


# C++ PoP – Sections Electricité et Microtechnique

Printemps 2021 : *Planet Donut*  © R. Boulic & collaborators

Quelle stratégie permettra aux bases de devenir autonomes ?

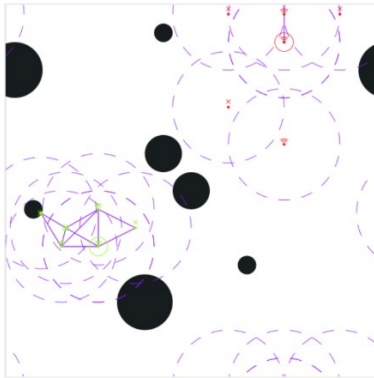
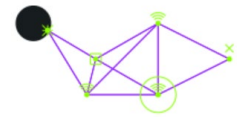


Fig 1 : exemple de planète comportant 7 gisements (cercles pleins noirs), 2 bases (les plus grands cercles de couleur) et un certain nombre de robots (représentation libre en réfléchissant au coût d'affichage). Les robots qui peuvent communiquer entre eux et/ou avec la base sont reliés par un trait. Les cercles en pointillés matérialisent le rayon maximum de communication. La base rouge n'a pas encore trouvé de gisement tandis que la base verte en exploite un. De plus la cohésion des robots de la base verte (image de droite) lui donne accès à l'information courante de tous ses robots.



Zoom-in sur la base verte et ses robots tous inter-connectés

## 1. Introduction

Ce projet est indépendant de celui du semestre dernier. La notion de graphe va néanmoins être utile ici aussi. 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 fonctionnalités* (*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 disponible dans le fichier **constantes.h** fourni en Annexe A. On utilisera toujours la *double précision* pour les données de position dans l'espace et les calculs en virgule flottante.

**Pitch du sujet** : on trouve dans le sous-sol de la planète Donut une ressource vitale que plusieurs bases spatiales automatisées cherchent à extraire (sans chercher à nuire volontairement aux autres bases). Pour cela chaque base dispose de robots spécialisés (prospection, forage, transport, communication). Les trois principales difficultés sont : 1) la position des gisements est inconnue, 2) la visibilité est nulle et 3) la communication entre robots et bases n'est possible que si la distance qui les sépare est faible. Les robots vont devoir s'organiser en réseau pour pouvoir transmettre de l'information à leur base afin que celle-ci prenne les bonnes décisions sous peine de disparaître par manque de ressource. Le but est de recueillir une quantité suffisante de ressource pour devenir autonome.

## 2. Modélisation des composantes de la simulation

**But :** Le but de ce projet est de mettre au point une simulation de la manière dont des bases spatiales s'organisent pour extraire la ressource de la planète Donut. Pour cela plusieurs algorithmes devront être proposés et testés conformément aux indications des rendus. Les décisions prises à chaque mise à jour devront s'appuyer sur les informations recueillies auprès des robots à l'échelle du réseau de communication qui s'établit dynamiquement entre les robots. Le rendu2 s'occupera de la lecture de fichier et de la mise en place de l'architecture modulaire du Modèle avec ses structures de données. Le dernier rendu intégrera **un dessin de l'état courant de la simulation après chaque mise à jour**. Cela sera complété avec une interface graphique pilotant quelques actions et affichant la valeur de l'état courant de chaque base afin de vérifier le bon fonctionnement de la simulation.

La simulation est effectuée 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** (pour fixer les idées cette unité de temps pourrait correspondre à 1h mais nous n'avons pas besoin de manipuler cette quantité). Nous demandons de structurer la mise à jour selon les étapes suivantes :

// entrée modifiée : ensemble des bases **taB**

**Boucle infinie de la simulation : un passage = une mise à jour**

```

Pour i de 1 à nbB           // dans le même ordre de celui du fichier de test
  Pour j de 1 à nbB         // dans le même ordre de celui du fichier de test
    update_voisin(taB[i].robots, taB[j].robots)
  connexion(taB[i])
  maintenance(taB[i])
  creation(taB[i])
  update_remote(taB[i].robots)
  update_autonomous(taB[i].robots)

Pour i de 1 à nbB           // dans le même ordre de celui du fichier de test
  Si taB[i].ressource ≤ 0
    destruction(taB[i]) // dont l'ensemble de ses robots

```

Pseudocode 1 : principe de la boucle de simulation (cf section 5 pour sa mise en œuvre)

Les composantes de la simulation sont décrites dans les sections suivantes.

### 2.1 L'espace se reboucle sur lui-même selon les deux axes

Nous allons simuler la géométrie de la **planète Donut** en effectuant la simulation dans un espace 2D qui se reboucle sur lui-même horizontalement (quand on sort du côté droit on rentre du côté gauche et vice versa) et verticalement (quand on sort du côté haut on rentre du côté bas et vice versa). Cette propriété de l'espace est illustrée ci-contre. La figure 2 montre aussi l'origine du système de coordonnées et l'orientation des axes X (horizontal) et y (vertical). L'espace est délimitée par les valeurs constantes  $[-dim\_max, dim\_max]$ . L'unité est le km.

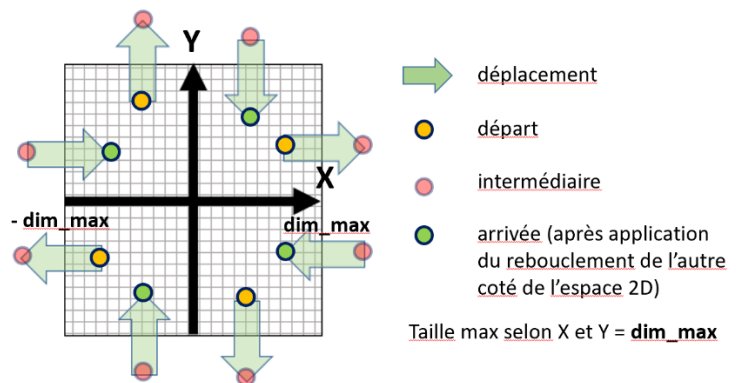


Figure 2 : propriété de rebouclage de l'espace 2D correspondant à la planète Donut

Les calculs géométriques (tests d'égalité de position sur nombre flottant, tests d'intersection, norme, etc...) et l'affichage des éléments de la simulation doivent tenir compte de la propriété de reboucllement. C'est pourquoi un module générique de bas niveau est responsable de cette gestion (section 7) afin de simplifier la mise au point des algorithmes au niveau de la simulation.

## 2.2 Ensemble des Gisements de Ressource

La planète contient un nombre **nbG** gisements de la ressource recherchée. Chaque gisement est approché par une forme circulaire caractérisée par son **centre** et son **rayon** compris entre **rayon\_min** et **rayon\_max** inclus. La quantité de ressource maximum initialement disponible dans un gisement est donnée par l'équation :

$$\text{gisementR} = 1000. * (\text{rayon} / \text{rayon\_min})^2 \quad (1)$$

Cette quantité diminue dès que le gisement est exploité par des robots de forage. Si la quantité courante est strictement supérieure à **deltaR**, un robot de forage peut extraire la quantité **deltaR** pour charger un robot de transport.

Les gisements ne se superposent pas. Leur position n'est pas connue des bases spatiales ; il faut déployer un ou plusieurs robots de prospection pour les trouver.

## 2.3 Les bases spatiales

La planète est exploitée par **nbb** bases spatiales qui sont en concurrence (pacifique) pour l'acquisition de la ressource. Plusieurs bases peuvent exploiter les mêmes gisements si elles savent où ils se trouvent. Une base spatiale elle-même est approchée par une forme circulaire avec un centre et un rayon **rayon\_base**. Les bases ne se superposent pas. Leur position est fixe. Initialement les bases ne connaissent pas la position des autres bases. Pour que cette simulation soit intéressante nous posons qu'une base se superpose pas non plus avec un gisement (cette vérification sera effectuée à l'étape de lecture de fichier).

Chaque base contient par défaut un robot de communication qui joue le rôle d'une antenne et qui y reste en permanence pour pouvoir établir la communication avec les robots qui ont quitté la base (sections suivantes). De même, chaque base contient une quantité initiale de ressource **iniR** qu'elle gère pour créer et entretenir des robots destinés à l'acquisition de la ressource. Une base peut créer jusqu'à **max\_robot** par mise à jour avec l'algorithme **creation** (Pseudocode 1). Si une base gère mal sa quantité courante de ressource celle-ci peut devenir nulle, ce qui entraîne sa destruction (avec ses robots).

### 2.3.1 But d'une base :

Le but d'une base est de recueillir une quantité de ressource **finR** qui lui permet ensuite d'être *autonome*. Dès qu'une base devient *autonome*, elle reste présente mais cesse toute activité d'exploitation des gisements. En particulier ses robots restent immobiles.

### 2.3.2 Critère de succès de votre stratégie :

Votre stratégie de gestion de la ressource est appliquée à toutes les bases de la même manière ; ce qui diffère est la présence ou pas de gisement dans le voisinage de la base. Il s'agit donc de proposer une stratégie robuste qui permette au plus grand nombre de bases de survivre indépendamment de la répartition des gisements sur la planète.

## 2.4 Les robots

Au niveau de chaque base, chaque robot est identifié par un numéro positif différent appelé **uid** (comme *unique id*). De même un robot n'est associé qu'à une seule base. La création d'un robot par la base lui coûte une quantité de ressource qui dépend de son type. De même un robot peut parcourir une distance maximum qui dépend de son type. Il doit donc tenir à jour un compteur de *distance parcourue* **dp**. Si un robot atteint son quota de distance maximum, il reste immobile mais peut quand même servir de relai de communication. Si un robot revient au centre de sa base avant d'avoir parcouru sa distance maximum, son compteur de distance parcourue **dp** doit être ré-initialisé à zéro par une opération de maintenance obligatoire effectuée par sa base.

Noter que la maintenance doit être faite avant la création de nouveaux robots (cf pseudocode1). Une opération de maintenance coûte à la base une quantité de ressource donnée par l'équation 2 :

$$\text{Coût de la maintenance} = \text{cout\_reparation} * dp. \quad (2)$$

En plus de sa distance max et de sa distance parcourue, un robot est caractérisé par sa position courante, par la position d'un **but** qu'il doit atteindre et par un booléen **atteint** indiquant si ce but est actuellement atteint. Tous les robots peuvent se déplacer de manière omni-directionnelle, c'est-à-dire qu'ils peuvent complètement changer de direction d'une mise à jour à la suivante. Un robot en déplacement parcourt systématiquement une distance de **deltaD** par unité de temps en direction de son but sauf pour le dernier déplacement pour lequel cette distance peut être inférieure. La direction du déplacement est déduite du vecteur reliant la position courante du robot à celle de son but.

Par ailleurs on suppose qu'un robot sait toujours implicitement la position de sa base (cette méthode appelée *odométrie* est en réalité peu précise mais on ne s'en soucie pas pour ce projet) ; il sait dans quelle direction se diriger pour revenir à la base. S'ils sont en déplacement vers leur but, les robots ne peuvent pas réaliser certaines actions pendant l'unité de temps de la mise à jour comme détaillé ci-dessous.

Une base peut produire et gérer quatre types de robots dont voici les propriétés et les actions possibles.

#### **2.4.1 Prospection** : (coût de création : **cout\_prosp** , distance maximum : **maxD\_prosp** ).

Si son but est atteint et que le robot n'est pas en déplacement, une seule mise à jour suffit pour déterminer si sa position courante est au-dessus d'un gisement. S'il y a succès, son booléen **found** passe à vrai et il mémorise la position du **centre du gisement**, son **rayon** et la **capacité actuelle du gisement**. Si c'est jugé utile, un booléen de **retour** passe à vrai pour exprimer que le but du robot de prospection est sa base, par exemple pour ré-initialiser sa distance parcourue à zéro. Une autre option est de poursuivre la prospection ; dans ce cas la base fait passer le booléen **found** à faux, ce qui indique que les coordonnées mémorisées pour le gisement ont été utilisées par la base et qu'il s'agit de trouver un gisement différent de celui qui est mémorisé. Pour cela la base indique un nouveau but à atteindre.

En respectant les indications ci-dessus, vous devrez proposer un `algorithme de prospection` mis en œuvre par la base pour décider et actualiser nombre de ces robots et leur but (mode REMOTE). Un robot qui n'est pas en contact avec la base est en mode AUTONOMOUS (section 2.5.4).

#### **2.4.2 Forage** : (coût de création : **cout\_forage** , distance maximum : **maxD\_forage** ).

Un robot de forage ne peut plus bouger une fois le forage effectué. Etant donné son coût important, sa faible distance maximum et son immobilisation définitive sur un gisement, un tel robot ne doit être envoyé que lorsqu'un gisement suffisamment intéressant est trouvé. Il faut choisir comme but du robot de forage le point du cercle du gisement qui est le plus proche de la base.

Si son but est atteint, le forage est immédiatement opérationnel. Le robot peut accéder à la capacité actuelle du gisement à chaque mise à jour. Chaque mise à jour lui permet de délivrer au maximum une quantité de ressource **deltaR** si 1) la capacité actuelle du gisement est strictement supérieure à **deltaR** et si 2) un robot de Transport a atteint le robot de forage. Il en résulte que un seul robot de transport peut être chargé par mise à jour.

En respectant les indications ci-dessus, vous devrez proposer un `algorithme de decision de forage` mis en œuvre par la base pour décider et actualiser le nombre de ces robots et leur but (mode REMOTE). La base peut obtenir la capacité actuelle du gisement grâce au robot de forage s'ils sont en communication. Un robot qui n'est pas en contact avec la base est en mode AUTONOMOUS (section 2.5.4).

### **2.4.3 Transport** : (coût de création : **cout\_transp** , distance maximum : **maxD\_transp** ).

Le but d'un robot de transport doit être un robot de forage auprès duquel il doit faire un chargement de ressource pour le ramener à sa base. Si son but est atteint, une seule mise à jour suffit pour charger une quantité de ressource **deltaR** auprès du robot de forage. C'est le robot de forage qui détermine quel robot de transport est chargé pendant la mise à jour au cas où plusieurs sont présents simultanément. Un booléen de **retour** passe à vrai pour exprimer que le chargement est effectué et que le but du robot de transport est le centre de sa base.

Du point de vue de la base, un aspect à prendre en compte est la possible compétition entre plusieurs bases sur un même gisement. Cela peut conduire à y envoyer plus de robots de transport si un gisement est connu d'une seule base. Inversement, sachant que la capacité d'un gisement est faible une base peut décider de ne plus envoyer autant de robots de transport car le gisement sera plus rapidement épuisé.

En documentant les choix effectués pour ces cas de figure, vous devrez proposer un `algorithme de decision de transport` mis en œuvre par la base pour décider et actualiser le nombre de ces robots et leur but (mode REMOTE). Un robot qui n'est pas en contact avec la base est en mode AUTONOMOUS (section 2.5.4).

### **2.4.4 Communication** : (coût de création : **cout\_com** , distance maximum : **maxD\_com** ).

Comme déjà mentionné, chaque base doit contenir en son centre un robot de communication (fixe) pour communiquer avec les autres robots afin d'obtenir des informations utiles pour la planification :

- Position et rayon d'un gisement
- Capacité actuelle d'un gisement
- Position des robots
- Booléens indiquant leur état

Ces informations sont utilisables par la base pour prendre ses décisions (création de robots, détermination ou changement du but d'un robot).

Cependant un robot peut seulement communiquer à une distance maximum de **rayon\_com**. C'est pourquoi des robots de communication supplémentaires peuvent être envoyés en dehors de la base pour servir de relai et transmettre l'information de proche en proche jusqu'à la base (Fig 1 image de droite). Une fois atteint son but, le robot de communication y reste pour toujours ; c'est une simple antenne.

Dans ce projet, vous devrez proposer un `algorithme de placement de robot de communication` mis en œuvre par la base pour décider et actualiser le nombre de ces robots et leur but (mode REMOTE). Un robot qui n'est pas en contact avec la base est en mode AUTONOMOUS (section 2.5.4).

Un robot de communication peut transmettre de l'information même en déplacement vers son but.

## **2.5 Communication entre bases et robots**

Tous les robots peuvent servir de relai de communication quelle que soit leur action courante. Au début de chaque cycle de mise à jour, chaque robot doit actualiser sa liste des robots voisins (à l'intérieur du rayon **rayon\_com**) pour savoir avec qui il peut communiquer (réalisé par **update\_voisin** dans le Pseudocode 1).

### 2.5.1 Usage du graphe des robots voisins pour modéliser la communication :

On tire parti du graphe que l'on peut construire en posant que chaque robot est un nœud de ce graphe. L'ensemble des voisins de chaque robot permet de tenir à jour une **liste d'adjacence** pour chaque nœud.

Une base est représentée par un seul nœud : celui de son robot de communication qui y reste en permanence.

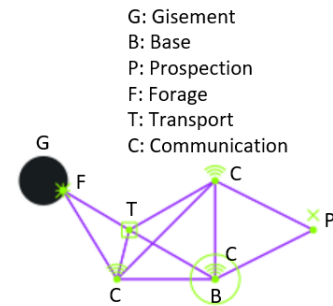


Fig 3 : réseau totalement connecté

**Simplification de l'accès à l'information** : Dans ce projet nous ne simulons pas la transmission des données de nœud en nœud ; la base, représentée par le nœud de son robot de communication, peut accéder à l'information mémorisée dans un autre nœud s'il existe un chemin entre les deux nœuds. On ne se préoccupe pas non plus de chercher un chemin optimal ; l'existence d'un chemin suffit. La figure 3 montre que tous les robots de la base verte sont reliés à leur base. Une base peut ainsi collecter l'information des robots, un par un.

Par contre, une base ne peut pas « copier » les informations accumulées par une autre base.

### 2.5.2 Algorithme de connexion :

Vous devrez proposer l'algorithme de **connexion** (Pseudocode 1) qui détermine le *mode de contrôle* de chaque robot :

- **REMOTE** : contrôlé par sa base car le robot y est connecté
- **AUTONOMOUS** : il décide de son action de manière autonome car pas de chemin vers sa base

Cet algorithme doit aussi construire pour chaque base *une structure de donnée qui mémorise l'état des ensembles de robots rencontrés en parcourant son réseau de communication*.

### 2.5.3 Contrainte de perte de mémoire des bases à la fin de chaque mise à jour :

Nous posons qu'il n'est pas possible de mémoriser au-delà de la mise à jour courante la structure de donnée construite par l'algorithme de connexion : cette structure de donnée disparaît à la fin de chaque mise à jour.

De même une base ne peut pas mémoriser l'ensemble des gisements au-delà de la mise à jour courante.

En effet, ni le mode de contrôle ni cette structure de donnée ne sont mémorisés dans un fichier quand on fait une sauvegarde de la simulation. Cela compliquerait trop le projet de le faire.

Dans ce projet, Les autres algorithmes utiliseront le mode de contrôle et la structure de donnée construite par l'algorithme de connexion pour prendre les décisions de création de robot et de détermination des actions des robots (but, etc).

### 2.5.2 Interconnexion des réseaux de communication :

Il y a interconnexion des réseaux de deux bases si des robots des deux réseaux sont à une distance inférieure à **rayon\_com**. L'interconnexion peut aussi se produire au niveau de nœuds isolés de bases différentes mais cela n'est pas exploité par les robots isolés ; il y a seulement une visualisation de l'interconnexion.

**Conséquences** : dès qu'il y a interconnexion, chaque base a accès aux informations des robots de l'autre réseau. Une base doit s'en servir pour prendre ses décisions de prospection, de forage et de transport.

La mise en œuvre de l'interconnexion des réseaux constitue la tâche la plus avancée du projet. Il faut s'assurer que les autres aspects fonctionnent correctement avant de s'engager à la mettre en œuvre.

### 2.5.3 Communication et contrôle du point de vue d'une base **b** :

- **S'il existe un chemin quelconque entre un robot et la base **b****, celle-ci utilise la structure de donnée qu'elle a construite avec l'algorithme de **connexion**. Dans ce contexte les robots associés à une base **b** sont contrôlés en mode REMOTE par cette base **b**. Les robots associés à d'autres bases **{b'}** ne peuvent pas être contrôlés par la base **b**.
- **Sinon**, la base **b** sait seulement que son robot existe mais elle ne peut pas le contrôler directement ni savoir son état courant. On pose que le robot est en mode AUTONOMOUS (cf section 2.5.4).

### 2.5.4 Communication et contrôle du point de vue d'un robot :

- **S'il existe un chemin quelconque entre le robot et la base **b** qui l'a créé**, celui-ci est piloté en mode REMOTE par sa base ; il faut mettre à jour son état courant auquel la base **b** a accès.
- **Sinon**
  - **s'il existe un chemin quelconque entre le robot et une base **b'** qui ne l'a pas créé ou seulement entre le robot et d'autres robots quelconques**, il est en mode AUTONOMOUS. Dans ce mode il cherche seulement à atteindre son but initial. Une fois celui-ci atteint le robot effectue sa tâche (prospection, forage, transport) puis revient vers sa base (prospection, transport) ou reste sur son but (communication).

## 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)**
- **Lecture** d'un fichier pour initialiser l'état de la planète (section 4).
- **Ecriture** d'un fichier décrivant l'état actuel de la planète (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

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

Votre programme doit être capable d'initialiser l'état de la simulation de la planète Donut à partir d'un fichier texte. Il doit aussi pouvoir mémoriser la configuration actuelle de la planète 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).

Le format de fichier est décrit dans le tableau suivant. Le nombre maximum de caractères par ligne est de **max\_ligne**. Le fichier contient d'abord les données des gisements puis les données des bases incluant leurs données de haut niveau puis leurs listes de robots avec leurs données spécifiques. Il doit y avoir une ligne de fichier par robot. Le format indique le nombre et la nature des données.

## format général du fichier

```

# Nom du scenario de test
#
nbG # passer à la ligne ; puis seulement un gisement par ligne
    x y rayon capacite

nbB # passer à la ligne pour fournir les donnée d'une base et ses robots
    x y ressource nbP nbF nbT nbC
        # nbP robots prospecteurs ; seulement 1 robot par ligne
        uid dp x y xb yb atteint retour found xg yg rayong capaciteg

        # nbF robots forage ; seulement 1 robot par ligne
        uid dp x y xb yb atteint

        # nbT robots transport ; seulement 1 robot par ligne
        uid dp x y xb yb atteint retour

        # nbC robots communication ; seulement 1 robot par ligne
        uid dp x y xb yb atteint

    # s'il y a plus d'une base, fournir ses données à partir de la ligne suivante

```

Dans ce format **x** et **y** désignent les coordonnées de la position d'une entité, **xb** et **yb** sont les coordonnées d'un but de robot, **xg** et **yg** celles d'un gisement trouvé par un robot de prospection. La distance parcourue est indiquée par **dp**. Les données **uid** sont les identificateurs uniques des robots (**unique au niveau de chaque base**). La quantité de ressource d'un gisement est indiquée par **capacite** ; celle gérée par une base est notée **ressource**. La capacité du gisement trouvé par le robot de prospection et foré par le robot de forage est notée **capaciteg**. Les données **atteint**, **retour** et **found** sont des booléens expliqués en section 2.4. Pour le cas particulier de **found**, si ce booléen est *faux* cela veut que le robot de prospection est à la recherche d'un gisement et ne l'a pas encore trouvé : dans ce cas précis les données **xg yg rayong capaciteg** doivent être absentes du fichier.

## 4.2 Calcul de la liste d'adjacence

La liste d'adjacence est la liste des robots voisins ; elle n'est pas mémorisée dans le fichier. En effet il est moins risqué d'utiliser une fonction pour faire ce calcul automatiquement après la lecture plutôt que d'écrire manuellement cette information dans le fichier. C'est pourquoi, pour le dernier rendu, on demande d'ajouter ce calcul après la lecture du fichier à l'aide de la fonction **update\_voisin()** aussi utilisée pour la simulation (Pseudocode 1).

Cette fonction doit être appelée indépendamment de la boucle de simulation car la simulation n'est pas lancée automatiquement après une lecture. Le lancement de la simulation doit être fait indépendamment depuis l'interface graphique.

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

La **lecture** doit vérifier les conditions de non-superposition des gisements, des bases et entre gisements et bases. Il faut également vérifier qu'aucune valeur de uid n'est dupliquée. Enfin, il faut vérifier que chaque base possède au moins un robot de communication dont la position est le centre de la base.

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):

- **Rendu2** : Dès la première erreur détectée à la lecture, il faut afficher dans le terminal le message d'erreur fourni avec le module **message** (section 8) et quitter le programme avec **exit()**.



- **Rendu3**: 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 avec le module **message** (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 trois parties:

- Les boutons des actions ou paramètres d'affichage graphique (Fig 4a en haut à gauche, et Fig4b)
- le dessin de l'état courant de la planète dans un *canvas* (Fig 1 et Fig 4a en haut à droite).
- L'affichage de l'état courant des bases (Fig4a partie basse)

### L'interface utilisateur doit contenir (Fig 4b):

Commandes générales :

- **Exit** : quitte le programme de simulation
- **Open** : remplace la planète par le contenu du fichier dont le nom est fourni. En cas d'erreur détectée à la lecture, les structures de données sont supprimées, ce qui produit un écran blanc.
- **Save** : mémorise l'état actuel de la planète dans le fichier dont le nom est fourni. L'ORDRE des bases doit être LE MEME que dans le fichier lu.

Simulation :

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

Paramètres des options d'affichage :

- **Toggle link** active l'affichage des liens (actif par défaut)
- **Toggle range** active l'affichage des cercles **rayon\_com** (non-actif par défaut)

Affichage de l'état courant des bases dans l'ordre suivant :

uid, nombres de robots : Prospecteur, Forage, Transport et Communication, niveau de la ressource (valeur brute et en % du niveau final).



a



b

**Fig 4 : Interface graphique (GUI)**

**Important** : l'ORDRE des bases dans l'interface graphique doit être LE MEME que l'ordre des bases dans le fichier lu. Si une base est détruite on peut soit avoir une ligne vide ou avoir une ligne de moins.

## 6. Affichage et interaction dans la fenêtre graphique

A partir du rendu 3, l'exécution du programme ouvre une fenêtre GTKmm contenant l'interface graphique utilisateur (Fig 4) et le dessin de la planète dans un *canvas*, qui couvre l'espace  $[-dim\_max, dim\_max]$  selon X et Y. L'espace 2D de la planète Donut est représenté seulement par le carré de sa bordure extérieure.

**Taille de la fenêtre d'affichage en pixels** : La taille initiale du *canvas* dédié au dessin de la planète est de **taille\_dessin** en largeur et en hauteur. 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 (le carré de la planète Donut reste un carré, un cercle reste un cercle, quelle que soit la taille et la proportion de la fenêtre). La Fig1 montre un exemple d'affichage du jeu.

**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 :

- La bordure du carré 2D de la planète Donut est **grise**
- Les gisements devront être **des cercles** remplis en **noir**. Le programme de démo ajoute la visualisation de la quantité extraite du gisement avec un arc en blanc (cette visualisation n'est pas exigée).
- Si **activé** : les liens du réseau de communication doivent être **des lignes** dessinées en **violet**.
- Si **activé** : **les cercles** de communication de rayon **rayon\_com** sont dessinés en **gris**.
- Chaque base est **un cercle** plein d'une couleur primaire. Elles sont dessinées dans l'ordre des **couleurs primaires** : **rouge, vert, bleu, jaune, magenta, cyan** et dans l'ORDRE des bases dans le fichier lu ; la première base lue dans le fichier utilise la couleur rouge, la suivante la couleur verte, etc... S'il y a plus de bases que ces 6 couleurs prédéfinies dans **graphic** (section 7) alors on ré-utilise les mêmes 6 couleurs primaires avec un modulo. Une base conserve toujours la MEME couleur pendant toute la durée de la simulation, même si d'autres bases sont détruites au cours du temps.
  - Le rayon d'une base étant trop petit pour que ce cercle soit visible, nous demandons que sa **visualisation** utilise un **rayon de 10\*rayon\_base**.
- Les robots sont de la même couleur que leur base. **Un point suffit pour les représenter** ; vous êtes libres de choisir une forme plus complexe pour les distinguer les uns des autres (à définir dans **geomod**) mais attention au coût de l'affichage graphique.

## 6.1 Interaction avec le clavier

Utiliser la touche clavier 's' pour faire la même action que le bouton Start (et mettre à jour l'interface)

Utiliser la touche clavier '1' pour faire la même action que le bouton Step

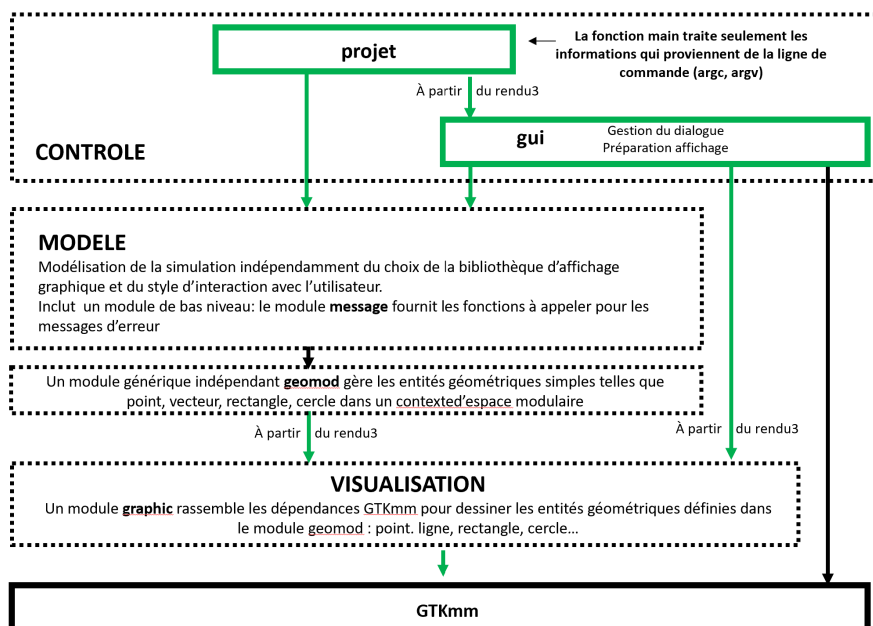


Fig 5 : Architecture logicielle minimale à respecter

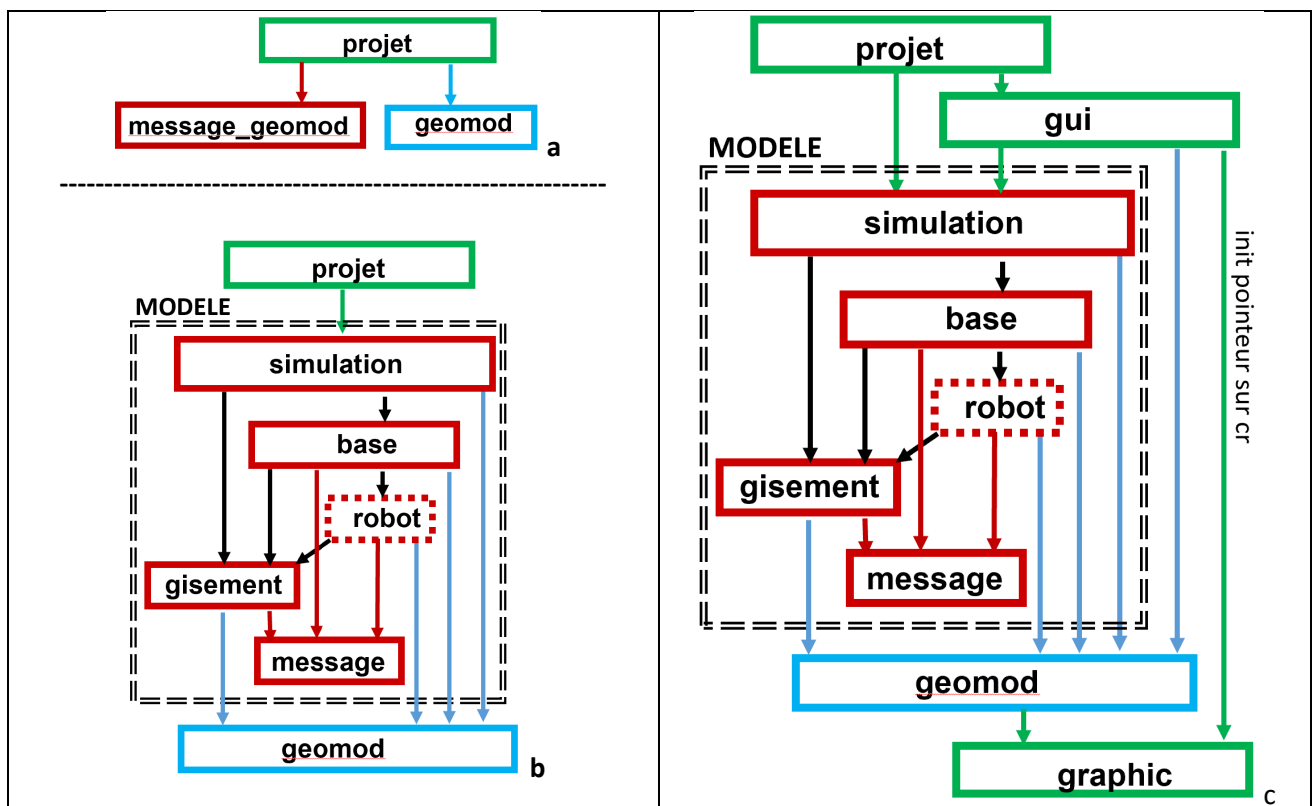
## 7. Architecture logicielle

### 7.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 5 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. 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 rendu3 pour rassembler 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 de la planète. 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 **geomod** gère les entités géométriques simples telles que point, vecteur, rectangle, cercle dans un espace modulaire ; 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 **geomod**. Le module **graphic** rassemble les dépendances vis-à-vis de GTKmm pour faire les dessins.



**Figure 6** : (a) architecture pour le test du module geomod (rendu1) ; (b) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (rendu2); (c) modules et dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendu3)

## 7.2 Décomposition du sous-système **MODELE** en plusieurs modules

Le Modèle doit être organisé en plusieurs modules, pour maîtriser la complexité du problème et faciliter sa mise au point. La Figure 6 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). 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 6)<sup>1</sup>.

- **Niveau intermédiaire**: il faut au moins considérer un module distinct pour les entité de gisement, base et de robot. Le module **robot** doit être conçu comme une hiérarchie de classes pour intégrer les 4 sortes de robots.
- **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 détectés à la lecture d'un fichier. Une fonction 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 programme de notation automatique.

### 7.3 Module générique indépendant geomod pour les calculs géométriques modulaires

#### 7.3.1 Indépendance de geomod

L'idée fondamentale concernant ce module est qu'il doit être conçu pour être utilisable par d'autres programmes très différents de Planet Donut. 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). Pour cette raison **AUCUN des noms de types/concepts de Planet Donut ne doit apparaître dans geomod** ; on doit seulement y trouver des types génériques tels que point, vecteur, rectangle, cercle dans l'espace 2D.

Inversement, ce sont les classes du MODELE qui vont contenir des attributs de type point, vecteur, rectangle ou cercle mis à disposition par **geomod** et qui vont appeler les fonctions mises à disposition pour traiter tous les calculs géométriques modulaires.

#### 7.3.2 Usage autorisé de types concrets pour les types de geomod

Un type **concret** est typiquement une structure dont on montre le modèle dans l'interface du module (**geomod.h**) de façon à *pouvoir accéder aux champs* de cette structure dans tous les modules de haut-niveau qui déclarent des variables de ces types.

Il est tout à fait pertinent de mettre à disposition de tels types concrets pour des entités de bas niveau comme des points, vecteur, rectangle et cercle. La programmation orientée objet n'interdit pas l'usage de types concrets s'ils sont utilisés dans le bon contexte ; **geomod** en est un car les entités offertes par ce module sont élémentaires et seront validées et stabilisées dans le premier rendu du projet.

### 7.4 Module graphique de bas-niveau (graphic) (à partir du rendu3)

La visualisation du MODELE se fait par l'intermédiaire des éléments géométriques élémentaires mis à disposition dans **geomod**. Ce module **geomod** va offrir des fonctions de dessin qui vont prendre en compte la nature modulaire de l'espace, comme pour les autres calculs géométriques (détails dans le document du rendu1). Par exemple une fonctions *cercle\_dessin(paramètre\_de\_type\_cercle)* va vérifier si le cercle déborde de l'autre côté de l'espace 2D (comme un cercle sur le côté gauche dans Fig1) et si nécessaire va demander de dessiner un cercle supplémentaire (au-delà du côté droit) pour faire apparaître le morceau de cercle qui reboucle dans la Fig1. Cela peut conduire à dessiner jusqu'à 4 cercles.

Cependant le module **geomod** doit rester indépendant d'une librairie 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 de cercle, rectangle ou de droites (cf section 6 pour les conventions à suivre pour les formes et les couleurs).

<sup>1</sup> 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**.

## 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 certains arguments optionnels sur la ligne de commande.

**8.1 Rendu1** : Son architecture très simple de type *scaffolding* est précisée par la Fig 6a. Il s'agit de tester l'ensemble des fonctionnalités que doit offrir le module **geomod**. Voici la syntaxe d'un appel :

```
./projet i a b c d e ...
```

La valeur indiquée par l'argument **i** désigne le scénario du test à effectuer. Le reste des arguments sert à fournir les paramètres du test.

**8.2 Rendu2** : Son architecture est précisée par la Fig 6b.

Ce rendu SANS GTKmm sera toujours testé 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 en vérifiant les points suivants à la lecture du fichier de test :

- Absence de collision entre gisements, entre bases, entre gisements et bases
- Absence de duplication d'uid **de robots au niveau de chaque base**
- Présence d'un robot communication au centre de la base
- ~~Cohérence entre ensemble des gisements et position/rayon mémorisée par un robot forage~~

Un rapport devra décrire les choix de structures de donnée minimales (initialisation et calcul des voisins).

**8.3 Rendu3** : Ce rendu utilise toujours GTKmm (avec l'architecture de la Fig 6bc). 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 des bases de la planète.

Ce rendu sera testé en effectuant plusieurs lecture/écriture/relecture avec le GUI pour vérifier que l'affichage est bien correct. Avant de commencer la lecture d'un fichier, il faut ré-initialiser les structures de données et libérer la mémoire. Plusieurs scénarios de simulation seront testés.

Un rapport final devra décrire les algorithmes mis en œuvre par les bases.

## **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.

constexpr double <b>dim_max</b> (1000.) ;	km
constexpr double <b>rayon_min</b> (50.) ;	km
constexpr double <b>rayon_max</b> (150.) ;	km
constexpr double <b>rayon_base</b> (1.) ;	km
constexpr double <b>rayon_com</b> (300.) ;	km
constexpr double <b>deltaD</b> (5.) ;	km
constexpr double <b>maxD_prosp</b> (10*dim_max.) ;	km
constexpr double <b>maxD_forage</b> (1.42*dim_max) ;	km
constexpr double <b>maxD_transp</b> (5*dim_max) ;	km
constexpr double <b>maxD_com</b> (1.42*dim_max) ;	km
constexpr unsigned int <b>max_robot</b> (3) ;	
constexpr double <b>iniR</b> (1000.) ;	ktonne
constexpr double <b>finR</b> (10*iniR) ;	ktonne
constexpr double <b>deltaR</b> (iniR/4.) ;	ktonne
constexpr double <b>cout_reparation</b> (0.0005) ;	ktonne/km
constexpr double <b>cout_prosp</b> (iniR/100.) ;	ktonne
constexpr double <b>cout_forage</b> (iniR/10.) ;	ktonne
constexpr double <b>cout_transp</b> (iniR/10.) ;	ktonne
constexpr double <b>cout_com</b> (iniR/1000.) ;	ktonne
constexpr unsigned <b>max_ligne</b> (80);	nombre de caractères par ligne de fichier

---

## **ANNEXE B : constante destinée au sous-système de Contrôle**

constexpr unsigned <b>taille_dessin</b> (800);	en pixels
--	-----------