

Selon nos conventions de programmation, un nom de constante définie avec **constexpr** suit la même règle qu'un nom de variable (E12). Le texte de la donnée les fait apparaître en **gras** dans le texte.

```
#include "shape.h "       // nécessaire pour utiliser epsil_zero et disposer des symbols de graphic.h
enum Etat_neutraliseur {EN_PANNE, EN_MARCHE} ;

constexpr short unsigned maxF(25) ;

constexpr double dmax(128.)
constexpr double delta_t(0.125)      // seconde
constexpr double r_spatial(16.)
constexpr double r_reparateur(2.)
constexpr double r_neutraliseur(4.)
constexpr double vtran_max(4.)              // par seconde
constexpr double vrot_max(0.125)            // rd/s          env. 7°/s
constexpr double epsil_alignement(0.01)     // rd            env. 0.6°

constexpr double desintegration_rate(0.0002) ;
constexpr double risk_factor(3.) ;
constexpr double d_particule_min(8*shape ::epsil_zero) ;

constexpr unsigned max_update(600) ;
constexpr unsigned modulo_update(100) ;
```

ANNEXE B : constantes destinées générique définie dans shape.h

```
constexpr double epsil_zero(0.125) ;
```

ANNEXE C : constante destinée au sous-système de Contrôle

```
constexpr unsigned taille_dessin(500);                    en pixels
```