# C++ PoP – Sections Electricité et Microtechnique

Printemps 2022 : *Tchanz* 

La simulation peut-elle atteindre la stabilité ?

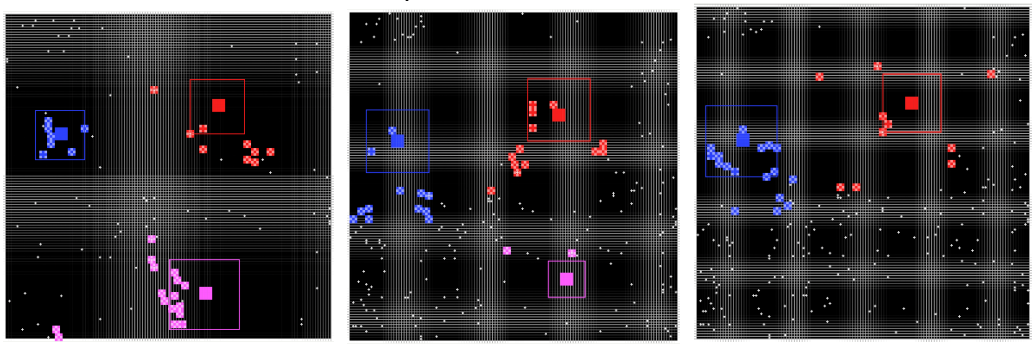


Fig 1 : 3 instants d'une simulation (plusieurs fourmilières ont déjà disparu)

## *1. Introduction*

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

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

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

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

La suite de la donnée indique les ***variables*** en ***italique gras*** et les **constantes** globales en **gras** (la valeur des constantes est visible dans les annexes de ce document ; des fichiers .h seront fournis. On utilisera des entiers pour les positions car ce projet travaille dans un espace 2D discrétisé. Par contre pour des calculs en virgule flottante, on utilisera toujours la *double précision*.

***Pitch du sujet*** *: une ressource aléatoire de nourriture est collectée par des fourmis appartenant à une ou plusieurs fourmilières qui sont organisées selon des tâches et des rôles bien définis. Outre les fourmis Collector en charge de ramener la nourriture, on trouve les fourmis Defensor qui cherchent à limiter le passage dans leur fourmilière, les fourmis Predator qui ont un comportement agressif vis-à-vis des fourmis des autres fourmilières, et finalement l'unique fourmi Generator qui recueille la nourriture et donne naissance aux autres fourmis. La question principale est de savoir si un équilibre peut s'instaurer entre plusieurs fourmilières lorsque la source de nourriture est renouvelée avec une probabilité constante au cours du temps.*

# 2. Modélisation des composantes de la Simulation

**But :** Le but de ce projet est de mettre au point une simulation responsable de la mise à jour de l'état de **nbF** fourmilières (nombre limité à **maxF**). Cela est effectué sous la forme d'une boucle infinie de mise à jour de l'état de ses composantes. Chaque mise à jour correspond à l'écoulement d'***une*** *unité de temps* (nous n'avons pas besoin de manipuler cette quantité). Nous demandons de structurer la mise à jour selon les étapes suivantes :

```
// entrée modifiée : ensemble de la nourriture et ensemble des fourmilières

   Génération aléatoire de nourriture dans l'espace libre

   Pour k de 1 à nbF          // dans le même ordre de celui du fichier de test
     Calcul de sizeF à partir des nombres courants de fourmi
     Ajustement/Déplacement de la fourmilière si nécessaire
     Mise à jour Generator: consommation, naissance (age=0), déplacement
       // avec éventuel passage à vrai de end_of_klan pour Generator

     Si end_of_klan est false
       Pour chaque autre type de fourmi // Collector puis Defensor puis Predator
         Incrémentation de l'age
         Déplacement/Action
           // avec éventuel passage à vrai de end_of_life

   Pour k de 1 à nbF
     Destruction des fourmis dont l'age est bug_life ou si end_of_life est vrai
     Destruction de la fourmilière si le booleen end_of_klan est vrai
```

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

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

## 2.1 L'espace est une grille discrète

Le monde 2D est représenté par une grille à deux indices de taille **g_max** pour les deux indices.

Pour respecter le choix de représentation visuelle du monde (Figure 2) , nous exigeons que les indices de *ligne* **i** et de *colonne* **j** de la grille de la structure de données soient obtenus comme suit à partir des indices d'abscisse **x** et d'ordonnée **y** d'une entité de la simulation :

- ***i* = g_max-1 – *y***
- ***j* = *x***

La Figure 2 montre en hachuré l'espace occupé par la cellule d'abscisse 0 et d'ordonnée 0 : elle couvre le carré ayant pour coin les points (0,0) et (1,1). On y voit aussi l'espace occupé par une fourmilière d'origine ($x_f$, $y_f$) valant (5,3) et un élément de nourriture de coordonnées (14, 14) représenté par un losange.

Les types de fourmis sont montrés dans la Figure 3.

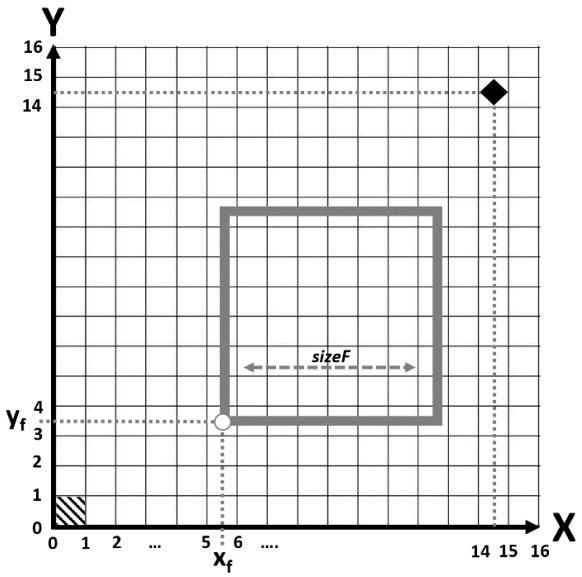



**Figure 2** : monde 2D pour **g_max** valant 16 avec origine en bas à gauche, axe X horizontal orienté positivement vers la droite et axe Y vertical orienté positivement vers le haut. Le carré hachuré illustre l'espace occupé par la cellule d'indices (*x=0,y=0*). Une fourmilière est repérée par les coordonnées ($x_f$, $y_f$) de son coin inférieur gauche. La taille interne d'une fourmilière est un carré de coté ***sizeF***.

Le monde contient ***nbN*** éléments de nourriture et ***nbF*** fourmilières (au maximum **maxF**). La position ($x_f$,$y_f$) d'une fourmilière est celle de son *coin inférieur gauche* (Fig2). Son espace est un carré dont les 4 coins doivent être dans **[0, g_max-1].** Sa bordure occupe une largeur d'une cellule de la grille du monde 2D (cf section 2.2.1 pour le calcul de sa taille).

### 2.1.1 Position et taille des Fourmis

La position (x,y) d'une fourmi est celui du *centre* d'un carré (cercle blanc dans Fig 3). L'annexe A donne la valeur des constantes impaires des cotés du carré : **sizeG** pour la *fourmi Generator*, **sizeC** pour les fourmis *Collector*, **sizeD** pour les fourmis *Defensor* et **sizeP** pour les fourmis *Predator*.

## 2.2 Les Fourmilières

Une fourmilière contient une seule fourmi Generator et peut contenir un nombre variable de fourmis de différents types: ***nbC*** fourmis Collector, ***nbD*** fourmis Defensor et ***nbP*** fourmis Predator. Le nombre total de fourmis d'une fourmilière ***nbT*** est donné par: ***nbT*** = 1 ***+ nbC + nbD + nbP***.

### 2.2.1 Calcul et validation de la taille d'une fourmilière

Si c'est possible, le nombre de cellules **internes** (en plus de sa bordure) ***sizeF*** d'un coté du carré de la fourmilière vaut :

$$sizeF = Partie\_entière(sqrt(4*(sizeG^2 + sizeC^2*nbC + sizeD^2*nbD + sizeP^2*nbP)))$$ (Equ. 1)

La valeur de cette taille désirée ***sizeF*** n'est validée que si elle respecte les contraintes suivantes:

- le carré d'une fourmilière dont la taille du coté ***side*** vaut ***sizeF+2*** ne doit pas sortir du monde 2D.
- deux Fourmilières ne doivent pas se superposer (leur bordures non plus).

Il faut d'abord évaluer s'il y a dépassement/superposition à cause de la valeur désirée ***sizeF***, à partir de l'origine de la fourmilère (coin inférieur gauche). La Fig 3bc montre un cas de succès. En cas d'échec (Fig 3d) il faut aussi tester si on peut agrandir la fourmilière en appliquant ***sizeF*** à partir d'un autre coin, dans l'ordre : *supérieur gauche, supérieur droit et inférieur droit*. On accepte la taille désirée ***sizeF*** dès le premier coin sans dépassement/superposition ; cela implique de mettre à jour la position du coin inférieur gauche (Fig 3e).

En cas d'échec pour les 4 coins, la taille de la fourmilière conserve son ancienne valeur[1] ; on pose que la fourmilière est dans un état **FREE** si sa taille peut augmenter et, sinon, elle est dans un état **CONSTRAINED**. Cette information est utilisée pour définir le comportement de certains types de fourmis.

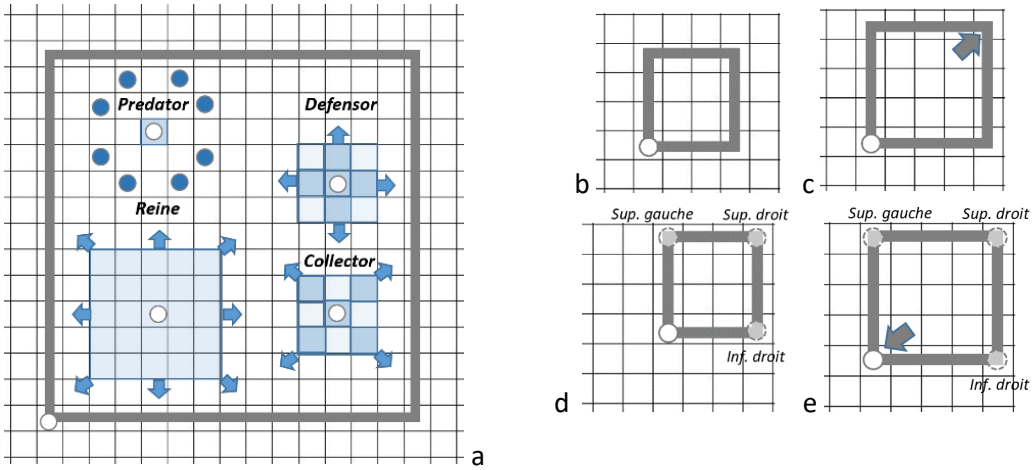



**Fig 3** : (a)Tailles et types de déplacement des fourmis ; les fourmis Collector et Defensor occupent le même espace carré, le motif permet de les distinguer; les cercles bleus indiquent les possibilités de prochaines positions pour le Predator ; (b et c) cas où une fourmilière peut grandir d'une unité ; (d) cas où elle ne peut pas grandir à cause du bord du monde ; (e) solution: grandir à partir du coin supérieur droit ce qui pousse son origine vers le bas et à gauche.

[1] On ne cherche pas à évaluer les tailles intermédiaires entre la taille actuelle et la taille désirée

### 2.2.2 Destruction d'une fourmilière (booléen end_of_klan)

Si la fourmi Generator n'existe plus (section 2.3.2), le booléen ***end_of_klan*** devient vrai et la fourmilière doit être détruite en fin de mise à jour d'un cycle de la simulation (pseudocode1).

## 2.3 Les types de fourmis

### 2.3.1 Durée de vie et booléen end_of_life

On pose que la fourmi Generator possède une durée de vie sans limitation si elle dispose de nourriture (section suivante). Les autres types de fourmis vivent **bug_life** cycles de mise à jour. Le cycle de la simulation pendant lequel elles naissent compte comme premier cycle de vie car leur état est actualisé dans le même cycle de simulation (pseudocode). L'age d'une fourmi déjà existante est incrémenté au début de sa mise à jour puis doit disparaitre si cet age est égal à l'age maximum **bug_life** en fin de mise à jour (pseudocode).

Comme le montre le pseudocode d'une mise à jour, il est possible qu'une fourmi d'une fourmilière soit détruite par une fourmi Predator d'une autre fourmilière avant ou même après avoir effectué sa propre mise à jour. Quand cela arrive, le booléen ***end_of_life*** de la fourmi détruite doit passer à vrai au moment de l'action de la fourmi Predator. En cas de mise à jour d'une fourmi dont le booléen ***end_of_life*** est vrai, celle-ci occupe encore son espace dans la grille mais n'effectue aucun déplacement ni action. C'est seulement en fin de pseudocode que toutes les fourmis ayant le booleen ***end_of_life*** à vrai sont éliminées des structures de données et que la nourriture qu'elle transportaient éventuellement est laissée sur place.

### 2.3.2 Fourmi de type Generator

Une fourmi Generator est mise à jour avant les autres types de fourmis (pseudocode). La fourmi Generator doit toujours rester à l'intérieur de la bordure de la fourmilière (c'est à dire pas de superposition avec la bordure de la fourmilière). Elle peut se déplacer dans toutes les directions, d'une case à la fois par mise à jour. Les cellules doivent être libres pour pouvoir s'y déplacer. Si la fourmilière décroit de taille et que la fourmi Generator se superpose à sa bordure, alors celle-ci *n'existe plus* (le booléen **end_of_klan** passe à vrai ce qui empêche la mise à jour des autres fourmis) et la fourmilière disparait à la fin de la mise à jour (pseudocode).

#### 2.3.2.1 Stockage de nourriture

La Fourmi Generator est responsable du stockage du total de la nourriture dans ***total_food***. La valeur **val_food** de l'élément de nourriture rapporté par une fourmi Collector est ajoutée à ***total_food*** si cette fourmi Collector *parvient au contact de sa fourmilière* par un coté ou un coin.

#### 2.3.2.2 Consommation de nourriture et risque de destruction

La consommation de la nourriture depend du nombre total de fourmis ***nbT*** : une quantité de nourriture de (***nbT******food_rate**) est ainsi consommée seulement si la valeur de ***total_food*** le permet.

Si ***total_food*** devient négatif ou nul à cause de la consommation de nourriture alors la Fourmi Generator *n'existe plus* (le booléen ***end_of_klan*** passe à vrai) et la fourmilière disparait (cf fin du pseudocode).

#### 2.3.2.3 Naissance de fourmis

La probabilité de naissance d'une nouvelle fourmi est **Min(1, *total_food**birth_rate)** ; cf section 3.1.1 pour la mise en oeuvre de cette probabilité. En cas de naissance, le type de fourmi qui est créé depend du mode (**FREE** ou **CONSTRAINED**) actuel de la fourmilière car les proportions de type de fourmis changent selon le mode. Le nom du mode (**free** ou **constrained**) remplace le **X** dans le nom de la constante indiquant les proportions ci-dessous. Cela étant posé, les priorités de créations conduisent à la méthode de creation suivante:

- Il faut d'abord garantir une proportion d'au moins **prop_X_collector** de Collector. Si c'est vérifié, il faut alors garantir une proportion d'au moins **prop_X_defensor** de Defensor. Enfin si c'est le cas, alors on doit créer un Predator.

Exemple: soit une fourmilière en mode **FREE** avec des seuils de proportion de **0.7** de Collector et de **0.2** de Defensor. Si cette fourmilière contient 100 fourmis dont 72 fourmis Collector, 23 fourmis Defensor et 5 fourmis Predator, alors les seuils de proportions minimales sont respectés pour les fourmis Collector et Defensor, donc on va donc créer une fourmi Predator à la prochaine naissance. Si par contre on avait eu la distribution de 75,

18 et 7 alors on aurait donné naissance à une fourmi Defensor, tandis qu'une Distribution de 68, 28 et 4 aurait produit la naissance d'une fourmi Collector.

Condition supplémentaire de naissance: un algorithme de recherche doit trouver un espace libre de la taille de la fourmi à l'intérieur de la fourmilière pour autoriser sa création. On acceptera un algorithme de balayage systématique du carré de la fourmilière et qui retourne le premier espace libre.

### 2.3.3 Fourmi de type *Collector*

Une fourmi Collector peut se déplacer seulement en diagonale, d'une case à la fois par mise à jour (Fig 3). Les cellules doivent être libres pour pouvoir s'y déplacer. Ce type de fourmi est chargé d'aller chercher un (seul) élément de nourriture à la fois. On suppose qu'une fourmi Collector connait la position de tous les éléments de la simulation (autres fourmis et nourriture). A chaque mise à jour elle se dirige toujours vers l'élément de nourriture le plus proche **et** qui se trouve sur la même famille de diagonales que le centre de la fourmi:

- S'ils appartiennent à la même famille de diagonales, l'élément de nourriture est acquis dès que la fourmi se superpose avec cet élément de nourriture. Dans ce cas la variable ***food*** de la fourmi passe de l'état **EMPTY** à l'état **LOADED** et l'élément de nourriture n'est plus visible pour les autres fourmis Collector (il n'existe plus). La section 2.3.3.6 détaille l'action d'une fourmi Collector dans l'état **LOADED**.
- S'ils n'appartiennent pas à la même famille de diagonales, ou si la fourmi est déjà dans l'état **LOADED**, l'élément de nourriture se comporte comme un obstacle pour la fourmi Collector.

2.3.3.1 Test d'atteignabilité: Pour déterminer si un élément de nourriture se trouve sur la même famille de diagonales que le centre d'une fourmi Collector, ils doivent vérifier l'une ou l'autre des propriétés suivantes:

- Même parités des 2 coordonnées de la fourmi et de la nourriture (paire ou impaire)
- Parités différentes des 2 coordonnées de la fourmi et de la nourriture (paire et impaire)

2.3.3.2 Critère de distance entre une fourmi Collector et un élément de nourriture: une fois que l'on sait qu'un élément de nourriture est atteignable, notons (**vx, vy**) les coordonnées du vecteur reliant le Collector à l'élément de nourriture. Le critère de proximité est donné par **Max(|vx|, |vy|).**

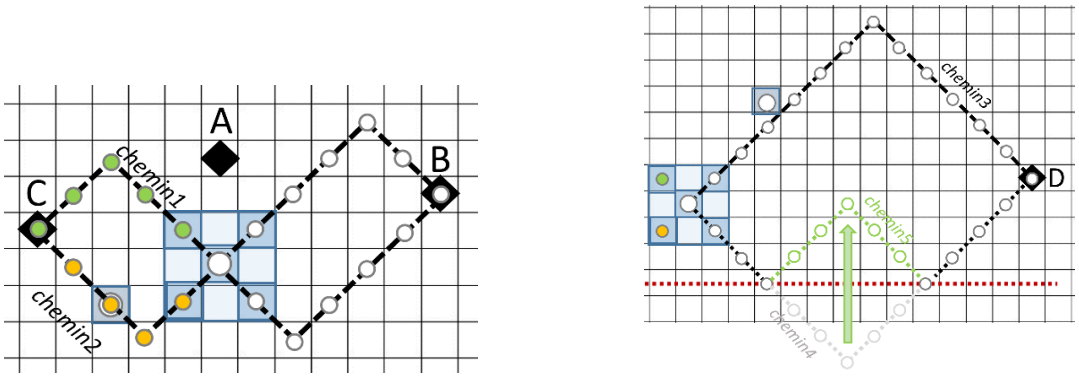



**Fig 4**: (a) détermination de l'élément de nourriture vers lequel la fourmi Collector va se diriger pour la mise à jour courante. L'élément A n'est pas atteignable car il est sur l'autre famille de diagonales que la fourmi. L'élément B demande 5 déplacements tandis que l'élément C n'en demande que 4: l'élément C est donc choisi. Il reste à déterminer lequel des 2 chemins extrêmes a le moins de superpositions avec d'autres fourmis: c'est le chemin1 qui est choisi. (b) Si un chemin passe à l'extérieur du monde (chemin4 vers D), on construit le chemin corrigé en réfléchissant la partie sous la ligne rouge décalée de **sizeC/2** par rapport au bord (chemin5) ; c'est lui qui est choisi car il n'a pas de superpositions.

2.3.3.3 Evaluation de 1 ou de 2 chemins: en cas d'égalité des valeurs absolues de vx et vy, il n'existe qu'un seul chemin. En cas d'inégalité, il existe plusieurs chemins possibles mais on demande d'examiner seulement les 2 chemins extrêmes qui comportent un seul changement de direction selon les axes (1,1) et (1,-1)[2] comme dans l'exemple de la Fig 4a :

- **(vx+vy)/2** sauts selon (1,1) puis **(vx-vy)/2** sauts selon (1,-1)
- **(vx-vy)/2** sauts selon (1,-1) puis **(vx+vy)/2** sauts selon (1,1)

[2] Vous pouvez faire un autre choix d'axes de direction du moment qu'ils sont orthogonaux et selon les diagonales

On choisit alors le chemin qui produit le moins de superposition de cellules de la fourmi Collector avec d'autres fourmis ou élément de nourriture (Fig 4a chemin1). Le test de superposition doit être effectué sur l'ensemble des cellules de la fourmi Collector. On examine les superpositions seulement quand la fourmi se trouve au centre des cellules des points du chemin. S'il y a égalité du nombre de cellules avec superposition, il faut privilégier le chemin qui permet d'avancer pour la mise à jour courante. Si les 2 chemins permettent d'avancer, vous avez la liberté de choix.

2.3.3.4 Construction d'un chemin alternatif s'il fait sortir la fourmi du monde, même partiellement (Fig 4b chemin4): Il faut construire le chemin qui se réfléchit à l'intérieur du monde sans faire sortir la fourmi du monde (Fig4b chemin5). Le chemin réfléchi est systématiquement obtenu en prenant la symétrie de la partie incorrecte par rapport à la ligne ou colonne décalée de **sizeC/2** par rapport à la première ou dernière ligne/colonne (ligne rouge de la Fig4b).

2.3.3.5 Fin prématurée: Si par malheur une fourmi Collector **LOADED** est au contact d'une fourmi Predator d'une autre fourmilière (avant ou après son déplacement) alors son booléen ***end_of_life*** passe à vrai et elle disparait à la fin de la mise à jour (pseudocode1) et un nouvel élément de nourriture est créé à la position de son centre (cf section 2.1.5).

2.3.3.6 Action avec/sans nourriture: Une fourmi Collector dans l'état **LOADED** doit seulement se diriger vers sa fourmi Generator. Le calcul de chemin est du même type que pour aller vers un élément de nourriture. Il suffit qu'elle soit partiellement en contact avec la bordure ou un coin de sa fourmilière pour que la valeur de l'élément de nourriture soit ajouté à ***total_food*** de la Fourmi Generator et la fourmi Collector repasse dans l'état **EMPTY** ; cela est fait au cours de la même mise à jour.

Une fourmi Collector qui est dans l'état **EMPTY** au début de sa mise à jour et qui n'a aucun élément de nourriture disponible doit advancer dans la meilleure direction qui la fait sortir de sa fourmilière (objectif prioritaire) et qui l'éloigne des bords du monde (objectif secondaire en cas d'égalité de l'objectif prioritaire pour plusieurs directions). Elle peut rester immobile une fois à l'extérieur de sa fourmilière.

### 2.3.4 Fourmi de type *Defensor*

Une fourmi Defensor peut se déplacer seulement en horizontalement ou verticalement, d'une case à la fois par mise à jour. Les cellules doivent être libres pour pouvoir s'y déplacer.

A sa création elle peut être placée n'importe où dans un espace libre de la fourmilière. Ensuite elle doit éventuellement se déplacer pour toujours rester sans superposition et *tangente au coté interne de la bordure* de la Fourmilière (Fig 5a). Par exemple, si la taille de la fourmilière augmente elles doivent s'y déplacer (Fig 5b). Si la taille diminue et qu'une fourmi Defensor se superpose à la bordure ou est à l'extérieur de sa fourmilière alors son booléen ***end_of_life*** passe à vrai et elle disparait en fin de pseudocode (Fig 5c).

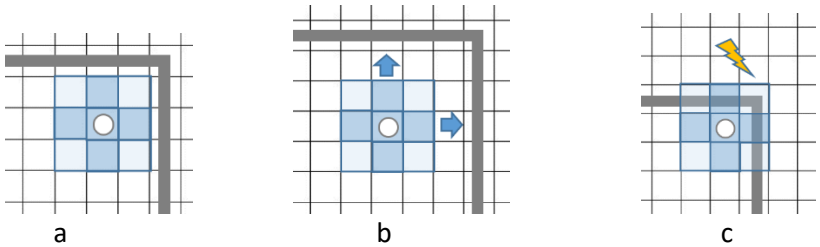



Figure 5: (a) choix possible de positionnement initial d'un Defensor à sa naissance, (b) la fourmilière ayant grandi, le Defensor doit être à nouveau tangent ; pour cela il doit bouger soit horizontalement vers la doite ou verticalement vers le haut (mais ne peut pas faire les 2 en même temps). (c) ici la taille de la fourmilière ayant diminué il faudrait que le Defensor bouge à la fois horizontalement vers la gauche et verticalement vers le bas mais cela n'est pas possible pendant une seule mise à jour. Le Defensor sera donc supprimé en fin de mise à jour.

Toute fourmi Collector d'une autre fourmilière est détruite (son booléen ***end_of_life*** passe à vrai) si elle se trouve temporairement au contact d'une fourmi Defensor avant ou après sa mise à jour.

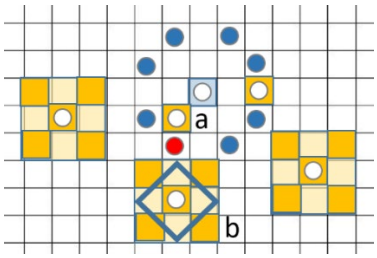
#### 2.3.5 Fourmi de type *Predator*

Une fourmi Predator peut se déplacer comme le cavalier aux échec (un saut par mise à jour, voir Fig 3a). La cellule doit être libre pour pouvoir s'y déplacer (sauf s'il s'agit d'une prédation de Collector/Predator).

A sa création elle est placée dans un espace libre à l'intérieur de la fourmilière. Ensuite, tant que la fourmilière est dans l'état **FREE** une fourmi Predator *doit rester à l'intérieur de la bordure de* sa fourmilière et se diriger vers la plus proche[3] fourmi Collector ou Predator d'une autre fourmilière qui entre dans sa fourmilière. Si la fourmilière est dans l'état **CONSTRAINED** et qu'il n'y a aucune fourmi infiltrée à l'intérieur de la bordure de sa fourmilière alors la fourmi Predator se dirige à l'extérieur vers la plus proche fourmi Collector/Predator d'une autre fourmilière. Une fourmi Predator élimine une fourmi Collector ou Predator d'une autre fourmilière si elle *se superpose ou est en contact par un coté* avec ces autres fourmis (elle peut être en contact avec plusieurs fourmis simultanément comme sur la Fig 6). Le booléen ***end_of_life*** de la ou des fourmis capturées passe à vrai. Une fourmi Predator n'a aucun impact sur une fourmi Defensor ou Generator.

**Attention:** s'il y a contact avec une fourmi Predator adverse, les 2 fourmis Predator sont éliminées.

**Figure 6**: une fourmi Predator bleue effectue un movement (point rouge) vers la fourmi Predator **a** qui est la plus proche et élimine aussi la fourmi LOADED **b**. Cependant, elle disparaît aussi car en contact par un coté avec la fourmi Predator adverse **a**.




## 2.4 Création des éléments de nourriture

Un élément de **Nourriture** occupe une seule cellule (x,y) du monde 2D (Fig 2). Sa valeur nutritive est **val_food**. La décision de creation est effectuée au début de chaque cycle de simulation avec une probabilité de **food_rate** (cf section 3.1.1). S'il y a création, le centre doit être créé *aléatoirement* (cf section 3.1.2) dans le domaine **] 0, g_max-1 [** et ne PAS se superposer à une fourmilière, une fourmi ou un élément de nourriture déjà existant[4]. Si la position n'est pas autorisée, il faut recommencer le tirage d'une nouvelle position (x,y) jusqu'à un maximum de **max_food_trial** tentatives. Après quoi on abandonne la création de l'élément de nourriture.

# *3 Actions à réaliser par le programme de simulation*

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

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

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

## 3.1 Contrainte de mise en œuvre du hasard avec C++

### 3.1.1 Génération d'un booléen true avec une probabilité p

Deux tâches doivent faire appel à des probabilités pour leur exécution lors d'une mise à jour:

- Booléen de création d'un élément de nourriture avec une probabibilité de **food_rate**.
- Booléen de création d'une fourmi par une fourmi Generator (section 2.3.2.3).

[3] On utilise la distance euclidienne pour le critère de proximité

[4] Pour ces raisons, la création d'un nouvel élément de nourriture devra être effectuée au niveau du module **simulation** afin d'avoir accès à l'information des autres modules (Fig 9).

Avec C++11, il faut créer un objet de type **bernoulli_distribution** pour chaque générateur de booléen :

```
double p; // probabilité ; voir section 2.3.2.3 et Annexe A
bernoulli_distribution b(p); //booléen true avec une probabilité p
default_random_engine  e;  //à créer une seule fois par execution du programme
… ensuite à chaque mise à jour on obtient un booléen avec cet appel…
      if(b(e)) // création… ;
```

#### 3.1.2 Génération d'une valeur unsigned dans un intervalle [min,max] avec une probabilité uniforme

Ce type de générateur doit être appelé pour les coordonnées (x,y) d'un nouvel élément de nourriture:

```
uniform_int_distribution<unsigned> u(min,max);
default_random_engine  e;  //à créer une seule fois par execution du programme
… ensuite chaque appel u(e) fournit une valeur dans [min, max] pour créer
les valeurs aléatoires de x et y.
```

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

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

### 4.1 Caractéristiques des fichiers tests

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

Le format de fichier est décrit dans le tableau suivant. Le fichier contient d'abord les données des éléments de nourriture puis les données des fourmilières incluant leurs données de haut niveau puis leurs listes de fourmis avec leurs données spécifiques. Il doit y avoir une ligne de fichier par fourmi. Le format indique le nombre et la nature des données. Des exemples de fichiers sont fournis.

format général du fichier

```
# Nom du scenario de test
#
nbN # passer à la ligne ; puis seulement un élément de nourriture par ligne
    x1 y1
    …
nbF # passer à la ligne pour fournir les donnée d'une fourmilière et ses fourmis
    x y side xg yg total_food nbC nbD nbP
        # nbC fourmis Collector ; seulement 1 fourmi par ligne
        x1 y1 age1 food1
        …
        # nbD fourmis Defensor ; seulement 1 fourmi par ligne
        x1 y1 age1
        …
        # nbP fourmis Predator ; seulement 1 fourmi par ligne
        x1 y1 age1
        …
     # s'il y a plus d'une fourmilière, fournir leurs données à la suite
```

Dans ce format ***x*** et ***y*** désignent les coordonnées de la position d'une entité, **xg** et **yg** sont les coordonnées de l'unique fourmi Generator de la fourmilière ; **side** est la taille du coté d'une fourmilière ; **food** est un booléen.

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

La **lecture** doit vérifier les conditions de non-superposition des éléments de nourriture entre eux, des fourmilières entre elles, des fourmis entre elles et des éléments de nourritures avec les fourmis (rappel : un élément de nourriture n'existe plus lorsqu'il est transporté par une fourmi Collector). Si plus de données que nécessaire sont fournies sur la ligne de fichier elles sont simplement ignorées sans générer d'erreur de lecture. Un commentaire peut aussi suivre les données et ne doit pas poser de problèmes.

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

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

## *5. Interface utilisateur (GUI)*

Nous utilisons une seule fenêtre graphique divisée en 2 parties:

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

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

Commandes générales :

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

Simulation :

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

Affichage de données générales :

- ***Nombre d'éléments de nourriture disponibles***

Affichage des donnée de la fourmilière courante (initialement non définie):

- ***Indice de la fourmilière***
- ***total_food***
- ***nombre de Collectors, Defensors, Predators***


**Fig** 7 : GUI

## *6. Affichage et interaction dans la fenêtre graphique*

A partir du rendu 2, l'exécution du programme ouvre une fenêtre GTKmm contenant l'interface graphique utilisateur (Fig 7) et le dessin de la simulation dans un *canvas*. Le dessin du monde complet doit couvrir l'espace **[0, g_max]** selon X et Y (Fig 2). L'espace 2D du monde est représenté seulement par une bordure d'épaisseur de une cellule d'indice **0** et **g_max-1** selon X et Y.

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

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

- On travaille en **dark mode** = le fond du dessin est noir.
- La bordure du carré 2D du monde est **blanche** (occupe les bords d'indice **0** et **g_max-1**)
- Un élément de nourriture est un losange **blanc** plein qui occupe une cellule de la grille (Fig 2)**.**
- Chaque fourmilière possède une bordure d'une couleur primaire. Elles sont dessinées dans l'ordre des **couleurs primaires : rouge, vert, bleu, jaune, magenta, cyan** et dans l'ORDRE des fourmilières dans la simulation : la première fourmilière lue dans le fichier est la première dans la simulation et elle utilise la couleur rouge, la suivante la couleur verte, etc... S'il y a plus de fourmilières que ces 6 couleurs prédéfinies dans **graphic** (section 7) alors on ré-utilise les mêmes 6 couleurs primaires avec un modulo. Une fourmilière conserve toujours la MEME couleur pendant toute la durée de la simulation, même si d'autres fourmilières sont détruites au cours du temps.
- La bordure et les fourmis sont de la même couleur que leur fourmilière. Il faut respecter leur taille. Faire un motif à base de carrés en X pour les Collectors et en + pour les Defensor et un simple carré pour les Prédators (Fig 3a).

### 6.1 Interaction avec le clavier

Utiliser la touche clavier '**s**' pour faire la même action que le bouton Start/Stop
Utiliser la touche clavier '**1**' pour faire la même action que le bouton Step
Utiliser la touche clavier '**n**' pour passer à l'affichage des données de la fourmilière suivante (rebouclage à un affichage vide quand on arrive en fin de liste de fourmilières)
Utiliser la touche clavier '**p**' pour passer à l'affichage des données de la fourmilière précédente (rebouclage à la dernière fourmilière quand c'est effectué depuis le mode d'affichage vide)

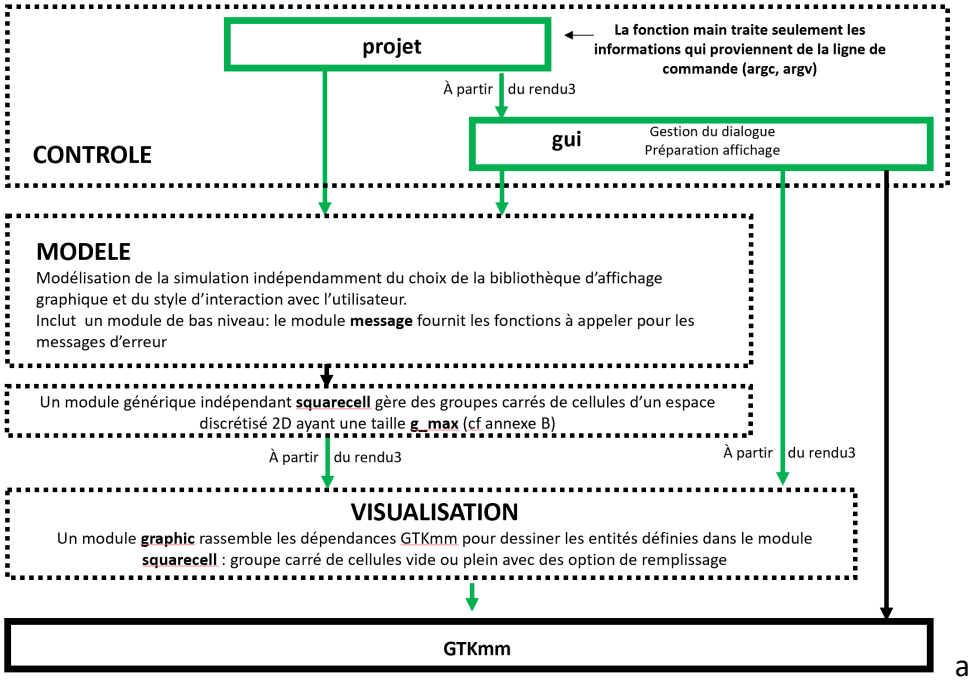
## *7. Architecture logicielle*

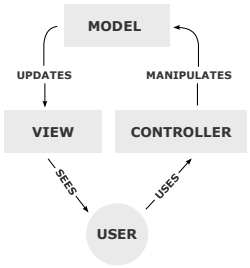
### 7.1 Décomposition en sous-systèmes

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

- **Sous-système de Contrôle : son but est de gérer le dialogue avec l'utilisateur (Fig 8b).** Si une action de l'utilisateur impose un changement de l'état de la simulation, ce sous-système doit appeler une fonction du **sous-système du Modèle** qui est le seul responsable de gérer les structures de données de la simulation (voir point suivant).
  Le sous-système de contrôle est mis en œuvre avec deux modules :
  - Le module **projet** qui contient la fonction **main** : son rôle est modeste car il est seulement responsable de traiter les éventuels arguments fournis sur la ligne de commande au lancement du programme. Pour le rendu1, le sous-système de **Contrôle** ne contient que le module **projet**.
  - le module **gui** est créé à partir du rendu2 pour gérer le dialogue utilisateur à l'aide de l'interface graphique mise en œuvre avec GTKmm.
- **Sous-système du Modèle**: est responsable de gérer les structures de données de la simulation. Il est mis en œuvre sur plusieurs niveaux d'abstractions selon les Principes d'Abstraction et de Ré-utilisation (section 7.2).
- **Utilitaire générique indépendant du Modèle :** un module **squarecell** gère des groupes carrés de cellules d'un espace discrétisé 2D ayant une taille **g_max** (cf annexe B) ; c'est l'équivalent d'une bibliothèque mathématique.

- **Sous-Système de Visualisation :** le module **graphic** dessine l'état courant de la simulation à l'aide des entités élémentaires gérées par le module **squarecell**. Le module **graphic** rassemble les dépendances vis-à-vis de GTKmm pour faire les dessins.




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

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

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

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

- ***Au plus haut niveau***, le module **simulation** gère le déroulement de la simulation et les autres actions (lecture, écriture de fichier). Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi, en vertu du principe d'abstraction le module **simulation** *est le seul module* dont on peut appeler des fonctions en dehors du MODELE (Fig 9)[5].

- ***Niveau intermédiaire***: il faut au moins considérer un module distinct pour les entité de nourriture, fourmilière et de fourmi. Le module **fourmi** doit être conçu comme une hiérarchie de classes pour intégrer les 4 sortes de fourmis (nous accepterons une version simplifiée, sans hiérarchie de classe, pour le **rendu1** seulement).

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

## 7.3 Module générique indépendant squarecell pour les calculs géométriques discrets

### 7.3.1 Indépendance de squarecell

Ce module est équivalent à une bibliothèque mathématique destinées à être utilisées par de nombreux autres modules de plus haut niveau (principe de Ré-utilisation). L'idée fondamentale est qu'il doit aussi être conçu pour être utilisable par d'autres programmes très différents de Tchanz. Pour cette raison **AUCUN** des noms de

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

types/concepts de Tchanz ne doit apparaître dans **squarecell** ; on doit seulement y trouver des types génériques tels que point (désignant une seule cellule de coordonnées **x** et **y**), vecteur (reliant deux cellules), et groupe carré de cellules dans l'espace 2D (**Square**). On y trouvera des calculs de distance et des tests de superposition ou proximité entre les entités **Square** passées en paramètres ou mémorisées par ce module (cf Rendu1).

Les classes du MODELE devront utiliser les types mis à disposition dans l'interface de **squarecell** et vont appeler les fonctions aussi mises à disposition dans cette interface pour traiter tous les calculs géométriques discrets (ex : calcul de distance, détection de superposition).

En particulier, deux fonctions offertes par l'interface devront faire explicitement des tests de validité sur les valeurs d'indice et de carré. Elles devront être utilisées au moment de la lecture de fichier (rendu1) et la détection d'une erreur devra conduire à l'arrêt du programme avec un appel aux fonctions fournies dans le module **error_squarecell**. Nous vous encourageons à systématiquement détecter les valeurs incorrectes d'indices de coordonnées (x,y) au sein de vos autres fonctions de ce module pour l'efficacité de la mise au point du projet.

### 7.3.2 L'usage de types concrets est imposé pour les types de squarecell

Un type **concret** est typiquement une structure dont on montre le modèle dans l'interface du module (**squarecell.h**) de façon à *pouvoir accéder directement 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 pour alléger l'écriture du code de plus haut niveau. La programmation orientée objet n'interdit pas l'usage de types concrets s'ils sont utilisés dans le bon contexte ; **squarecell** en est un car les entités offertes par ce module sont élémentaires et seront validées et stabilisées dans le premier rendu du projet.

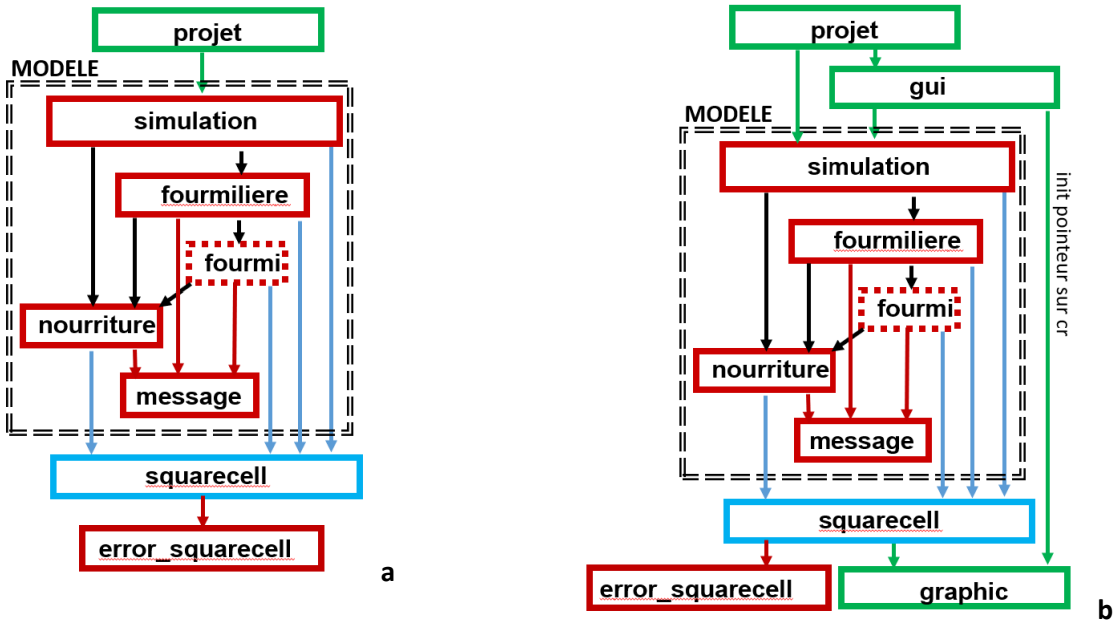



**Figure 9** : (a) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (rendu1); (b) modules et dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendus 2 et 3)

## 7.4 Module graphique de bas-niveau (graphic)

A partir du rendu2 le module **squarecell** va offrir des fonctions de dessin pour le type **Square** avec suffisamment d'options générales pour permettre de différencier les éléments du Modèle. Cependant le module **squarecell** doit rester indépendant d'une libraire graphique particulière (principe de regroupement des dépendances). C'est pourquoi les dépendances vis-à-vis de la bibliothèque **GTKmm** doivent être

rassemblées dans le module **graphic**. C'est dans ce module qu'on définit une table de couleurs prédéfinies et les fonctions de tracé des formes géométriques de carrés ou de droites (cf section 6 pour les conventions à suivre pour les formes et les couleurs).

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

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

**8.1 Rendu1 :** Son architecture est précisée par la Fig 9a.

Ce rendu sera toujours testé en indiquant un nom de fichier de test sur la ligne de commande selon la syntaxe suivante **: ./projet test1.txt**

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

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

Le rendu1 ne doit PAS utiliser GTKmm.

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

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

Un rapport devra décrire les choix de structures de donnée en *anticipant* comment elles seront utilisées pour le rendu final.

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

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

Un rapport final devra décrire votre approche pour les déplacements non-précisés dans les spécifications.

## *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.

enum Etat_fourmiliere {**FREE, CONSTRAINED**} ;
enum Etat_collector {**EMPTY, LOADED**} ;

constexpr short unsigned **maxF**(25) ;
constexpr short unsigned **sizeG**(5) ;
constexpr short unsigned **sizeC**(3) ;
constexpr short unsigned **sizeD**(3) ;
constexpr short unsigned **sizeP**(1) ;

constexpr short unsigned **bug_life**(300) ;
constexpr short unsigned **val_food**(50) ;

constexpr double **food_rate**(0.1) ;
constexpr double **max_food_trial**(10) ;
constexpr double **birth_rate**(0.00005) ;
constexpr double **prop_free_collector**(0.85) ;
constexpr double **prop_free_defensor**(0.1) ;
constexpr double **prop_constrained_collector**(0.6) ;
constexpr double **prop_constrained_defensor**(0.1) ;

---

## *ANNEXE B : constantes destinées au module générique squarecell*

constexpr short unsigned **g_dim**(7) ;
constexpr short unsigned **g_max**(pow(2,**g_dim**)) ;

---

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

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