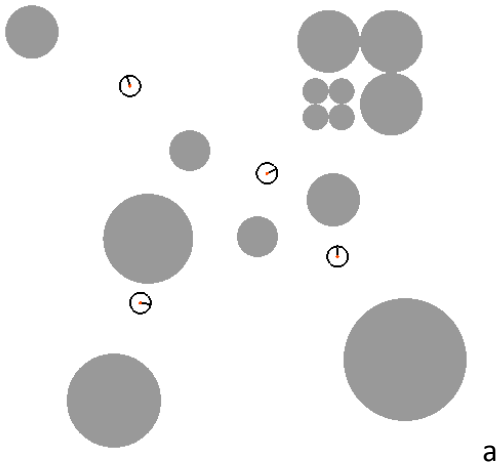
V 1.0

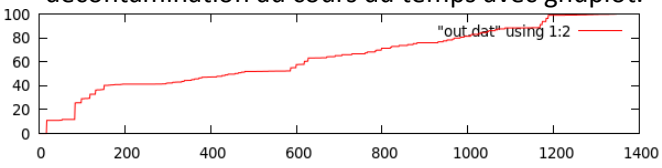
Projet Informatique – Sections Electricité et Microtechnique

Printemps 2018 : *Decontaminators* © R. Boulic

Vos robots pourront-ils décontaminer efficacement le site irradié ?



a



b

Figure 1 : (a) exemple d'un état de la simulation dans un système de coordonnées ***Monde*** montrant le site contaminé dans l'espace [-DMAX, DMAX] selon X et Y. Exemple d'entités manipulées : **robot** (représentation simplifiée avec un cercle épais vide avec un rayon indiquant l'outil de décontamination selon l'axe X_r), **Particules** (petits disques pleins ; leur taille dépend de leur état de décomposition). (b) visualisation de l'évolution du pourcentage de décontamination au cours du temps avec gnuplot.

1. Introduction

Le but de ce projet est de réaliser une simulation de l'évolution d'un ensemble de ***robots*** non-holonomes, c'est-à-dire qui bougent comme une chaise roulante, pour atteindre le plus rapidement possible des ***particules*** contaminées et les éliminer. Les particules présentent une propriété de décomposition qui transforme une particule en quatre particules indépendantes après un temps aléatoire. La simulation s'effectue dans un espace continu 2D avec une discrétisation temporelle DELTA_T de la mise à jour. Vous êtes responsables de mettre en œuvre une stratégie de coordination et de déplacement de vos robots en respectant les règles concernant les collisions et d'élimination des particules contaminées. L'évolution du taux de décontamination est enregistrée pour analyser l'impact de votre approche.

Il n'y a aucun lien vis-à-vis du précédent projet excepté 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 un autre grand principe, celui de *séparation des fonctionalités* (*separation of concerns*) qui devient nécessaire 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 (notion de type *opaque* et de *contrat*). Par ailleurs *l'ordre de complexité* des algorithmes sera testé avec des fichiers tests plus exigeants que les autres. L'évaluation du travail portera essentiellement sur la résolution du problème du point de vue informatique : structuration des données, modularité du programme, robustesse et ré-utilisabilité des modules, stratégie de test, ordre de complexité calcul/mémoire des algorithmes mis en oeuvre, compromis performance/occupation mémoire.

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 des notes des rendus.

La suite de la donnée utilise un certain nombre de constantes (en MAJUSCULE) disponibles dans les fichiers **constantes.h** et **tolerance.h**. Ces fichiers sont fournis en Annexe A. On utilisera la *double précision* pour les données et les calculs en virgule flottante.

2. Système à simuler

La simulation est effectuée dans un système de coordonnée ***Monde*** d'origine (0,0) ; **X est l'axe horizontal, orienté positivement vers la droite**, **et Y est l'axe vertical, orienté positivement vers le haut**. Le site contaminé est limitée à un domaine **[-DMAX, DMAX]** ; il est dessiné à l'aide d'un carré indiquant sa frontière. La position (x,y) du centre des particules doit être comprise dans ce domaine. Le déplacement des robots est autorisé en dehors de cet espace.

2.1 Les éléments de la simulation

2.1.1 Robot non-holonome : position, orientation et degrés de mobilité

La figure 2a précise le type de robot mobile que nous utilisons dans ce projet ; cela correspond à un robot aspirateur de type roomba par exemple. La Fig 2b 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*. Comme pour une chaise roulante, il n'est pas possible de bouger latéralement selon Y_r mais on peut atteindre n'importe quel point du plan 2D en combinant les deux degrés de mobilité disponibles. 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.

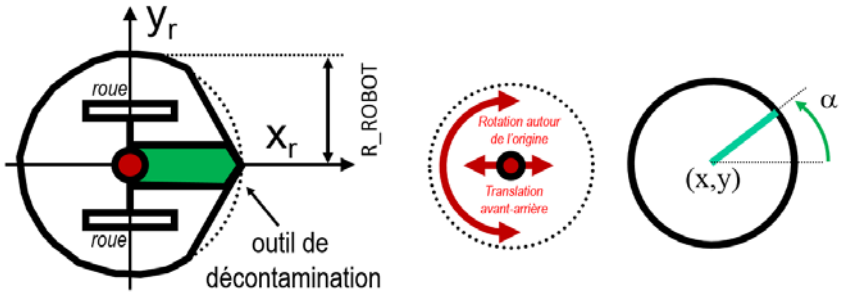


Fig 2 : Vues de dessus du robot ; (a) système de coordonnées (X_r, Y_r) du robot et orientation du robot par défaut ; on remarque les deux roues parallèles à l'axe X_r et l'outil de décontamination aussi orienté selon X_r, (b) les 2 degrés de mobilité possible (translation avant/arrière et rotation autour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

2.1.2 Robot non-holonome : simplification du déplacement

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]

Il est autorisé de combiner simultanément les deux mouvements de translation et de rotation du robot : cela correspond à un mouvement sur un arc de cercle.

L'intervalle de temps **DELTA_T** (Δt sur le dessin) d'une mise à jour étant considéré comme petit, cela permet de simplifier le calcul de la nouvelle position et orientation du robot selon les formules ci-dessous et les dessins de la Fig 3a :

- translation Δd selon l'axe X_r du robot: $\Delta d = vtran \cdot DELTA\_T$ **(Equ 1)**
- rotation $\Delta\alpha$ autour du centre du robot: $\Delta\alpha = vrot \cdot DELTA\_T$ **(Equ 2)**

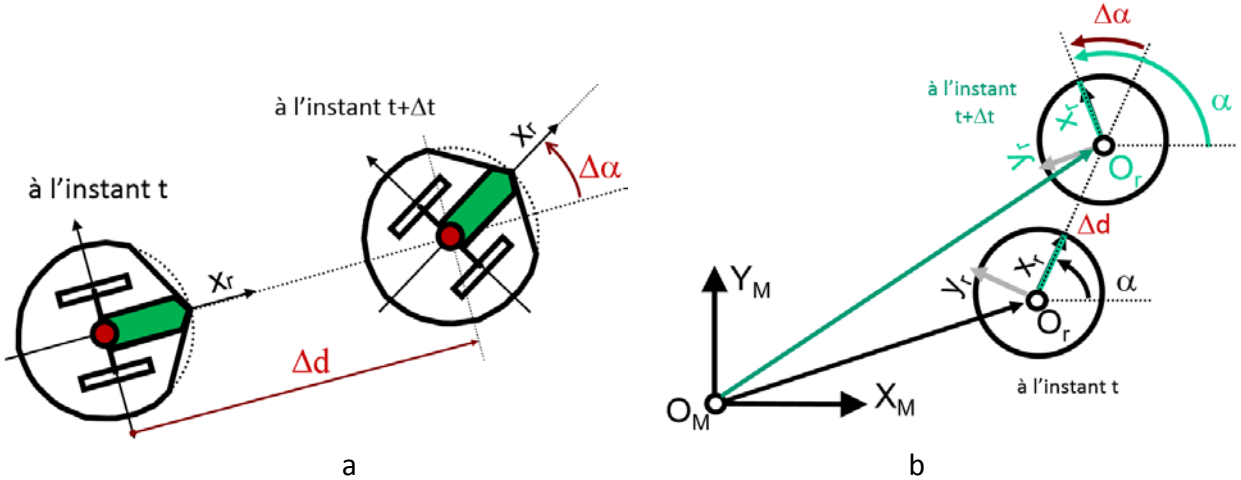


Fig 3 : (a) simplification du calcul du déplacement pour un petit DELTA_T, (b) mise à jour de la position du robot dans le repère Monde pour l'instant t + DELTA_T connaissant sa positon et son orientation à l'instant t.

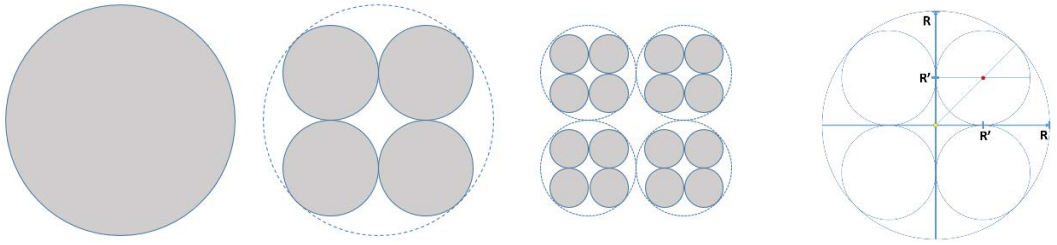
Pour trouver la nouvelle position et orientation du robot dans le repère Monde il suffit d'exploiter l'état courant du robot défini par sa position (x,y) correspondant au vecteur $\mathbf{O_M O_r}$ et son orientation α à l'instant t (Fig 3b) :

- Le vecteur de translation dans le repère Monde est donné par : $\Delta d .(\cos\alpha, \sin\alpha)$
- La nouvelle position est l'addition vectorielle de la position courante et du vecteur de translation
- La nouvelle orientation est donnée par $\alpha + \Delta\alpha$

2.1.3 Particule contaminée, probabilité et fin de décomposition

Une Particule contaminée est représenté par un disque plein (Fig 1). Elle est initialisée avec un rayon et une certaine énergie (la valeur de l'énergie n'a pas de lien avec le rayon).

Chaque particule à une probabilité constante de se décomposer en 4 particules. La Figure 4a,b,c montre deux étapes de décomposition. La Figure 4d montre la position relative et le rayon des nouvelles particules par rapport à la particule qui est décomposée.



a b c d

Fig 4 : (a,b,c) de gauche à droite, deux étapes de décomposition d'une particule. L'énergie d'une nouvelle particule est celle de la particule initiale multipliée par **E_PARTICULE_FACTOR**. (d) Le rayon R' d'une nouvelle particule est celui de la particule initiale multiplié par **R_PARTICULE_FACTOR** pour obtenir des particules tangentes.

La décomposition d'une particule est liée à la probabilité **DECOMPOSITION_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. L'encadré ci-dessous indique comment l'implémenter en langage C.

> La fonction **rand()**renvoie un nombre aléatoire (entier positif) entre **0** et **RAND_MAX** (définie dans stdlib.h). Tous les entiers de cet intervalle ont la même chance d'être renvoyés par **rand()**. Soit la probabilité ***p*** (comprise entre 0 et 1) d'une action donnée. Pour savoir si cette action a lieu pour la mise à jour courante, il suffit de normaliser l'entier renvoyé par **rand()** et de comparer avec ***p*** :
>
> Si **rand()/RAND_MAX ≤ p** Alors l'action a lieu pour cette mise à jour **(Equ 3)**

Fin de la décomposition : La décomposition ne continue pas si le nouveau rayon qui serait obtenu est strictement plus petit que **R_PARTICULE_MIN**

2.2 Gestion des collisions entre éléments de la simulation

Rappelons que les particules et les robots sont représentés par des cercles pour gérer le traitement des collisions. 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 **(Equ. 4)** où **D** est la distance entre les centres **C1** et **C2**

Le test de collision est utilisé comme suit :

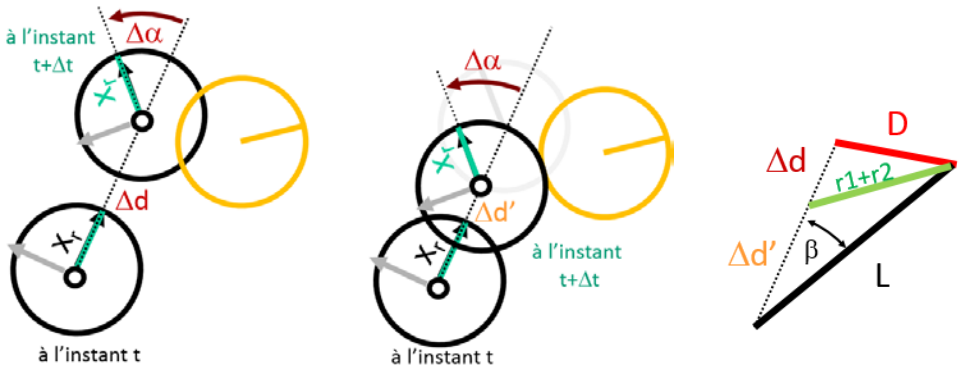
- La collision est interdite entre les entités de la simulation dans leur configuration initiale (particule-particule, robot-robot, robot-particule): le projet doit détecter les configurations initiales incorrectes
- La collision doit être corrigée si elle se produit entre deux robots ou entre un robot et une particule: le robot qui vient de se déplacer et de créer la collision (Fig 5a) doit ***reculer le long de la ligne qui a***

permis son déplacement de façon à être tangent tel que ***D = r1 + r2*** (Fig 5b et 5c). S'il y a collision avec plus d'une autre entité (robot ou particule) on applique cette correction autant de fois que nécessaire en commençant par traiter les collisions entre robots. La rotation n'est pas altérée.

On note **L** la distance entre les centres des 2 entités avant le déplacement du robot et **D** la distance entre les centres qui cause la collision à cause du déplacement Δd donné par l'équation 1. On peut trouver la distance qui produit le cas tangent $\Delta d'$ en utilisant la loi des cosinus (Fig 5c). On s'en sert d'abord pour trouver le cosinus de l'angle β et dans un second temps pour trouver $\Delta d'$ en résolvant une équation du second degré et en gardant la valeur de la racine la plus petite :

- $\cos\beta = (\Delta d^2 + L^2 - D^2)/(2.\Delta d.L)$
- $(r1 + r2)^2 = \Delta d'^2 + L^2 - 2.\Delta d'.L.\cos\beta$

d'où $\Delta d'^2 - 2.\Delta d'.L.\cos\beta + L^2 - (r1 + r2)^2 = 0$ (Equ. 5)



a b c

Fig 5 : (a) le déplacement d'un robot cause une collision avec un autre robot, (b) cette collision est corrigée en reculant le long de la ligne qui a permis le déplacement jusqu'au moment où les deux robots sont tangents. La variation d'orientation n'est pas modifiée. (c) grandeurs utiles pour trouver la valeur de $\Delta d'$; la grandeur D en rouge est la distance qui a permis de détecter la collision.

Gestion de collision en cas de recul du robot: Dans ce contexte, **vtran** étant négatif, on obtient une valeur négative de Δd. Cependant la méthode décrite plus haut reste valable ; il suffit de travailler avec la valeur absolue de Δd puis de prendre l'opposé de la valeur $\Delta d'$ obtenue.

2.3 Conditions à remplir pour la décontamination

Il faut remplir deux conditions pour qu'un robot puisse décontaminer (faire disparaître) une particule (Fig6) :

- Une collision a été détectée et corrigée comme décrit dans la section précédente
- **Après la correction** il y a ***alignement*** de l'outil de contamination avec le centre de la particule. Soit **RP** le vecteur reliant le centre R du robot au centre P de la particule, et le vecteur **Xr** du robot ; nous posons qu'il y a ***alignement*** si la valeur absolue de l'angle $\Delta\gamma$ entre ces deux vecteurs est tel que :

$$|\Delta\gamma| < \text{EPSIL\_ALIGNEMENT} \qquad \text{(Equ. 6)}$$

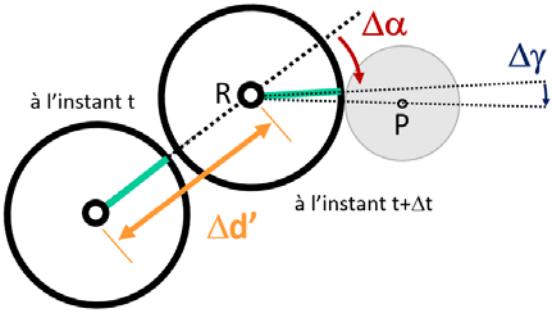


Fig 6 : après mise à jour de la position d'un robot avec correction de collision, la seconde condition à remplir pour faire disparaitre la particule est que l'écart angulaire $|\Delta\gamma|$ soit plus petit que EPSIL_ALIGNEMENT .

2.4 Contrôle d'un robot : méthode de base

C'est votre responsabilité de déterminer des valeurs de **vtran** et **vrot** à l'intérieur de leur intervalle autorisé qui permettent de se rapprocher de plus en plus d'une particule cible donnée à chaque mise à jour.

Par exemple, pour la vitesse de rotation on cherche à aller le plus vite possible mais sans dépasser l'orientation qu'on veut atteindre. On peut donc calculer l'écart angulaire $\Delta\gamma$ (exprimé entre $+\pi$ et $-\pi$) obtenu entre l'axe Xr du robot et le vecteur qui relie le centre du robot et le centre de la particule (Fig 6). Ensuite **vrot= $\Delta\gamma$ /DELTA_T** sauf si cette valeur est en dehors de l'intervalle autorisé [-VROT_MAX, VROT_MAX] ; dans ce cas on prend la borne autorisée la plus proche.

Par contre il est important de prendre en compte l'orientation relative du robot vis-à-vis de la cible pour déterminer **vtran**. En effet on ne peut que avancer/reculer selon l'axe Xr. L'idée de procéder séquentiellement, d'abord tourner sur place pour s'aligner sur la cible puis avancer en ligne droite, est correcte mais très lente. Or le but du projet est d'effectuer la décontamination dans le temps le plus court possible. Une idée à approfondir est la suivante :

- Tant que le robot tourne le dos à la cible ($|\Delta\gamma| > \pi/2$), on tourne seulement sur place (vtran=0)
- Si $|\Delta\gamma| < \pi/2$ alors **vtran = L/DELTA_T** avec une limitation supplémentaire dépendante de $\Delta\gamma$ en plus du respect de l'intervalle autorisé, **L** étant la distance entre le centre du robot et celui de la particule.

2.5 Coordination des robots

L'objectif des robots est d'atteindre l'ensemble des particules le plus vite possible pour les décontaminer sans se gêner mutuellement. Leur position initiale peut être quelconque (fournie dans un fichier).

Idée de l'algorithme de coordination : en première approximation, il est plus important d'atteindre les particules possédant le plus grand rayon en premier car elles ne gêneront plus les déplacements ensuite. Il faut donc trier les particules selon leur rayon puis les traiter en commençant pas le plus grand en lui affectant le robot qui peut la supprimer *le plus rapidement possible*. Cette durée doit prendre en compte l'écart en position **L** et l'écart en orientation $|\Delta\gamma|$ car les deux éléments sont importants comme discuté à la section précédente. On peut estimer cette durée comme si on travaillait en mode séquentiel, c'est-à-dire qu'on fait la somme de la durée minimum pour parcourir la distance **L** et de la seconde durée minimum pour corriger l'orientation $|\Delta\gamma|$.

Le robot choisi passe dans un état « occupé » et est associé à cette particule. Cet algorithme de coordination peut associer plus d'un robot à une particule à cause du risque de décomposition. Il est possible de ne pas utiliser tous les robots si on peut justifier que cela rallongerait le temps de décontamination, par ex. en causant des collisions entre les robots.

L'algorithme de coordination doit être ré-exécuté à chaque fois qu'une particule est éliminée ou décomposée en 4 nouvelles particules (section 3.1).

2.6 Aspects tactiques du contrôle des robots

L'algorithme de coordination décrit ci-dessus n'aborde pas toutes les décisions que doivent prendre les robots lors de leur déplacement. Il reste la gestion de plusieurs aspects *tactiques* qui pourraient être gérés par chaque robot à chaque mise à jour. Pour simplifier, on suppose que chaque robot peut exploiter la position des particules et la position et la vitesse des autres robots pour prendre ses décisions. Voici une liste de sous-problèmes **tactiques** que l'on peut *optionnellement* gérer de manière locale:

- *Comment traiter les autres particules qui se trouvent sur le chemin du robot* : ignorez-vous ces particules au risque de perdre du temps à cause des collisions ? choisissez-vous de l'éliminer dès qu'il y a une collision ? ou essayez-vous d'éviter cette collision ?
- *Votre robot peut perdre du temps à cause des collisions avec d'autres robots* : les ignorez-vous ou essayez-vous de les éviter ? (on ne peut pas les détruire).

Le minimum demandé dans ce projet pour la gestion des sous-problèmes tactiques est de *traiter les collisions lorsque le robot se déplace à l'aveugle vers sa cible sans se soucier de ce qui se trouve sur son chemin*. Un modeste bonus est prévu pour une tactique qui élimine les particules sur le chemin de sa cible lorsque cela permet de gagner du temps par rapport aux collisions seules. Attention, cet aspect peut prendre du temps par rapport aux points associés ; il faut savoir s'arrêter.

2.7 Mesure du taux et du temps de décontamination

Initialement le *taux de décontamination* est nul puisque toutes les particules sont présentes. Le temps de décontamination est la durée minimum à laquelle le taux atteint la valeur de 100% lorsque toutes les particules auront été éliminées. Soit S_i la somme initiale des énergies de toutes les particules et S_d la somme des énergies des particules éliminées. Le taux de décontamination en poucentage T_d est donné par :

$$T_d = 100. (S_d / S_i) \qquad \text{(Equ. 7)}$$

3 Actions à réaliser

Le but du programme est d'exécuter la simulation de la décontamination jusqu'à atteindre le taux de 100%. L'exécution de la simulation peut être activée ou en pause. Dans le mode Pause, il doit être possible de réaliser les actions suivantes par l'intermédiaire de l'interface graphique (détails section 5):

- **Lecture** d'un fichier pour initialiser l'état de la simulation (section 4). Avant de commencer la lecture elle-même il faut ré-initialiser les structures de données et libérer la mémoire si nécessaire. La simulation reste en mode Pause après la lecture du fichier.
- **Ecriture** d'un fichier décrivant l'état de la simulation à l'instant courant (section 4)
- **Lancement** de la simulation en continu ou seulement pour une seule unité de temps DELTA_T
- **Enregistrement** de l'évolution du taux de décontamination
- **Optionnel contrôle manuel d'un robot** en utilisant les touches du clavier

3.1 Structure de la simulation

Lorsqu'elle est activée la simulation consiste en une boucle de mise à jour calculant l'évolution de l'état de tous ses constituants après un intervalle de temps DELTA_T depuis la mise à jour précédente. La boucle s'exécute tant qu'on n'a pas atteint le taux de 100% de décontamination.

Pour ce projet, la mise à jour est *simplifiée* car on réagit immédiatement au déplacement d'un robot sans attendre que tous les autres robots soient aussi mis à jour. On adoptera la structure suivante pour un pas de mise à jour (= pour un seul DELTA_T):

1) Décomposition aléatoire des particules (section 2.1.3)

- Si au moins une particule s'est décomposée
    - Mise à jour de la **coordination** globale : chaque robot connaît sa particule cible (section 2.5)

3) Pour chaque Robot (prérequis: chaque robot connait sa particule cible qui existe encore)

- Analyser le contexte local du robot pour résoudre les aspects **tactiques** optionnels (section 2.6)
    - Déterminer la combinaison désirée de vitesse linéaire et la vitesse angulaire
    - Tester le déplacement du robot pour un seul DELTA_T
        - D'abord traiter les collisions avec les autres robots (reculer le robot courant)
        - Ensuite seulement, traiter les collisions restantes avec les particules (reculer le robot courant)
        - Si l'outil de décontamination est aligné avec le centre de la particule tangente
            - La particule est éliminée (mettre à jour le taux de décontamination)
- Si une particule a été éliminée
    - Mise à jour de la **coordination** globale : chaque robot connaît sa particule cible (section 2.5)

Tous les calculs de déplacement et de collision doivent être faits en virgule flottante double précision.

3.2 Tâche de bas-niveau

Plusieurs actions du projet ont besoin d'effectuer des calculs vectoriels et/ou géométriques pour prendre certaines décisions : détection de collision entre des cercles, détermination de distances et d'écart angulaire entre deux vecteurs, etc…

Du point de vue de la programmation modulaire, il faut factoriser autant que possible les opérations communes (Principe de Ré-utilisation). Pour ce projet, un module **utilitaire** de bas-niveau de gestion de point, de vecteur et de cercles dans l'espace 2D doit être crée pour gérer cet aspect ; ce module doit être indépendant des entités de la simulation, c'est-à-dire qu'il n'a aucune idée de ce qu'est un robot ou une particule. Il ne connait que des points, des vecteurs et des cercles dans le plan. L'implémentation **utilitaire.c** doit au moins offrir les fonctions documentées dans l'interface fournie **utilitaire.h** (détails en section 8.2.1 : **rendu1**).

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

Pour tester les rendus 2 et 3 de votre projet nous exécuterons plusieurs scénarios de complexité progressive dont une partie sera publique et disponible dans des fichiers. Votre programme doit être capable de lire de tels fichiers. Il doit aussi pouvoir mémoriser la configuration courante de la simulation dans un fichier. Cela vous permettra de pouvoir reproduire une simulation donnée et de créer vos propres scénarios de tests avec un éditeur de texte.

Caractéristiques de fichiers tests :

Le nombre maximum de caractères par ligne est de MAX_LINE.

Les lignes vides commençant par **\n** ou **\r**, les commentaires commençant par **#** précédé éventuellement d'espaces, et les espaces avant ou après les données doivent être ignorés ; *il peut y en avoir un nombre différent d'un fichier à un autre*.

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).

Le fichier utilise le mot clef du type **FIN_LISTE** pour la fin d'une liste non-vide de données et permettre une bonne vérification de la validité du fichier lu. Ce mot clef apparait seul sur une ligne mais peut être précédé d'éventuellement espaces ou tabulations qui ne sont pas obligatoires mais seulement destinés à rendre le fichier plus lisible pour un être humain. Si le nombre d'élément d'une telle liste est zéro, le mot clef **FIN_LISTE** ne doit pas être utilisé.

Voici le format général	remarques
# Nom du scenario de test **#** **nb Robot** **x0 y0 alpha0 x1 y1 alpha1 …** **…** **FIN_LISTE** **nb Particule** **e0 r0 x0 y0 e1 r1 x1 y1 …** **…** **FIN_LISTE**	Pour chaque robot on indique sa position: x et y, et son orientation α (en rd). Il y a un nombre entier de robots par ligne Pour les particules on indique son énergie, son rayon et sa position x et y. Il y a un nombre entier de particules par ligne.

5. Description de l'interface utilisateur (GUI)

Comme dans les séries 19-21, nous nous en tiendrons à deux fenêtres graphiques : une pour l'interface graphique et l'autre pour le dessin de la simulation.

La fenêtre d'interface utilisateur doit contenir (Fig 7):

- **Opening** : remplace la simulation par le contenu du fichier dont le nom est fourni.
- **Saving** : 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
- **Recording** : checkbox permettant d'indiquer si on désire enregistrer le taux de décontamination (section 2.7) pendant la simulation en vue d'afficher le résultat avec gnuplot (section 5.1).
- **Rate :** affiche la valeur courante du taux de décontamination
- **Turn** : affiche le compteur entier du nombre de mises à jour de la simulation
- **Control mode** : radio_button permettant de prendre le contrôle des vitesses d'un robot (cf 6.2).
  - Robot control : affichage des vitesses du robot manuellement contrôlé
- **Exit** : ferme les fenêtres et fichiers et quitte le programme

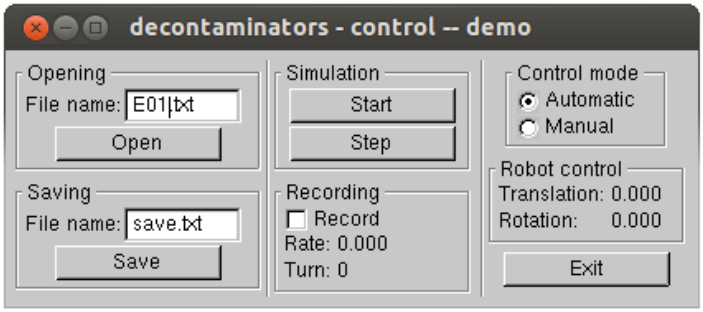


Fig 7 : Interface graphique (GUI)

5.1 Enregistrement du taux de décontamination pour affichage avec gnuplot

La Checkbox « Record » de l'interface graphique active/désactive l'ajout dans un fichier **out.dat**, après la mise à jour, de la **valeur du compteur entier du nombre de mises à jour** suivi par le taux de décontamination (ces valeurs sont séparées par un espace).

Le fichier **out.dat** doit être visualisable avec **gnuplot** dès le retour en mode Pause qui ferme ce fichier (et désactive la checkbox par la même occasion). L'utilité de **gnuplot** est de montrer l'évolution de cette grandeur au cours du temps. L'exemple de la figure 1b est obtenu avec la commande gnuplot suivante:

plot "out.dat" using 1:2 with lines

6. Affichage et interaction dans la fenêtre graphique

6.1 Affichage dans la Fenêtre de visualisation de la simulation :

Cette fenêtre sert à afficher l'évolution de la simulation. La taille initiale de l'espace visualisé est [**-DMAX, DMAX**] selon X et Y. Cet espace est matérialisé par le dessin d'un carré de cette taille ; il deviendra visible lorsque la proportion de la fenêtre ne sera plus un carré. Le cadrage est ajusté seulement lorsqu'on change la taille de la fenêtre ; cet ajustement doit être fait SANS introduire de distorsion dans le dessin.

6.2 Interaction avec la souris et le clavier dans la fenêtre de visualisation :

Control mode: lorsque l'option **Manual** du Control mode est activée, l'utilisateur peut sélectionner un des robots pour contrôler ses deux vitesses linéaire et angulaire.

(dé-)Sélection du robot : elle s'effectue en plaçant le curseur de la souris sur le robot et en appuyant sur le bouton gauche de la souris. Le robot passe dans l'état sélectionné et ses vitesses linéaire et angulaire deviennent nulles. On dé-sélectionne le robot en cliquant à nouveau avec le bouton gauche de la souris (n'importe où). Si on clique sur un autre robot celui-ci devient le nouveau robot sélectionné.

Quand il est dans l'état sélectionné, on peut modifier ses vitesses linéaire et angulaire avec les quatre touches « flèches » seulement de la façon suivante :

Touche -> : **soustraire DELTA_VROT** à la vitesse angulaire dans le but de le faire tourner sur sa droite
Touche <- : **ajouter DELTA_VROT** à la vitesse angulaire dans le but de le faire tourner sur sa gauche
Touche ^ : **ajouter DELTA_VTRAN** à la vitesse linéaire dans le but d'avancer plus rapidement
Touche v : **soustraire DELTA_VTRAN** à la vitesse linéaire pour ralentir et/ou reculer

Et pendant ce temps-là la vie continue… La simulation se poursuit pendant qu'on contrôle manuellement le robot sélectionné.

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 :** mis en œuvre avec Le module principal **main ;** c'est le seul module écrit en C++**.** Il est responsable de la gestion du dialogue utilisateur avec l'interface graphique (rassemble toutes les dépendances GLUT-GLUI et la gestion de la projection OPENGL). 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 la simulation (voir point suivant). Le module main.cpp est aussi responsable de traiter les arguments transmis au programme (section 8).

- **Sous-système du Modèle** (rectangle en pointillé dans la figure 8 et 9): est responsable de gérer la simulation. Il est mis en œuvre avec plusieurs modules sur plusieurs niveaux d'abstractions selon les Principe d'Abstraction et de ré-utilisation (section 7.2).

- **Sous-Système de Visualisation :** Les dépendances d'affichage avec OPENGL sont concentrées dans le module de bas niveau **graphic** écrit en C (fourni en TP et à compléter). Ce module offre des fonctions pour dessiner des éléments géométriques (ex : cercle)… Elles seront utilisées dans les modules du modèle pour le dessin des robots et des particules (Principe de Ré-utilisation).

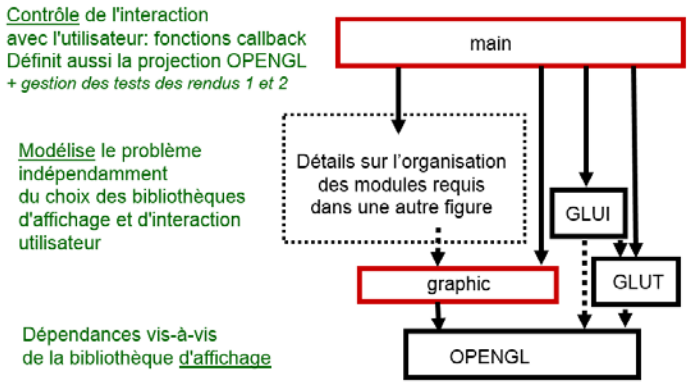


Fig 8 : Architecture logicielle minimale à respecter

7.2 Décomposition du sous-système Modèle en plusieurs modules

Le Modèle doit être organisé en plusieurs modules, tous écrits en C, pour maîtriser la complexité du problème et faciliter sa mise au point. La Figure 9c présente l'organisation minimale à adopter en termes d'organisation des modules :

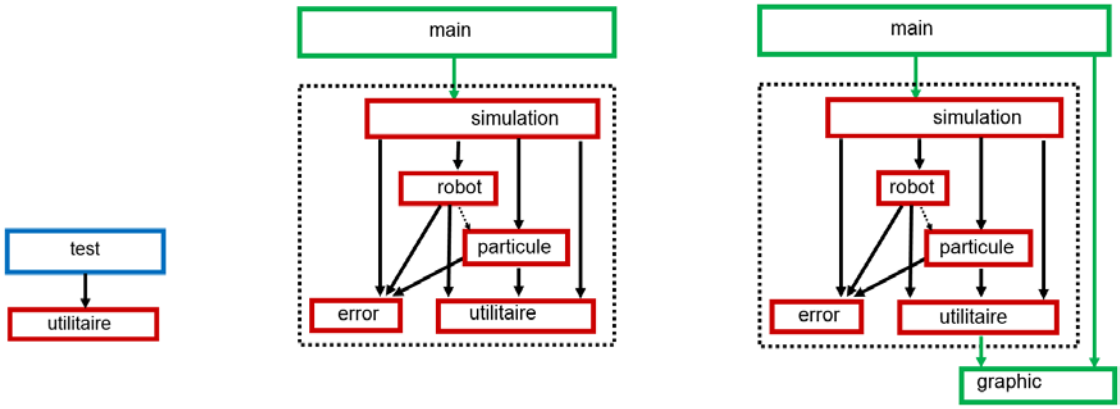
- ***Au plus haut niveau***, le module **simulation** gère l'évolution de la simulation et centralise les opérations de dessin, de lecture et d'écriture de fichier. Ce module doit garantir la cohérence globale du Modèle. C'est

pourquoi, en vertu du principe d'abstraction le module **simulation** *est le seul module* dont on peut appeler des fonctions en dehors du Modèle (Fig 9b et c)[1].

- Niveau(x) intermédiaire(s): **robot, particule.** Chaque module doit réaliser les actions utiles pour chaque entité (création, modification, destruction, lecture, écriture, dessin, etc…)**.**

- Module de bas niveau **utilitaire** (Principe de ré-utilisation): il est demandé de créer un module **utilitaire** de bas niveau mettant à disposition un ou plusieurs **types concrets** pour réaliser des opérations sur **des** éléments **2D** tels que des **points,** des **vecteurs**, des **cercles** (ex : opérations vectorielles, calcul de la position d'un point, etc…). Le **rendu1** porte sur l'écriture et le test (Fig 9a) d'un sous-ensemble des fonctions détaillées dans un document distinct. Pour le **rendu2** le module sera complété par les fonctions destinées à l'affichage graphique car il concentre les dépendances vis-à-vis du module **graphic** (Fig 9c).

- Module de bas niveau **error** : ce module est fourni. Il devra être utilisé lorsqu'il y a détection d'une erreur au moment de la lecture d'un fichier et de la vérification des superpositions interdites (**rendu2**).



a b c

Figure 9 : (a) architecture du test unitaire du module **utilitaire** pour le **rendu1** ; (b) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (mode **Error** détaillé en section 7.3); (c) dépendances supplémentaires vis-à-vis du module **graphic** pour l'évaluation de l'affichage graphique et de la simulation (mode **Draw** détaillé en section 7.3)**.**

Remarque : la dépendance entre les modules **robot** et **particule** n'est pas obligatoire ; on peut travailler avec le module simulation à la place d'un lien entre robot et particule.

7.3 Aperçu des trois rendus et syntaxe à respecter

7.3.1 Résumé des aspects importants:

Rendu1 : écriture et test du module utilitaire de bas-niveau seulement (Fig9a). PAS de graphique.
Rendu2 : test du format des fichiers et vérification des collisions initiales (3.1.1), initialisation de la fenêtre graphique et du GUI (section 5)
Rendu3 : simulation, enregistrement de l'évolution du taux de décontamination, performances, rapport.

7.3.2 Syntaxe à respecter pour le rendu1 (Fig9a)

Le rendu1 se concentre sur l'écriture du module **utilitaire** seulement. L'interface **utilitaire.h** est fournie ainsi qu'un module compilé **test.o** contenant une fonction main() ; vous devez écrire l'implémentation des fonctions dans **utilitaire.c** et faire l'édition de liens avec **test.o** pour produire un exécutable **test.x**.

7.3.3 Syntaxe à respecter pour le rendu2 et rendu3

Votre exécutable doit s'appeler **projet.x**.

[1] 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** comme le montre l'unique flèche de dépendance entre le module **main** et le module **simulation**.

On doit pouvoir lui transmettre deux arguments optionnels au lancement, sur la ligne de commande. La syntaxe est la suivante :

```
./projet.x [Error|Draw, nom_fichier]
```

Arguments optionnels à la suite du nom de l'exécutable: le premier argument désigne le mode de test du programme : on peut choisir **Error** ou **Draw**. Il faut ensuite ajouter un **nom de fichier** de test comme second argument. Exemples :

```
./projet.x    Error    E01.txt
./projet.x    Draw     D01.txt
```

Votre fonction **`main()`** est chargée de vérifier la cohérence de l'appel de l'exécutable. Par exemple, si **`argc`** vaut 3 alors **`argv[1]`** ne peut être que **Error** ou **Draw**. Si c'est le cas, alors une fonction du sous-système « modèle » doit être appelée pour gérer la lecture du fichier obtenu avec **`argv[2]`** et construire les structures de données.

Le mode **`Error`** sert à détecter s'il y a une erreur dans le fichier fourni. Le programme s'arrête dès la première erreur trouvée dans le fichier. Il s'arrête aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce mode il y a affichage d'un message indiquant le succès de la lecture. Ce mode n'a aucun affichage graphique.

Le mode **Draw** crée l'interface graphique du projet (GUI) et dessine dans la fenètre OPENGL l'état initial correspondant au fichier lu puis attend que l'utilisateur exploite l'interface graphique, par exemple pour donner un autre nom de fichier à lire, etc. Il contient aussi la détection d'erreur potentielle dans le fichier fourni. Cependant, lorsqu'une erreur est détectée il y a affichage du message d'erreur dans le terminal puis destruction des structures de données qui ont été construites.

Si **aucun argument** n'est transmis, le programme initialise l'interface graphique comme le programme de démonstration. On pourra ensuite utiliser le bouton **Open File** pour initialiser et exécuter une **simulation**. En cas d'erreur dans le fichier lu, le message d'erreur est affiché dans le terminal, les données lues sont supprimées et le programme attend qu'on lui demande de lire à nouveau un fichier.

ANNEXE A : constantes utilisées dans le projet

Fichier tolerance.h : constantes générales de bas niveau

```
#define  EPSIL_ZERO          1e-2        // en m
#define  EPSIL_ALIGNEMENT    0.0625      // en rd
```

Fichier constantes.h : constantes spécifiques au projet

```
#include "tolerance.h"
#define  DELTA_T             0.25        // en seconde
#define  VTRAN_MAX           0.75        // en m/s
#define  VROT_MAX            0.5         // en rd/s
#define  DELTA_VROT          0.125       // en rd/s
#define  DELTA_TRAN          0.25        // en m/s
#define  DMAX          20                // en m
#define  R_ROBOT             0.5         // en m
#define  R_PARTICULE_MAX     4           // en m
#define  R_PARTICULE_MIN     0.3         // en m
#define  R_PARTICULE_FACTOR  0.4142
#define  E_PARTICULE_MAX     1
#define  E_PARTICULE_FACTOR  0.25
#define  DECOMPOSITION_RATE  0.025
#define  MAX_LINE            120
```