V 1.0

Projet Informatique – Sections Electricité et Microtechnique

Printemps 2017 : *Bugs' Life* © R. Boulic

Votre simulation converge-t-elle vers un état stable ?

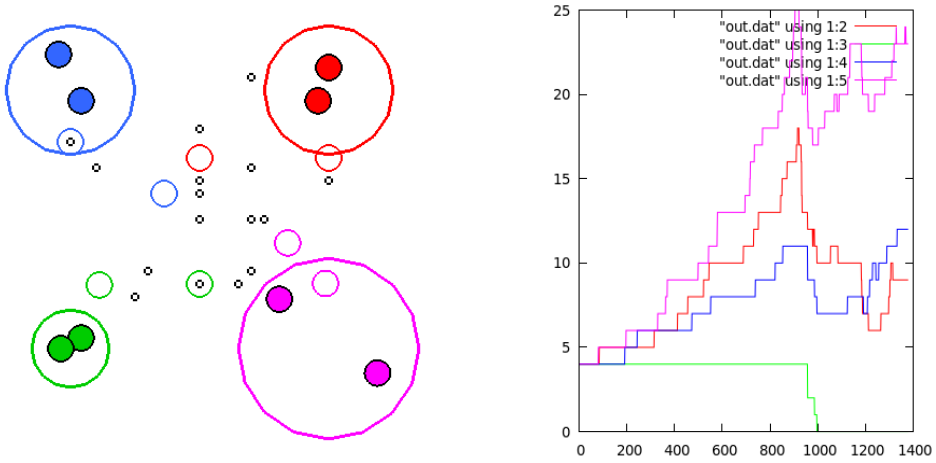

Figure 1 : exemple d'un état de la simulation (gauche) et visualisation de l'évolution du nombre total de fourmis au cours du temps (droite) avec gnuplot (avec correspondance des couleurs).

1. Introduction

Le but de ce projet est de réaliser une simulation de l'évolution d'un ensemble de ***fourmilières*** qui se partagent des éléments de ***nourriture*** dispersés autour des fourmilières. La simulation s'effectue dans un espace continu 2D et gère le déplacement des deux types de fourmis : le ***garde*** reste dans la fourmilière pour protéger la réserve de nourriture tandis que ***l'ouvrière*** va chercher un élément de nourriture et le ramène à la fourmilière à laquelle elle est rattachée. Des règles de comportement à respecter (transport de nourriture, relation vis à vis des fourmis de fourmilières différentes) offrent un cadre dans lequel vous pouvez intervenir à deux niveaux : statégique (niveau fourmilière et choix initial de nourriture à rapporter) et tactique (modification du choix de nourriture selon le contexte). L'évolution du nombre total de fourmi pour chaque fourmilière est enregistrée pour analyser l'impact de votre approche.

Il n'y a aucune dépendance 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 3 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 (px).

On utilisera la *double précision* pour les données et les calculs en virgule flottante.

2. *Monde 2D* et composants de la simulation

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**. Par souci de simplification de mise en œuvre des comportements simulés, chaque élément de la simulation est modélisé par un **cercle** dont rayon dépend du type d'entité. La position (x,y) des éléments de nourriture et des fourmilières est limitée à un domaine **[-DMAX, DMAX]** ; seul le déplacement des ouvrières est autorisé en dehors de cet espace. Il y a au plus **MAX_FOURMILIERE**.

Les Figures 1 et 2 montrent un exemple de visualisation des éléments d'une simulation à un instant donné. La mise à jour est effectuée chaque **DELTA_T** (section 3.2).

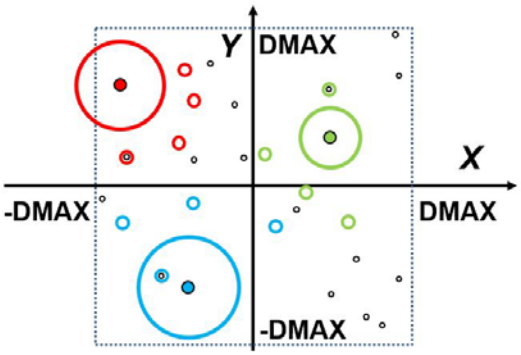

Fig 2 : Système de coordonnées ***Monde*** et exemple d'entités manipulées : **Fourmiliere** (cercle épais de couleur différentes), **Garde** (petits cercles pleins de même couleur que leur Fourmilière avec un bord noir), **Ouvrière** (petits cercles vides de même couleur que leur Fourmilière), élément de **Nourriture** (cercles vides avec un bord noir).

2.1 Les éléments de la simulation

2.1.1 Fourmilière

Une Fourmilière est un espace circulaire dont le centre doit être dans le domaine **[-DMAX, DMAX].** Son *rayon* est une fonction du nombre total **nbF** de fourmis **Garde** (**nbG**) et **Ouvrières** (**nbO**) qui lui sont associées **nbF = nbG + nbO** et de la nourriture totale **total_food** qu'elle contient :

Rayon_fourmilière = (**1** + sqrt(**nbF**) + sqrt(**total_food**))***RAYON_FOURMI** **(Equ. 1)**

De plus deux Fourmilières ne doivent pas se superposer. De ce fait le rayon obtenu ci-dessus est éventuellement diminué en calculant le rayon produisant le cas tangent et en lui soustrayant EPSIL_ZERO (Fig 2a).

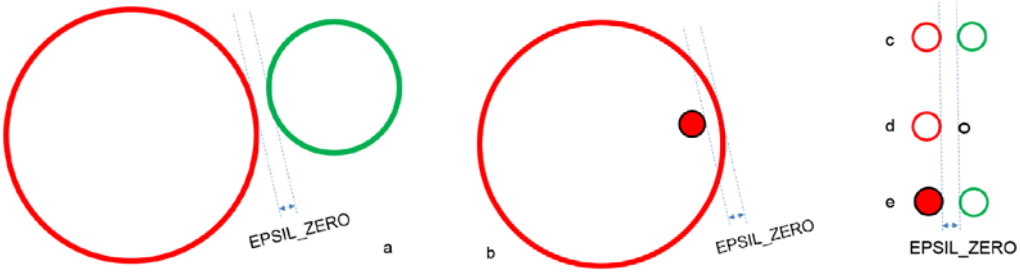

Fig 3 : prise en compte de la tolérance EPSIL_ZERO ; (a) le plus grand rayon *accepté* est celui produisant le cas tangent auquel on enlève EPSIL_ZERO ; (b) la position maximale *acceptée* pour un Garde est celle produisant le cas tangent avec le bord de la Fourmilière à laquelle on enlève EPSIL_ZERO ; on considère qu'il y a *contact* entre deux Ouvrières (c), une Ouvrière et un élément de Nourriture (d), un Garde et une Ouvrière (e) dès que la distance séparant le bord externe des deux entités est strictement inférieure à EPSIL_ZERO.

La fourmilière est responsable du stockage du total de la Nourriture **total_food** au centre de la Fourmilière et de sa consommation en fonction du nombre total de fourmis **nbF** : une quantité de nourriture de (**nbF*FEED_RATE**) est ainsi consommée à chaque mise à jour. Après cette consommation, si **total_food** est strictement inférieur à VAL_FOOD alors on arrondit **total_food** à **0**.

La probabilité de naissance d'une nouvelle fourmi est de **total_food*BIRTH_RATE** (cf section 2.3). La Fourmilière décide de son rôle de **Garde** ou d'**Ouvrière** à sa naissance. La position d'une fourmi à sa naissance est le centre de la Fourmilière. Les fourmis ont le droit de se superposer. Une fourmi vit **BUG_LIFE** cycles de mise à jour et disparait de la simulation lorsqu'elle atteint cet age maximum.

Incompatibilité des fourmis de fourmilières distinctes :
La superposition partielle Ouvrière-Ouvrière ou Ouvrière-Garde de deux Fourmilières distinctes cause la mort prématurée des deux fourmis.

Destruction de la fourmilière : une fourmilière dont le nombre de fourmis **nbF** est nul et dont la nourriture **total_food** est aussi nulle, doit être détruite et disparaître de la simulation.

La Fourmilière est représentée par un cercle vide à bord épais d'une couleur distincte des autres Fourmilières sauf Le noir qui est réservé pour les éléments de Nourriture.

2.1.2 Fourmi de type Garde

Une Fourmi de type **Garde** occupe un espace circulaire de rayon **RAYON_FOURMI** et doit rester en permanence et entièrement à l'intérieur de sa Fourmilière (Fig 3b). L'unique fonction d'un Garde est d'intercepter une (seule) fourmi Ouvrière provenant d'autres Fourmilières dès qu'elle entre dans la Fourmilière (section 2.2). Il y a interception dès qu'elles entrent en contact comme indiqué à la Figure 3e. Plusieurs Gardes peuvent se superposer dans la Fourmilière.

Un Garde est représenté par un cercle plein de même couleur que sa Fourmilière, avec un bord noir (Fig 2).

2.1.3 Fourmi de type Ouvrière

Une Fourmi de type **Ouvrière** occupe un espace circulaire de rayon **RAYON_FOURMI** et est chargée d'aller chercher un (seul) élément de nourriture à la fois et de le rapporter au centre de la Fourmilière pour son stockage. Une seule Ouvrière suffit pour rapporter un élément de Nourriture.

En aucun cas une Ouvrière ne cherche délibérément à atteindre une Ouvrière d'une autre Fourmilière. Il peut arriver par contre que, en chemin vers leur choix d'élément de Nourriture, deux Ouvrières de Fourmilières différentes entrent en contact (Fig 3c), ce qui cause leur mort prématurée (cf section 2.2 concernant le déplacement et l'évitement de collision).

L'acquisition d'un élément de nourriture est validée lorsque l'Ouvrière entre en contact avec cet élément (Fig 3d). L'élément de Nourriture prend alors la même position que l'Ouvrière pendant le trajet de retour.

Le dépôt de l'élément de Nourriture au centre de la Fourmilière est validé lorsque l'Ouvrière se trouve à une distance strictement inférieure à **RAYON_FOURMI** du centre. L'élément de Nourriture n'existe plus en tant qu'entité indépendante ; sa valeur est ajoutée à **total_food** qui exprime le total de Nourriture disponible.

Algorithme du « Bon Choix » de l'élément de Nourriture à aller chercher :
Pour pouvoir choisir, il faut d'abord connaître ce qui est à disposition. Dans cette simulation, on suppose qu'une Ouvrière connait la position de tous les éléments de nourritures pour décider lequel elle va chercher et ramener. Les Ouvrières d'une Fourmilière connaissent aussi la position des autres Fourmilières ainsi que des autres Ouvrières et Gardes. Ces informations vous permettent de définir un algorithme de choix efficace pour une Fourmilière donnée.

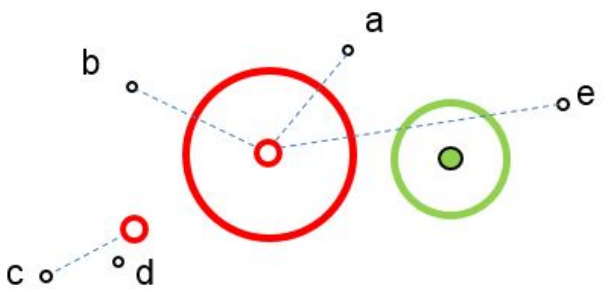

Fig 4 : choix d'élément de nourriture pour une Ouvrière au centre de la Fourmilière rouge. La cible **b** est préférable même si elle est un peu plus éloignée que la cible **a** car il y a moins de concurrence. S'il n'y a que la cible **e**, l'algorithme doit proposer un chemin qui ne passe pas à l'intérieur de la Fourmilière verte. Pour l'Ouvrière déjà en route, la toute nouvelle cible d devrait être choisie à la place de la cible initiale **c**.

En effet, le critère de la plus courte distance est un élément utile mais non suffisant pour effectuer « un bon choix » car l'élément le plus proche est peut être encore plus proche d'une autre Fourmilière et peut-être sera-t-il pris avant que l'Ouvrière y arrive (comparer Fig 4 cibles **a** et **b**). Le risque de mort prématurée est très grand dans ce cas (section 2.1.1). Enfin, traverser une autre Fourmilière que la sienne pour atteindre un élément de Nourriture (Fig 4 cible **e**) est *interdit* car il expose l'Ouvrière à une interception par un Garde (section 2.2). Dans ce cas l'algorithme peut définir une trajectoire qui contourne l'autre Fourmilière.

Pour faciliter la mise au point de votre *algorithme du « bon choix »* on suppose aussi que toutes les Ouvrières d'une même Fourmilière savent qui va chercher quel élément de nourriture. Par contre, les Ouvrières ne connaissent pas la destination des Ouvrières des autres Fourmilières.

L'algorithme du Bon Choix doit être activé dans les cas suivants :

- Elle vient de déposer un élément de Nourriture au centre de sa Fourmilière
- un élément de Nourriture plus proche que son choix initial vient d'apparaître (Fig 4 cible d et c)
- Son choix initial d'élément de nourriture vient d'être pris par une autre Ouvrière
- Elle vient d'acquérir son élément de nourriture et désire évaluer un chemin de retour.

On peut même imaginer qu'une Ouvrière d'une Fourmilière envisage d'aller chercher une quantité **VAL_FOOD** de nourriture stockée dans une autre Fourmilière ; cela présente un risque important de mort prématurée de l'Ouvrière mais qui peut se révéler positif sur le long terme pour la Fourmilière. Pour cette raison, cette stratégie ne peut être envisagée qu'en complément de l'action d'aller chercher de la nourriture en dehors des Fourmilières.

Une Ouvrière est représentée par un cercle vide de même couleur que sa Fourmilière (Fig 2).

2.1.4 Nourriture

Un élément de **Nourriture** est un espace circulaire de *rayon* **RAYON_FOOD** et sa valeur nutritive est **VAL_FOOD**. La création d'un nouvel élément de Nourriture est effectuée à chaque mise à jour avec une probabilité de **FOOD_RATE** (cf section 2.3). S'il y a création, le centre doit être créé *aléatoirement* (cf section 2.3) dans le domaine **[-DMAX, DMAX]** et ne PAS se superposer à une Fourmilière, une Ouvrière ou un élément de Nourriture déjà existant.

Seule une Ouvrière peut rapporter un élément de Nourriture à la Fourmilière (section 2.1.3).

En cas de mort prématurée d'une Ouvrière qui avait acquis un élément de Nourriture, celui-ci reste à la dernière position de cette Ouvrière jusqu'à ce qu'une autre vienne le prendre.

Un élément de Nourriture est représenté par un cercle vide noir (Fig 2).

2.2 Déplacement des fourmis

Par souci de simplification, une fourmi n'est pas orientée. Elle peut bouger dans n'importe quelle direction à chaque mise à jour avec une vitesse comprise entre 0 et **BUG_SPEED**, ce qui fait qu'elle peut parcourir au maximum une distance de **BUG_SPEED*DELTA_T** à chaque fois. Pour ce projet, nous garantissons que cette distance maximale est inférieure ou égale à **RAYON_FOURMI** (Fig 5). Grâce à cette propriété nous pouvons accepter l'approximation de tester la superposition des Ouvrières de Fourmilière différentes en examinant seulement la position finale du déplacement (= on ignore les positions intermédiaires).

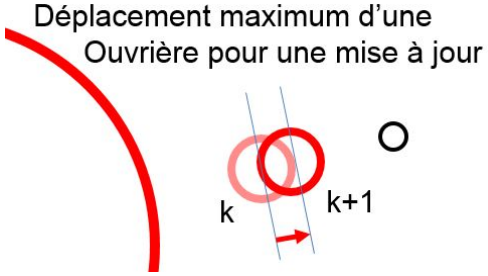

Fig 5 : Une Ouvrière se déplace en direction de son choix d'élément de Nourriture à rapporter. Chaque mise à jour autorise une distance maximum de déplacement de BUG_SPEED*DELTA_T

Cas général du déplacement sans obstacle entre l'Ouvrière et son but:

Connaissant la position courante de l'Ouvrière et celle de son choix d'élément de Nourriture à rapporter, il suffit de construire le vecteur **V** reliant ces deux points. Si sa norme est supérieure à **RAYON_FOURMI** l'Ouvrière va se déplacer d'un vecteur $\mathbf{V_d = BUG\_SPEED*DELTA\_T*V_n}$ **(Equ. 2)**

où $\mathbf{V_n}$ est le vecteur **V** normalisé.

Cas d'un obstacle fixe (Fourmilière) entre l'Ouvrière et son but ou son chemin de retour (section 2.1.3):

Ce cas doit être traité au moment du « bon choix » pour éviter de passer à travers d'une autre Fourmilière.

Evitement d'un obstacle mobile (Ouvrière d'une autre Fourmilière) entre l'Ouvrière et son but:

Une Ouvrière est autorisée à ajuster son déplacement vers un élément de Nourriture en fonction de la présence d'Ouvrières d'une autre Fourmilière dont on connait la position courante (mais pas le but). Ce cas n'est pas trivial à traiter c'est pourquoi il n'est pas explicitement requis de le prendre en compte. Voici quelques questions à examiner qui peuvent vous servir pour cet ajustement :

- L'Ouvrière a-t-elle le temps de prendre l'élément de Nourriture avant que l'autre Ouvrière la rejoigne ?
- L'autre Ouvrière vise-t-elle le même élément de Nourriture ou un autre ?

2.3 Mise en œuvre des probabilités utilisées dans la simulation

Plusieurs actions de la simulation sont liées à une probabilité, c'est-à-dire qu'il n'est pas certain qu'elles soient réalisées à chaque mise à jour (naissance de fourmi, création d'élément de Nourriture).

Leur mise en œuvre en C est très simple grâce à la fonction **rand()** qui renvoie un nombre aléatoire (entier positif) entre **0** et **RAND_MAX** (cette constante est 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 une mise à jour, 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)**

Pour obtenir une valeur aléatoire **val** dans un intervalle [MIN, MAX] différent de [0, RAND_MAX], il suffit de renormaliser la valeur comme suit : **val = (rand()/RAND_MAX)*(MAX – MIN) + MIN** **(Equ. 4)**

3 Actions à réaliser

3.1 Gestion du contexte de la simulation (détails en sections 4 et 5)

Il doit être possible de réaliser les actions suivantes par l'intermédiaire de l'interface graphique (section 5) lorsque la simulation est en mode Pause :

- **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. (Détails section suivante).
- **Ecriture** d'un fichier décrivant l'état de la simulation à l'instant courant (section 4)
- **Activer l'enregistrement** du nombre total de fourmis par fourmilière (section 5.1)
- **Lancement** de la simulation en continu ou seulement pour une unité de temps DELTA_T

3.1.1 Détection d'erreur à la lecture d'un fichier

L'opération de lecture doit vérifier quelques erreurs de format dans le fichier lu et les signaler en utilisant les fonctions mises à disposition dans le module **error**.

Le rendu1 détectera seulement des erreurs simples sur le nombre d'éléments et sur la validité partielle des positions (cf section rendu1 et fichier tests publics).

Le rendu2 détectera les conditions de non-superposition imposées sur les éléments de la simulation (Fourmilière, Nourriture) doivent être vérifiées après la **lecture** complète du fichier (cf section rendu2 et fichiers de tests publics).

Si la lecture d'un fichier produit une erreur, il faut appeler la fonction fournie dans le module **error**. Le message indique, si nécessaire, les types d'éléments fautifs et leur numéro entre **0** et **nb_élément-1**. De plus, l'ensemble du contexte de la simulation doit être détruit (= ré-initialiser les structures de données et libérer la mémoire si nécessaire).

3.2 Mise à jour de la simulation

Une mise à jour de la simulation consiste à calculer l'évolution de l'état de tous ses constituants après un intervalle de temps DELTA_T depuis la mise à jour précédente.

Pour ce projet, la mise à jour est *simplifiée* car on réagit immédiatement au déplacement d'une Ouvrière sans attendre que toutes les autres Ouvrières soient aussi mises à jour.

On adoptera la structure suivante de mise à jour (= pour un seul DELTA_T):

1) Création aléatoire de nouveaux éléments de Nourriture (sections 2.14 et 2.3)

2) Pour chaque Fourmilière
- Naissance éventuelle de Fourmi
- Consommation de Nourriture et arrondi à zéro si **total_food** < VAL_FOOD
- Mise à jour du rayon de la Fourmilière

- Pour chaque Fourmi Ouvrière
  - Incrémentation de l'age et mort naturelle celui-ci atteint BUG_LIFE
  - Eventuelle mise à jour du « bon choix » de l'élément de Nourriture à rapporter,
  - Selon l'avancement de la tâche de l'Ouvrière : déplacement, acquisition d'un élément de nourriture, transfert d'un élément de Nourriture à la Fourmillière.
  - En cas de superposition avec une Ouvrière ou un Garde d'une autre Fourmillière
    - Mort prématurée des deux entités.

- Pour chaque Fourmi Garde
  - Incrémentation de l'age et mort naturelle celui-ci atteint BUG_LIFE
  - Eventuel déplacement : soit en direction d'une Ouvrière d'une autre Fourmilière dès l'entrée de son centre dans la Fourmilière ou alors pour s'assurer que le Garde est entièrement dans la Fourmilière
  - En cas de superposition avec une Ouvrière d'une autre Fourmillière
    - Mort prématurée des deux entités.

- Destruction éventuelle de la Fourmilière si nbF est nul et total_food est nul

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

3.3 Tâche de bas-niveau

Plusieurs actions du projet ont besoin de tester la distance entre deux cercles pour prendre certaines décisions : ajustement de rayon de Fourmilière, correction de position de Garde, décision de contact entre certaines entités. Chaque cas est décrit précisément dans la section 2 et la Figure 3.

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 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 une fourmilière, une fourmi ou un élément de nourriture. Il ne connait que des points, des vecteurs et des cercles dans le plan et il doit offrir des fonctions génériques qui sont utiles pour réaliser les opérations et tests de plus haut niveau.

Les différents cas identifiés à la Figure 3 doivent se traduire par la mise à disposition d'une ou de plusieurs fonctions génériques. Par exemple les trois cas de ***contact au sens de la Fig 3 c,d,e*** entre 2 cercles de rayons r1 et r2 et distants de D, doivent être traités à l'aide d'une unique fonction qui renvoie VRAI si :

D - (r1 + r2) < EPSIL_ZERO (Equ. 5) à utiliser pour la simulation

L'écriture puis la lecture d'un fichier formaté peut conduire à une perte de précision ; c'est pourquoi les tests de superposition effectués lors de la lecture doivent être un peu moins stricts que l'équation 5. On utilise :

D - (r1 + r2) ≤ 0 (Equ. 6) à utiliser pour les tests en lecture

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

Pour tester les 3 rendus de votre projet nous exécuterons plusieurs scénarios de complexité progressive dont une partie sera publique et disponible dans des fichiers.

Votre programme doit donc être capable de lire de tels fichiers. Il doit aussi pouvoir mémoriser la configuration courante de la simulation. 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 Fourmiliere** **x y nbO nbG total_food rayon** **age posx posy butx buty bool_nourriture** **…** **FIN_LISTE** **age0 x0 y0 age1 x1 y1 …** **… agen xn yn** **FIN_LISTE** **…** **FIN_LISTE** **nb Nourriture** **x0 y0 x1 y1 …** **… xn yn** **FIN_LISTE**	Pour chaque fourmilière on indique les paramètres généraux, puis la liste des Ouvrières (une par ligne), puis la liste des Gardes (un nombre entier de Gardes par ligne) Pour la Nourriture, on indique la liste des éléments de nourriture qui ne sont pas encore pris par une Ouvrière (un nombre entier d'élément de nourriture par ligne).

5. Description de l'interface utilisateur (GUI)

Comme dans les séries 17-19, 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:

- **Exit** : ferme les fenêtres et fichiers et quitte le programme
- **Open File** : remplace la simulation par le contenu du fichier dont le nom est fourni (sections 2 et 3.1).
- **Save File** : mémorise l'état actuel de la simulation dans le fichier dont le nom est fourni
- **Start** : bouton pour commencer/stopper la simulation
- **Step** (lorsque la simulation est stoppée) : calcule seulement un pas
- **Record** : checkbox permettant d'indiquer si on désire enregistrer le nombre total de fourmis pendant la simulation (section 5.1)
- **Manual-Food** : checkbox ou radio_button permettant d'interrompre la création automatique de nourriture. A la place on peut ajouter la nourriture manuellement (cf 6.2)

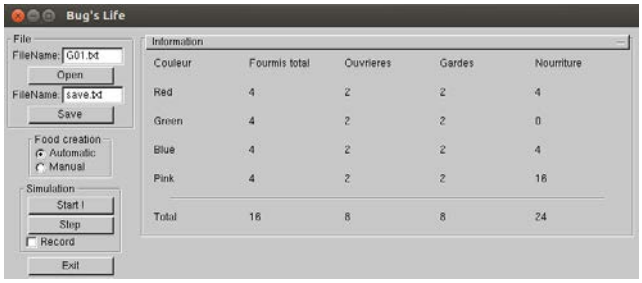

Fig 6 : Interface graphique (GUI) ; exemple produisant la Fig1

le programme tiendra à jour et affichera les informations suivantes pour chaque Fourmilière :

- Nb de fourmis total
- Nb d'Ouvrières
- Nb de Gardes
- Quantité de Nourriture totale **total_food**

5.1 Enregistrement du nombre de fourmis total 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 d'un compteur entier** (indice de mise à jour) suivi par le nombre total de fourmis de toutes les fourmilières (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 ces grandeurs au cours du temps. L'exemple de la figure 1 (droite) est obtenu pour 4 Fourmilières avec la commande gnuplot suivante:

plot "out.dat" using 1:2 with lines,"out.dat" using 1:3 with lines, "out.dat" using 1:4 with lines,"out.dat", using 1:5 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. 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 :

Manual-Food: lorsque l'option Manual-Food est activée, la nourriture n'est plus ajoutée automatiquement. A la place, et seulement quand cette option est active, on peut ajouter un seul élément de nourriture à la fois à l'endroit cliqué avec le bouton droit *Si cet endroit respecte la règle de création de la nourriture* (section 2.1.4).

7. Architecture logicielle

7.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 7 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 6): 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 Ouvrières, Gardes, Fourmilières et éléments de Nourriture (Principe de Ré-utilisation).

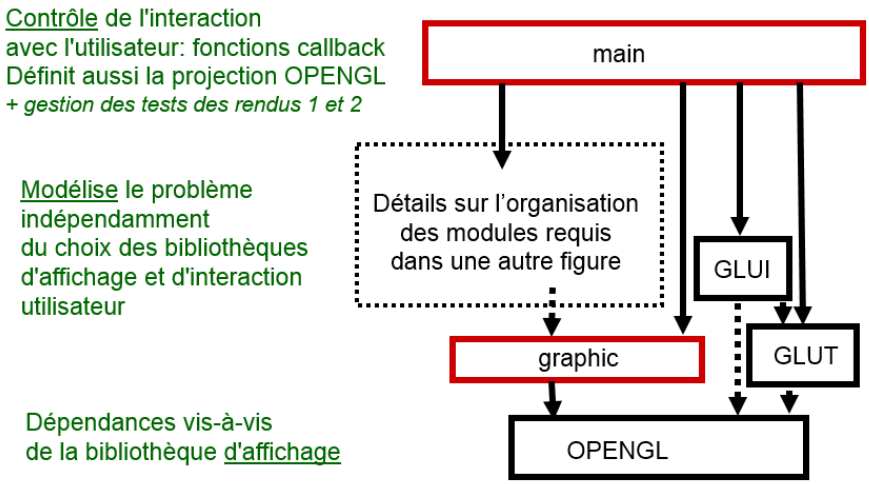

Fig 7 : Architecture logicielle minimale à respecter

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. Nous adoptons une approche par **type opaque** où un module exporte seulement des fonctions et/ou des symboles constants ; en particulier l'interface d'un module opaque n'exporte PAS la description des types et n'expose aucune variable globale. La Figure 7 présente l'organisation minimale à adopter en termes d'organisation des modules :

- ***Au plus haut niveau***, le module **modele** gère l'évolution de la simulation et centralise les opérations de dessin, de lecture et d'écriture de fichier ainsi que l'édition des éléments. Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi, en vertu du principe d'abstraction le module **modele** *est le seul module* dont on peut appeler des fonctions en dehors du Modèle (*traduction* : si le sous-système de Contrôle veut modifier l'état de la simulation cela doit se faire par un appel d'une fonction de **modele.h** comme le montre l'unique flèche de dépendance entre le module **main** et le module **modele**).

- ***Niveau(x) intermédiaire(s)***: **fourmiliere, fourmi, nourriture.** 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): c'est le seul module qui n'est pas opaque. En effet, les modules précédents ayant besoin de travailler avec **des éléments 2D tels que des points, des vecteurs**, **des cercles,** il est demandé de créer un module **utilitaire** de bas niveau mettant à disposition un ou plusieurs **types concrets** pour réaliser des opérations sur ces éléments (ex : opérations vectorielles, calcul de la position d'un point, etc…). IMPORTANT: un tel module de bas niveau doit être général ; il travaille seulement sur les types élémentaires tels que point, vecteur ou segment et il ignore les notions de fourmilière, fourmi, nourriture utilisés aux niveaux supérieurs du modèle. En particulier, comme le montre la Fig8, il n'y aucune dépendance allant du module **utilitaire** vers les modules fourmiliere, fourmi, nourriture ; les dépendances sont seulement en sens inverse.

- 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 (rendu1) et de la vérification des intersection/superpositions interdites (rendu2).

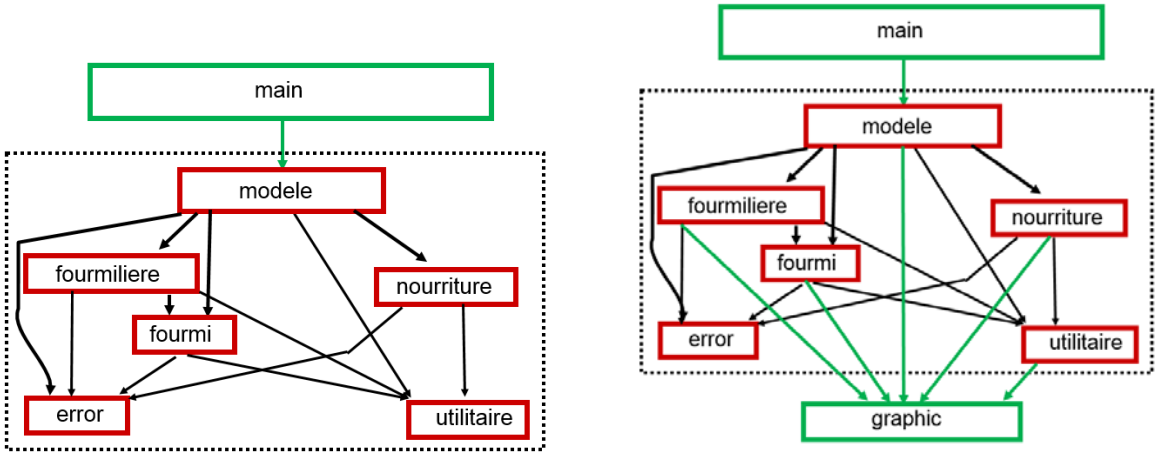

Figure 8 : architecture minimale pour le rendu1 (a), pour le rendu2 (b). Le module **utilitaire** est le seul module responsable de types concrets dont la description complète est visible dans **utilitaire.h**. La figure montre les seules dépendances externes autorisées en vert.

Remarque : il est autorisé d'ajouter des dépendances supplémentaires entre les modules **fourmiliere, fourmi et nourriture** si vous le jugez nécessaire. Il suffira de justifier ces dépendances dans le rapport du rendu2.

8. Méthode d'évaluation des rendus avec les fichiers de test

8.1 Construction/ré-écriture partielle du programme

Le projet intègre les connaissances du semestre d'automne et les compléments vus pendant les sept premières semaines du semestre de printemps. Pour ces raisons, le programme devra être partiellement ré-écrit entre les rendus car on ne dispose pas de toutes les connaissances nécessaires dès le début de semestre.

Résumé des aspects importants:
Rendu1 : test du format des fichiers (3.1.1). PAS de graphique.
Rendu2 : vérification après lecture (3.1.1), initialisation de la fenêtre graphique et du GUI (section 5), Analyse
Rendu final : simulation, enregistrement de l'évolution des nombres de fourmis, performances

A chaque rendu, le code source sera évalué selon les conventions de programmation et le respect des types **opaques** (section 7.2). Le module **utilitaire** doit aussi offrir des **types concrets** qui sont utilisés dans la définition des structures FOURMILIERE, FOURMI et NOURRITURE.

Détermination du *mode de test* du programme :
L'analyse des arguments transmis sur la ligne de commande (détails en 8.3) permet au module principal **main** de déterminer le ***mode de test*** du programme et le ***nom du fichier*** de test. Ces deux informations doivent être transmises à la fonction de lecture de fichier **`modele_lecture()`** du module **modele**. Les sections suivantes détaillent comment cette fonction doit travailler pour chaque rendu. On précise également les aspects techniques qui dépendent de l'avancement du cours.

8.2 Description des rendus

La répartition des points est détaillée dans le barème en Annexe D.

8.2.1 Rendu1 / test des erreurs de format de fichier simulation:

Ce mode de test est noté **Error**. L'architecture à adopter est donnée par la figure 8a. La fonction **`modele_lecture()`** travaille au niveau d'abstraction supérieur du modèle ; elle doit déléguer la mémorisation des entités et la détection des erreurs au modules **fourmiliere, fourmi et nourriture**. La liste des erreurs à détecter pour le rendu1 est visible dans **error.h** car il faut appeler les fonctions du module **error.c** pour afficher le message d'erreur standardisé. Il s'agit d'erreurs simples sur le nombre d'éléments (trop ou pas assez de données dans le fichier, validité partielle des positions) et sur la superposition des éléments de la simulation.

L'erreur doit être détectée dès que l'information est lue. On quitte la fonction de lecture dès la première erreur trouvée. S'il n'y a pas d'erreur dans un fichier vous devez appeler la fonction **`error_success()`**. Dans ce mode, le programme s'arrête juste après l'exécution de la fonction **`modele_lecture()`**.

Au stade du cours au moment du rendu1, l'allocation dynamique de mémoire n'est pas encore traitée ; c'est pourquoi il est autorisé de mémoriser les entités dans des tableaux de structure de taille constante. L'usage d'un tableau de *pointeurs de structure* est compatible avec les types opaques. Nous garantissons que le nombre d'entités ne dépassera pas la taille de **MAX_RENDU1**. Prenons un exemple, dans le module **nourriture** un tableau de structure NOURRITURE est défini en dehors de toute fonction avec le mot clef **static** :

```
static NOURRITURE tab_n[MAX_RENDU1] ;
```

C'est à vous de définir la structure NOURRITURE. Cette approche devra être revue pour le rendu 2 avec l'allocation dynamique de mémoire. Nous testerons 9 fichiers publics et 3 fichiers supplémentaires.

8.2.2 Rendu2 / test de la vérification des superposition interdites

Ce mode de test est noté **Verification**. L'architecture à adopter est donnée par les figures 7 et 8b. Les vérifications du fichier (8.2.1) doivent continuer à être faites. Cette fois-ci l'allocation dynamique de mémoire doit être utilisée pour créér le nombre exact d'entités dans les modules **fourmiliere, fourmi et nourriture**. Dans ce mode la vérification des superposition interdites (fourmilieres ou fourmis de fourmilières différentes) doit être faite, immédiatement après la lecture du fichier, avec la fonction suivante juste:

```
void modele_verification_rendu2(void)
```

Si un problème est détecté par cette fonction, on doit appeler la fonction dédiée du module **error** pour l'affichage du message d'erreur. Il faut transmettre à la fonction les deux types d'éléments fautifs et leur numéro entre **0** et **nb_élément-1**. On arrête la vérification dès la première erreur trouvée. Dans ce mode, en l'absence d'erreur on quitte le programme après l'appel de la fonction **modele_verification_rendu2**. La section 3.3 précise la manière de tester la superposition pour les valeurs lues dans un fichier formaté.

Nous testerons 2 fichiers publics et 2 fichiers supplémentaires. Pour réduire les différences entre le programme de démo et le vôtre, nos fichiers de test auront au maximum une erreur et nous accepterons les 2 variantes de messages possibles dus à l'ordre des éléments fautifs dans le message.

8.2.3 Rendu2 / test de l'initialisation du GUI et de la fenêtre de simulation

Ce mode de test est noté **Graphic**. L'architecture à adopter est la même que pour 8.2.2. Les vérifications du fichier (8.2.1 et 8.2.2) doivent continuer à être faites ; la seule différence est que, si une erreur est détectée à la lecture d'un fichier ou ensuite à la vérification, l'ensemble du contexte de la simulation est détruit mais l'exécution se poursuit. Dans tous les cas, le contrôle est passé à GLUI et initialise la fenêtre graphique et l'interface utilisateur comme dans les TP 17-19.

Nous vérifierons manuellement que l'affichage de l'état initial s'effectue correctement dans les 2 fenêtres avec 2 fichiers publics. La lecture de plusieurs fichiers sera aussi faite avec le bouton **Open file** dans un ordre quelconque sans redémarrer le programme. De plus, nous testerons la sauvegarde dans un nouveau fichier et la lecture de ce nouveau fichier pour l'un des fichiers publics et supplémentaires.

8.2.4 Rendu2 / Rapport de la phase d'Analyse (max 1 page)

L'architecture prévue pour le rendu final doit être décrite dans le rapport de la phase d'analyse :

- **Justifier l'architecture du rendu final :** conservez-vous l'architecture de la figure 8b ? Si vous ajoutez des relations supplémentaires entre les modules **fourmiliere, fourmi et nourriture**, justifiez chaque nouvelle relation en indiquant la ou les fonctions d'un module qui appelle la ou les fonctions de l'autre module. Justifiez aussi tout module supplémentaire que vous jugez utile.
- **Décrire les structures de données pour le rendu final** (indiquer les changements par rapport au rendu1). Tous les éléments peuvent augmenter/diminuer au cours d'une session; comment cela est-il géré au niveau des structures de données ?
- **Donner la liste des fonctions du module modele qui sont appelées dans le module main** (à quoi servent-elles?). Ces fonctions doivent exister dans le code source du rendu2 à l'état de stub = chaque fonction affiche seulement son nom dans le terminal avec un printf.

8.2.5 Rendu final / interaction

Le mode de test est noté **Final**. Nous testerons manuellement 4 fichiers publics et 2 fichiers supplémentaires. Leur complexité sera progressive pour tester les performances de votre programme pour les fonctionnalités demandées.

8.2.6 Rendu final /Rapport final

Entre 2 (min) et 4 pages maximum (SVP : ne gaspillez pas de papier avec des pages de titre et de table des matières). Le rapport est écrit en français ou en anglais ; une orthographe ou une grammaire défaillante peut induire les correcteurs en erreur. Le rapport contient :

Mise à jour de l'architecture logicielle et de la description de l'implémentation:

- décrivez les compléments ou remises en question depuis le rendu précédent. Documentez la nature des modifications si elles proviennent du rendu public. Fournissez le dessin de l'architecture logicielle finale seulement si elle est différente de celle de la figure 8b.
- précisez votre approche pour la gestion efficace des données. Donnez une estimation du coût calcul et mémoire lors du calcul de la mise à jour d'un pas de la simulation. Indiquez seulement ***le terme dominant*** en fonction des paramètres, par ex : $O(nbO^2 * nbG * (total\_food))$; ne PAS remplacer les paramètres par N.
- Illustration avec des images sur un exemple comportant les 4 types d'entités : montrer au moins 4 images de l'évolution de la simulation. Vous pouvez utiliser Alt-PrtSc pour récupérer une image de l'exécution de votre programme.

Méthodologie et conclusion : comment avez-vous organisé votre travail à plusieurs, indiquer la personne responsable de chaque module et comment vous avez organisé le travail au sein du groupe (par quels modules avez-vous commencé, comment les avez-vous testés, comment le feriez-vous maintenant avec le recul, quel

était le bug le plus fréquent, pourquoi ? et celui qui vous a posé le plus de problème et comment a-t-il été résolu, ...). Pour conclure fournissez une brève auto-évaluation de votre travail et de l'environnement mis à votre disposition (points forts, points faibles, améliorations possibles, trouvez-vous utile de pouvoir disposer des rendus intermédiaires publics)

8.3 Syntaxe du lancement du programme

Le nom de votre exécutable change selon les rendus (Annexe B dans document séparé). On doit pouvoir lui transmettre deux arguments optionnels au lancement, sur la ligne de commande. La syntaxe est la suivante :

```
./projet.x [Error|Verification|Graphic|Final, nom_fichier]
```

Si **aucun argument** n'est transmis (*à partir du rendu2 seulement*), 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**.

Arguments optionnels à la suite du nom de l'exécutable: le premier argument désigne le mode de test du programme. Lorsqu'on indique l'un des quatre mots clef parmi **Error**, **Verification**, **Graphic** et **Final** il faut ajouter un **nom de fichier** de test comme second argument.
Exemple :

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

La 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 l'un des 4 mots clef prévus. Si **`argv[1]`** est le mot clef **`Error`** alors une fonction de **`modele.c`** doit être appelée pour gérer la lecture du fichier obtenu avec **`argv[2]`**.

ANNEXE A : constantes utilisées dans le projet

Fichier tolerance.h : constantes générales

```
#define  EPSIL_ZERO          1e-3
```

Fichier constantes.h : constantes spécifiques au projet

```
#include "tolerance.h"
#define  DELTA_T             0.25
#define  BUG_SPEED           1          // ancienne valeur 4
#define  BUG_LIFE            1000       // meurt quand atteint cet age
#define  DMAX                20
#define  RAYON_FOURMI        1
#define  RAYON_FOOD          0.25
#define  VAL_FOOD            1
#define  FOOD_RATE           0.2        // ancienne valeur 0.1
#define  FEED_RATE           0.002      // anciennes valeurs 0.01   0.0625
#define  BIRTH_RATE          0.0005     // ancienne valeur 0.01
#define  MAX_RENDU1          5
#define  MAX_FOURMILIERE     10
#define MAX_LINE             120
```