Decontaminators

1 Implémentation

1.1 Architechture logicielle

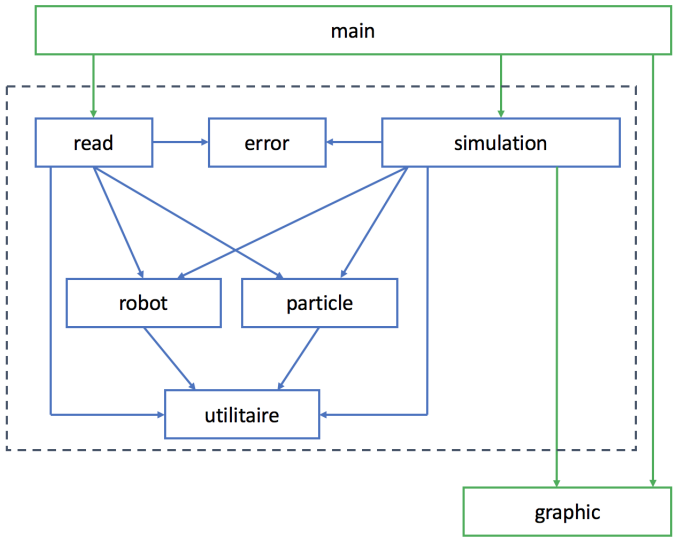

Figure 1 – architecture logicielle

Nous avons décidé d'ajouter un module pour la lecture et l'écriture des fichiers de sauvegarde. Premièrement car cela place l'automate de lecture entièrement dans un fichier, ce que nous trouvons plus cohérent. Deuxièmement car la lecture et le stockage des données ont été faites par différentes personnes. Travailler à deux personnes dans le même fichier semblait plutôt difficile lorsque nous avons réparti les tâches.

1.2 Structuration des données

Les données de la simulation sont structurées dans deux listes chainées, une pour les particules et une pour les robots. Pour accéder aux données nous avons créé un pointeur qui sert à sélectionner un robot ou une particule. Il est ensuite possible de sélectionner le prochain élement en appelant une fonction. Presque toutes les fonctions des modules *particle* et *robot* qui opèrent sur un élément, le font sur l'élément sélectionné.
Chaque élément robot contient les deux dernières positions et l'orientation du robot ainsi que son objectif actuel. Il comporte aussi un compteur de bloquage ainsi que des variables indiquant le controle manuel et la présence de coordonnées valides dans le champ cible. Les éléments particule stockent une position, un rayon et une énergie ainsi qu'une variable indiquant si la particule est ciblée par un robot. Chaque particule et chaque robot se voient également attribué deux identifiants et un pointeur "next" pour la liste chainée. De plus à chaque fois qu'une particule est créée dans la liste, elle est placée à la bonne place de façon à ce que la liste soit triée. La particule au plus grand rayon est pointée par la tête de liste.

1.3 Algorithme de coordination

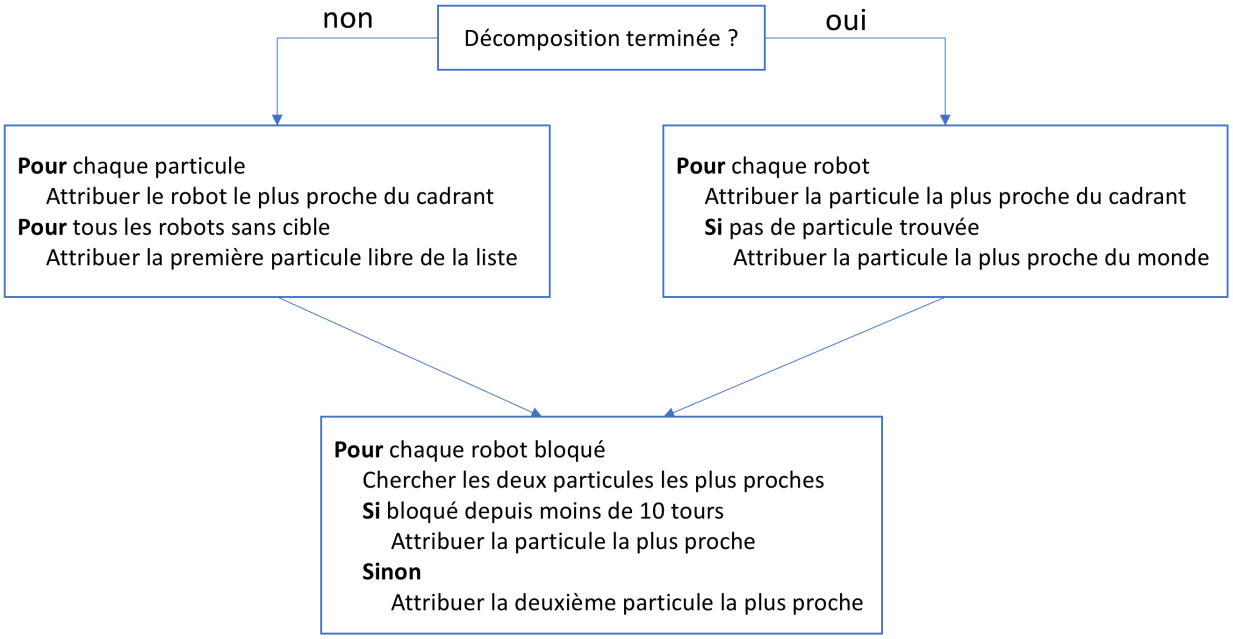

FIGURE 2 – schéma de l'algorithme de coordination

Au début de la simulation, s'il y a plus de quatre robots, chaque robot se voit assigné un cadran. C'est en quelque sorte son terrain de chasse. Pour les différentes méthodes d'assignation de cible, le robot va reçevoir une cible de son cadran en priorité. De plus chaque cadran recevra au moins un quart des robots. Le cadran sert a mieux répartir les robots dans le monde, grace à cette méthode ils ne se retrouveront pas tous au même endroit ce qui produira moins de collisions.
Sur le schéma si-dessus, on peut voir deux méthodes d'attribution des cibles. La première méthode est appliquée tant que les particules se décomposent. Une fois la décomposition terminée, il n'y a plus aucune raison de cibles les particules les plus grandes en premier car elles ne peuvent plus se décomposer. Nous attribuons donc à chaque robot la particule la plus proche. Lorsque un robot est bloqué pendant au moins un tour, il prend pour cible la particule la plus proche. Si un robot est bloqué pendant 10 tours, il prend pour cible la deuxième particule la plus proche. Ceci résoud le cas où un robot est bloqué entre deux particules.

1.4 Coût d'un pas

Pour calculer la complexité d'un pas, nous avons cherché la partie du pas la plus coûteuse. Il s'agit de la gestion des collisions et de l'attribution des cibles. Les deux parties ont la même complexité et les deux parties sont executées séquenciellement. Ceci nous laisse avec le coût calcul suivant :

$$O(nb\_robots * nb\_particules + nb\_robots^2) \tag{1}$$

Le coût mémoire d'un pas de simulation est de

$$O(nb\_robot + nb\_particule) \tag{2}$$

car tout le stockage se fait dans ces deux listes chainées. Il n'y a aucun stockage temporaire de particules ou de robots.

1.5 Illustrations

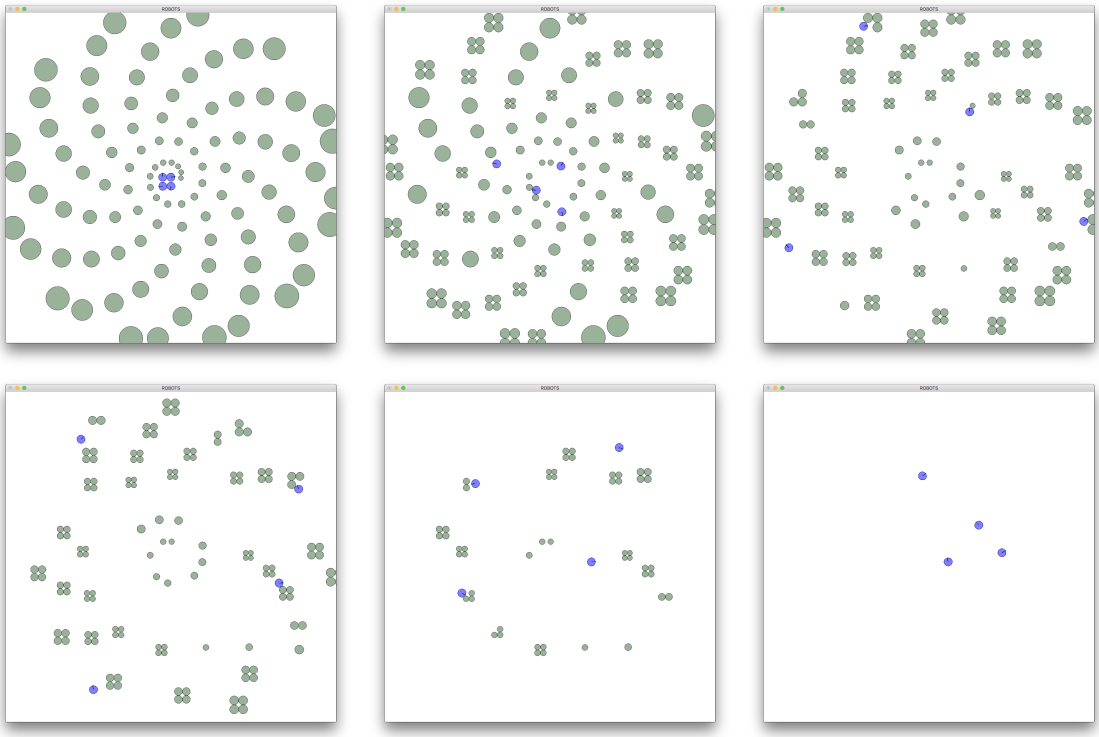

FIGURE 3 – Déroulement de la simulation (fichier : D03.txt) On peut constater que les robots vont d'abord vers particules les plus grandes (images 1 et 2), puis vers les particules plus proches (images 3,4,5 et 6) tout en restant dans leur cadran.

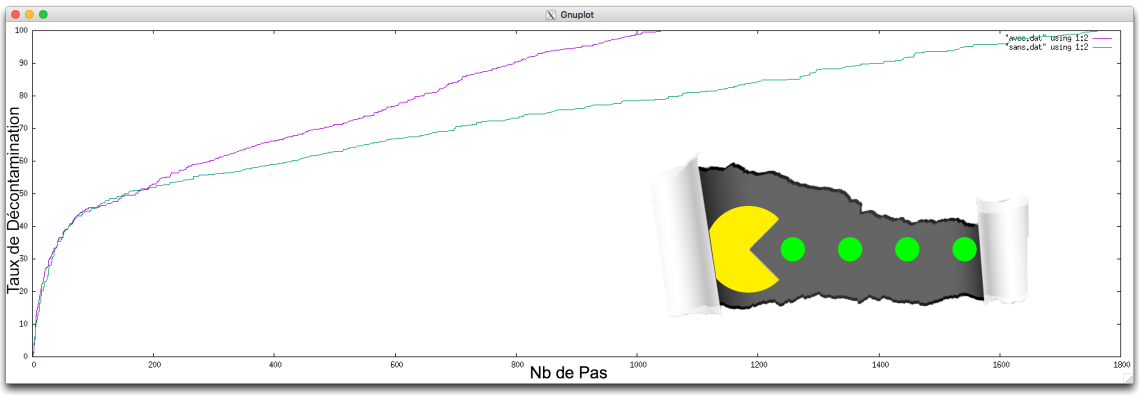

FIGURE 4 – Graphique obtenu avec gnuplot (fichier : D03.txt) En vert : algorithme ciblant les particules les plus grandes. En mauve : algorithme hybride qui cibles les plus grandes, puis les plus proches.

La *FIGURE 4* représente l'évolution du taux de décontamination. L'axe vertical représente le nombre de pas et l'axe horizontal représente le taux de décontamination. Ce graphe représente une simulation (violet) qui s'est terminée après 1040 pas et une simulation (vert) qui s'est terminée à 1762 pas. La simulation en violet est faite avec le changement de méthode d'attribution des cibles, lorsque les particules ont fini de se décomposer. Alors que dans la simulation en vert, les robots gardent la même technique et ciblent les plus grandes particules tout le long.
Le début du graphe est le même car les deux algorithmes sont identiques à ce moment. Pourtant, lorsque les particules ont fini de se décomposer l'algorithme ciblant les particules les plus proches est optimal. Le temps est beaucoup plus court, pour le graphe violet, car c'est vers le début de la simulation que les particules finissent de se décomposer. Ducoup c'est pendant plus des trois quarts du temps que l'algorithme utilisé est meilleur.

2 Conclusion

2.1 Organisation du travail

Le projet étant par groupe de deux, nous avons organisé notre travail en modules. Iacopo a réalisé les modules *graphic*, *robot* et *particle*, Lianyi a réalisé les modules *read*, *simulation* et *utilitaire*. Malgré cette répartition, nous avons passé beaucoup de temps à travailler ensemble. Nous avons commencé par faire la partie graphique, puis nous avons ajouté des fonctionnalités autour de celle-ci. Cela nous a permis d'avoir a un feedback visuel instantané à chaque ajout de fonction. Nous avons aussi créé toute une série d'outils de debug dans notre programme. Pour les activer il suffit de définir le symbole *DEBUG* dans le fichier *constantes*. Parmis ces outils, le plus utile est celui qui, lorsque l'on sélectionne un robot, affiche ses paramètres et fait apparaitre un point rouge sur sa cible. Lorsque confrontés au fameux *segmentation fault*, nous avons utilisé *GDB* qui est, selon nous, très utile dans ce cas.

2.2 Auto évaluation

La répartition des tâches dans les rendus intermédiaires nous ont permis d'organiser notre travail. Souvent nous avons passé beaucoup de temps a travailler ensemble sur le même problème plutôt que se répartir clairement les tâches. Cette méthode n'étais pas très rapide car il était souvent difficile de se mettre d'accord sur la façon d'implémenter certaines fonctionnalités.
Nous avons aussi passé beaucoup de temps à chercher certains bugs aux mauvais endroits. Par exemple, pour la gestion de collisions un décalage s'est introduit dans la position des robots. Leur position semblait être corrigée incorrectement par *util_inner_triangle*. Nous avons longtemps cherché le problème dans le module simulation, aors qu'il se trouvait dans le déplacement du robot. Le nouvel angle était modifié avant le déplacement dans la modification de la position du robot alors que l'angle était modifié après le déplacement dans la prévision servant à corriger une éventuelle collision, induisant une erreur minime à chaque déplacement.
Finalement, nous nous sommes beaucoup amusés a réaliser ce projet et nous avons laissé une petite surprise pour le correcteur.