C++ PoP – Sections Electricité et Microtechnique

Printemps 2024 : *MicroRécif* © R. Boulic & collaborators

Simulation de l'évolution de micro-organismes dans une goutte d'eau en 2D

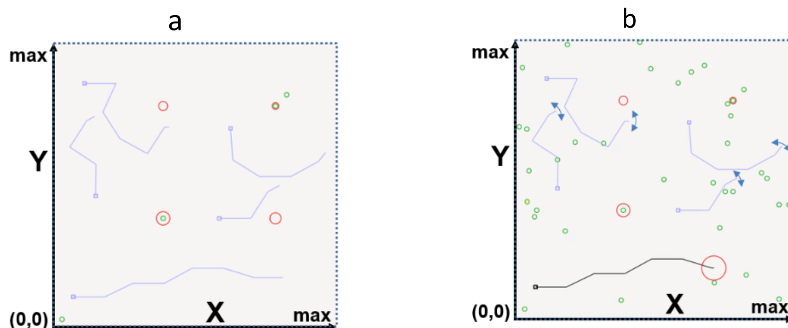


Fig 1 : espace Monde dans lequel évoluent les algues (vertes), les coraux (bleus ou noirs), et les scavengers (rouges). L'image b montre le type de mouvement de rotation des coraux vivants. Le scavenger du bas mange un segment de corail mort.

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 <u>individuel</u> noté pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de <u>l'ensemble du projet</u>. 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 la *double précision* pour des calculs en virgule flottante. Notez cependant que des entiers doivent être utilisés pour certaines variables (longueur ou rayon) pour simplifier la simulation.

<u>Pitch du sujet</u> *: trois types d'entités minuscules vivent dans l'équivalent d'une goutte d'eau dans un récipient carré qui est éclairé par une source lumineuse : peuvent-elles trouver un équilibre sur le long terme ? La plus simple est une algue fixée sur le fond ; elle n'a besoin que de lumière. Les algues servent de nourriture à une entité ayant la forme d'un segment linéaire partiellement mobile avec un coté fixe et pouvant tourner autour de ce point pour capturer les algues. Cette entité croit linéairement selon des règles librement inspirées par le corail. Il faudra en particulier empêcher les collisions entre les segments de ces sortes de coraux. Lorsque ces coraux arrivent en fin de vie, elles sont mangées par la dernière entité de type charognard (la donnée utilise plutôt le mot anglais scavenger et son abréviation «sca») est aussi circulaire et peut se déplacer en translation.*

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 l'ensemble des entités. 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. Un compteur de mises à jour ***nbSim*** doit permettre de mesurer la durée de la simulation. A priori celle-ci peut durer indéfiniment.

La simulation doit tenir à jour le nombre d'entité de chaque type (***nbAlg, nbCor, nbSca***) afin de les afficher dans l'interface graphique (section 6).

Nous demandons de structurer la mise à jour selon les étapes suivantes (détails dans les sections suivantes):

```
// UNE mise à jour de la simulation
// entrée modifiée : ensemble des entités

   Incrémentation du compteur de mise à jour

   Mise à jour des algues
      Incrémentation de l'age et disparition si l'age max_life_alg est atteint
      Si(activé) Création en fonction du taux de naissance

   Mise à jour des coraux
      Incrémentation de l'age et mort si l'age max_life_cor est atteint
      Si longueur_segment< l_repro alors Rotation pour atteindre algue cible
            Si algue cible atteinte alors extension longueur du segment de delta_l
      Sinon alternance entre ajout nouveau segment ou reproduction par division

   Mise à jour des scavengers
      Incrémentation de l'age et disparition si l'age max_life_sca est atteint
      Si LIBRE alors Déplacement vers DEAD corail disponible le plus proche
      Sinon       Alimentation sur le corail mort par déplacement de delta_l
                  Reproduction par division dès que son rayon atteint r_sca_repro
                  Le corail disparait dès que tous ses segments sont consommés
```

Ebauche de 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 indépendantes de celle-ci (cf section 4 et 5).

La figure 1 montre le système de coordonnée ***Monde*** de la simulation avec **X comme axe horizontal, orienté positivement vers la droite**, **et Y comme axe vertical, orienté positivement vers le haut**. Le récipient est représenté par un domaine **[0, max]** ; il est dessiné à l'aide d'un carré indiquant sa frontière. Les entités doivent être entièrement comprise dans ce domaine. Les composantes de la simulation sont décrites dans les sections suivantes.

<u>Modélisation de la gestion des collisions entre entités :</u>
Toutes les entités doivent rester entièrement dans le récipient ; ses 4 cotés devront être pris en compte comme frontière de l'espace de la simulation. Par ailleurs on peut supposer que les 3 entités existent sur 3 niveaux de profondeur distincts : les algues au fond, le corail au milieu, les scavengers en surface. Ainsi les entités algues et les scavengers peuvent se superposer entre elles et avec le corail. La seule restriction importante est que les segments des coraux ne peuvent PAS se superposer.

2.1 Les éléments de la simulation

En plus du récipient, trois types d'entités existent dans cette simulation : algue, corail, scavenger.

Toutes ces entités possèdent quelques propriétés communes :

- La ***position*** d'un point de référence (x, y) qui doit appartenir à l'espace du récipient **[1, max-1]**
    - Un ***age*** défini par le nombre de mises à jour effectuées. A leur création, l'***age*** doit être initialisé à 1 mise à jour. Par conséquent l'***age*** mémorisé dans un fichier doit aussi être strictement positif.

2.1.1 Entité Algue

Cette entité est fixe dans l'espace. Sa position initiale est déterminée à sa naissance dans l'intervalle **]0, max[** selon x et y. A chaque mise à jour de la simulation, si la création d'algue est activée, le taux de naissance **alg_birth_rate** détermine si une entité de ce type est créée (ou pas).

Lorsque l'age d'une entité de type algue est égal à son age maximum **alg_life_max** elle disparait de la simulation.

Lorsqu'une entité de type algue est consommée par une entité de type corail (conditions indiquées dans section suivante) la longueur du segment de corail augmente de **delta_l** et l'algue disparait de la simulation.

2.1.2 Entité Corail

La position (x,y) de son origine dans le repère Monde (nombre à virgule) reste aussi fixe dans l'espace. Chaque corail possède aussi un identificateur entier qui doit être unique, un statut (**DEAD** ou **ALIVE**), une direction de rotation (**TRIGO** ou **INVTRIGO**), un statut de développement (**EXTEND** ou **REPRO**) et un nombre de segment strictement positif. Chaque segment est représenté par une longueur (nombre entier positif) et par l'angle α entre le segment et l'axe *X* du repère Monde (nombre à virgule) comme illustré sur le dessin de la Fig 2b. L'angle α doit être mesuré en rd et être compris dans l'intervalle $[-\pi, \pi]$. On simplifie les calculs en considérant qu'un segment n'a pas d'épaisseur ; la visualisation lui en donne seulement une pour améliorer leur visibilité. L'extrémité du dernier segment est appelée son ***effecteur*** (comme pour un bras articulé en robotique).

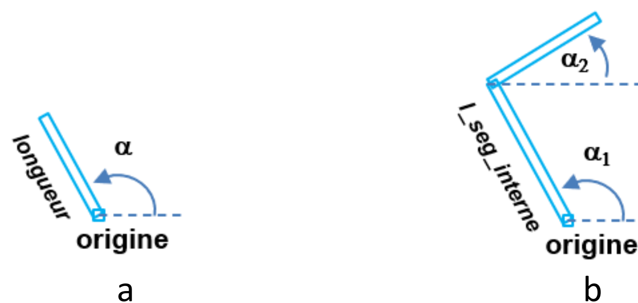


Fig 2 : Vues de dessus d'un corail à un seul segment (a) et d'un autre à deux segments (b).

Lorsque l'age d'une entité de type corail est égal à son age maximum **cor_life_max** son statut passe de ALIVE, représenté par la couleur bleue, à DEAD, représenté par la couleur noire ; ce corail reste immobile et ne disparait de la simulation que lorsque la consommation par un scavenger est terminée (détails section 2.1.3). Il faut néanmoins continuer à le prendre en compte pour les détections de collisions.

2.1.2.1 Mode d'alimentation d'un corail

Un corail est automatiquement en recherche de nourriture quand la longueur de son dernier segment est strictement inférieure à **l_repro**. Pour cela, seul le dernier segment d'un corail est mobile par rotation autour de son origine ou du point qui le relie au segment qui le précède.

Le corail étant incapable de savoir quelle est l'algue la plus proche, sa direction de rotation (notée ***dir_rot***) est constante et elle *change pour la direction de rotation inverse seulement quand une collision est détectée.* Les deux directions possibles sont donc dans le sens trigonométrique (TRIGO) ou l'inverse (INVTRIGO).

Lors d'une mise à jour dans le mode alimentation, l'incrément angulaire maximum est de **delta_rot** dans la direction ***dir_rot*** (Fig 3a). La figure 3b montre cependant que cet incrément est limité à la variation angulaire qui permet d'atteindre l'algue la plus proche (angulairement) dans cette direction. Il n'est possible de consommer qu'une seule algue par mise à jour de la simulation. Cette consommation produit l'accroissement **delta_l** de la longueur du segment. Cependant ATTENTION : si on détecte une collision du fait de la rotation totale ou partielle de **delta_rot** (Fig3c) ou de l'accroissement **delta_l** de longueur (Fig3d), alors la mise à jour avec éventuelle consommation d'algue est annulée et ***dir_rot*** passe dans l'état inverse.

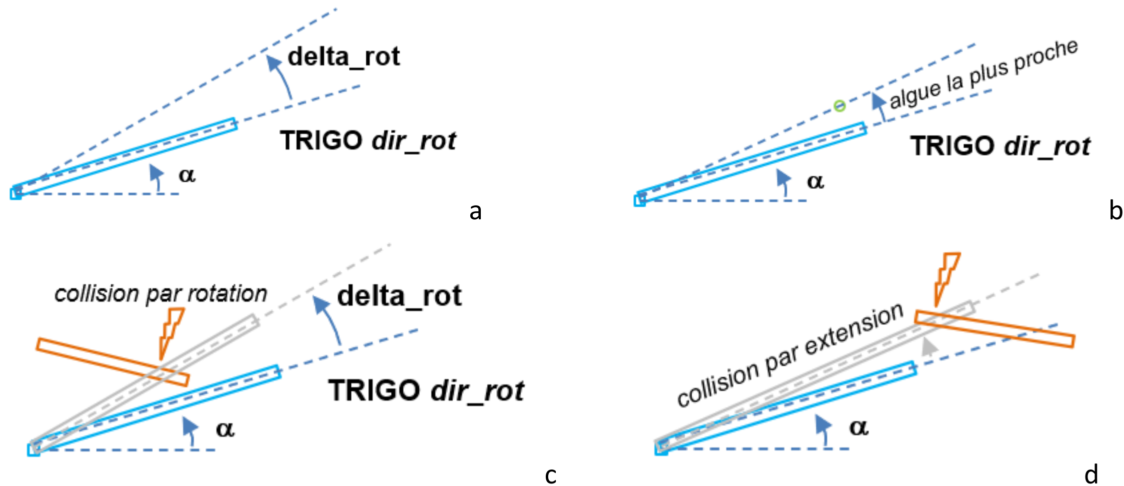


Fig 3 : recherche d'algue dans la direction TRIGO avec rotation maximum possible (a) ou arrêt au centre de la première algue trouvée (b). Invalidation de la mise à jour si une collision est détectée lors de la rotation (c) ou lors de l'extension **delta_l** résultant de la consommation d'une algue (d)

2.1.2.2 Alternance entre création d'un nouveau segment et création d'un nouveau corail

Le corail se transforme lorsque la longueur de son dernier segment atteint **l_repro**. Si son statut de développement est EXTEND alors le corail produit un nouveau segment de longueur **l_repro - l_seg_interne** (Fig 4a-b) et le statut passe à REPRO. Sinon, si le statut est à REPRO alors il crée un nouveau corail de longueur **l_repro - l_seg_interne** tandis que le dernier segment passe à **l_repro/2** (Fig 4c) et le statut passe à EXTEND.

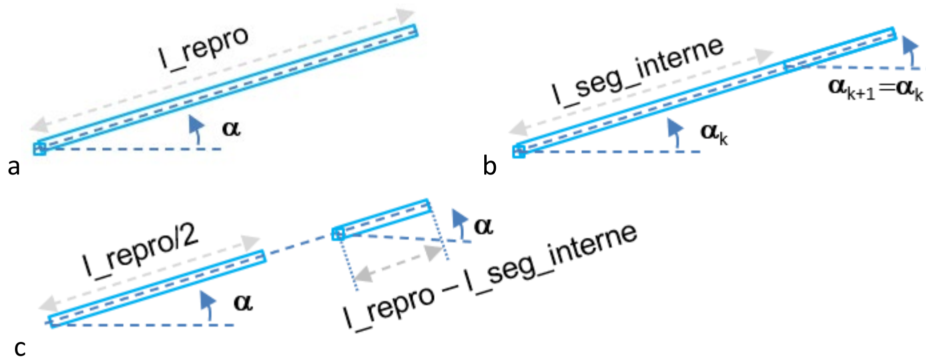


Fig 4 : création d'un nouveau segment : avant (a) et après (b) ;
création d'un nouveau corail : avant (a) et après (c)

2.1.3 Entité Scavenger

Cette entité peut être mobile en translation ; elle se déplace en ligne droite vers sa cible, au maximum d'une distance **delta_l** par mise à jour. La simulation doit respecter les contraintes suivantes pour déterminer cette cible:

- Seul un DEAD corail peut être une cible de scavenger ; le scavenger se place sur l'effecteur du corail.
- Un seul scavenger peut manger un DEAD corail ; il est alors dans l'état MANGE et mémorise d'identificateur du corail qu'il est en train de manger (Fig1).
- Un scavenger qui n'est pas en train de manger est dans l'état LIBRE ; il doit être attiré vers le DEAD corail dont l'effecteur est le plus proche s'il n'est pas en train d'être mangé par un autre scavenger.
    - S'il n'y a aucun DEAD corail disponible, le scavenger reste immobile (Fig1)

Lorsqu'un scavenger est dans l'état MANGE et que son rayon est strictement inférieur à **r_sca_repro** il consomme le corail en se déplaçant le long du dernier segment d'une distance **delta_l** par mise à jour (Fig 5a-b : ce nombre étant entier, on a la garantie qu'un segment est totalement consommé en un nombre entier de mises à jour). Chaque mise à jour de consommation par un scavenger augmente son rayon de la quantité **delta_r_sca**. Lorsque le rayon du scavenger est égal à **r_sca_repro** il réduit son rayon à **r_sca** en gardant le

même centre et produit derrière lui un nouveau scavenger dont le centre est est aussi sur la ligne d'orientation α, à la distance **delta_l** de son centre, et de rayon **r_sca** (Figure 5c-d).

Lorsque l'age d'une entité de type scavenger est égal à son age maximum **sca_life_max** elle disparait de la simulation.

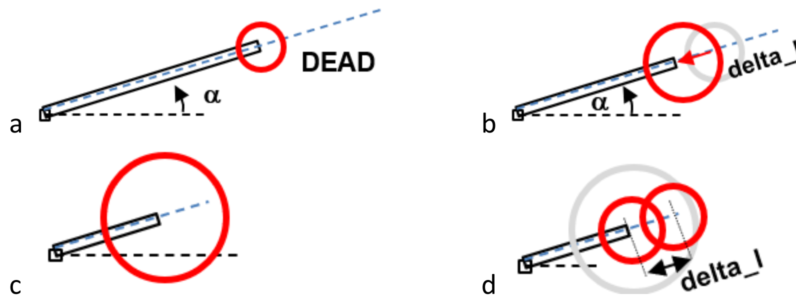


Fig 5 : un scavenger est en position avant (a) et après (b) consommation ;
Le rayon d'un scavenger a atteint la valeur **r_sca_repro** (c) ; il réduit son rayon à **r_sca** en créant un nouveau scavenger de même rayon derrière lui à une distance **delta_l** (d)

2.2 Gestion des collisions

Le choix d'effectuer la sauvegarde de la simulation dans un fichier formaté introduit des arrondis sur les valeurs sauvegardées. Ces arrondis nous imposent d'effectuer des tests de collisions moins stricts quand on lit un fichier en comparaison de ces mêmes tests pendant une mise à jour de la simulation. C'est pourquoi la valeur **epsil_zero** doit être considérée comme nulle pendant les tests de la lecture de fichier.

2.2.1 Appartenance des entités au récipient de la simulation

On ignore le rayon des algues et des scavengers pour ces tests d'appartenance au récipient de la simulation car ces paramètres sont simplement destinés à la visualisation de la simulation. Cette première famille de test n'a besoin d'être faite que lors de la lecture de fichier car la mise à jour de la simulation va garantir de ne pas ré-introduire cette classe de problèmes :

- inclusion des centres des algues et scavengers, et l'origine des coraux dans le domaine **[1, max-1]**

La famille de tests qui suit doit aussi être fait pendant la simulation mais au moins ils garantissent qu'on n'a pas besoin de tester l'intersection des segments de coraux avec les segments des bords du récipient :

- inclusion des autres centre de rotation et de l'effecteur des coraux dans le domaine **]epsil_zero, max – epsil_zero[**

2.2.1 test de collision/superposition de segments de coraux

Les segments des coraux ne doivent pas se superposer ou s'intersecter ni être en collision. Sachant que la lecture de fichier garantit qu'il n'existe pas de collision initiale, la seule possibilité de nouvelle collision est causée par la rotation **delta_r** ou la croissance **delta_l** du dernier segment d'un corail (section 2.1.2.1). Il faut donc refaire la détection de collision du dernier segment après la mise à jour de *chaque* corail. Si on détecte une collision/intersection (intra ou inter corail) alors la mise à jour courante du corail est annulée et la direction de rotation de recherche d'algue s'inverse pour la mise à jour suivante de ce corail. Il faut distinguer deux méthodes de détection qui seront détaillés dans le rendu1 :

- Par construction, deux segments consécutifs d'un corail possèdent un point commun. Pour ces cas, le test doit porter sur l'éventuelle superposition de tels segments consécutifs. On considère qu'il y a superposition de 2 segments consécutifs si *l'écart angulaire* entre les 2 segments (normalisé dans **[$-\pi$, π])** est nul **en lecture de fichier,** ou s'il appartient à l'intervalle **[-delta_rot, delta_rot]** et change de signe lors de la mise à jour en simulation.

- Pour tous les autres cas intra ou inter corail, le projet doit détecter les intersection/superposition par une méthode reposant sur une détection d'intersection avec un calcul de distance pour traiter correctement les cas de superposition/collision.

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 naissance des entités algues
- **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 Mise en œuvre avec C++ *(compilation avec l'option std=c++17)*

On exige l'usage de la fonction **atan2** de **<cmath>** pour obtenir une orientation à partir d'un vecteur (x,y).
Il faut inclure **#include <random>** pour la gestion des probabilités.

3.1.1 Booléen de création d'une algue avec la probabibilité de alg_birth_rate.

Avec C++11, il faut créer un objet de type **bernoulli_distribution** comme suit :

```
//le générateur de nombres aléatoire e  n'est créé qu'une seule fois
default_random_engine  e;       //ré-initialiser à chaque lecture de fichier
                                //ex : avec un appel e.seed(1) ;
double p(alg_birth_rate); // probabilité de la section 2.1.1, Annexe A
… à chaque nouvelle mise à jour:
bernoulli_distribution b(p); //booléen true avec probabilité p
… puis
      if(b(e)) // création d'une algue si le booléen est vrai
```

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'une nouvelle entité algue. Il utilise le même générateur de nombre aléatoire **e** que pour 3.1.1. De plus, sachant qu'on ne veut pas qu'une entité algue soit créée sur un bord du récipient, on produit un nombre entier compris dans l'intervalle [1, max-1]:

```
uniform_int_distribution<unsigned> u(1,max-1);
… ensuite chaque appel u(e) fournit une valeur dans [1, max-1] pour créer
les valeurs aléatoires de la position d'une nouvelle algue 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 **#** en début de ligne, 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 fichier contient d'abord les données des entités algues puis les entités corail puis les entités scavenger. Dans ce format **x** et **y** désignent les coordonnées de la position d'une entité ; **a** désigne une orientation en rd dans l'intervalle $[-\pi, \pi]$, **s** une longueur (entier) de segment et **id** un identificateur entier pour le corail seulement. Le statut_cor d'un corail est représenté par les valeurs 0 (DEAD) ou 1 (ALIVE). La direction de rotation dir_rot est représentée par 0 (TRIGO) ou 1 (INVTRIGO). Le statut_dev est représenté par les valeurs 0 (EXTEND) ou 1 (REPRO). Les informations d'angle et de longueur de segment d'une entité corail sont données sur les lignes distinctes. Le statut_sca d'un scavenger est représenté par les valeurs 0 (LIBRE) ou 1 (MANGE). S'il a le statut

MANGE alors corail_id_cible est l'identificateur du corail qu'il est en train de manger. Sinon il n'y a pas de donnée corail_id-cible après statut_sca. Le fichier contient au moins les 3 nombres d'entités de chaque sorte. Des exemples de fichiers sont fournis.

format général du fichier

```
# Nom du scenario de test
#
# nombre d'entité algue puis les données d'une entité par ligne
nbAlg
    x1 y1 age1 …

# nombre d'entité corail puis les données d'une entité
#        avec d'abord les données générales sur une ligne
#        puis une ligne par segment pour l'angle et la longueur
nbCor
    x1 y1 age1 id1 statut_cor1 dir_rot1 statut_dev1 nbseg1
        a11 s11
        a12 s12
        …
    x2 y2 age2 id2 statut_cor2 dir_rot2 statut_dev2 nbseg2
        a21 s21
        …

# nombre d'entité scavenger puis les données d'une entité par ligne
nbSca
    x1 y1 age1 rayon1 statut_sca1 corail_id_cible1
    …
```

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

La **lecture** doit vérifier :

- L'absence de collision détaillées en section 2.2 (**epsil_zero** est nul en lecture).
- L'age strictement positif
- La longueur des segments est comprise dans **[l_repro-l_seg_interne, l_repro[**
- L'angle des segments est compris dans **[-π, π]**
- Le rayon des scavengers est compris dans **[r_sca, r_sca_repro[**
- L'unicité des identificateurs de coraux
- L'existence d'un corail identifié avec l'identificateur mémorisé par un scavenger dans l'état MANGE

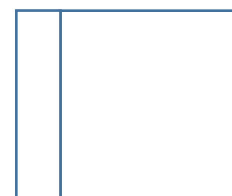
Si plus de données que nécessaire sont fournies sur la ligne de fichier elles sont simplement ignorées sans générer d'erreur de lecture. Un commentaire peut aussi suivre les données et ne doit pas poser de problèmes. Les conséquences d'une détection d'erreur dépendent du rendu du projet (section 8):

- Rendu1 : Dès la première erreur détectés à la lecture, il faut afficher dans le terminal le message d'erreur fourni avec le module **message** (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:

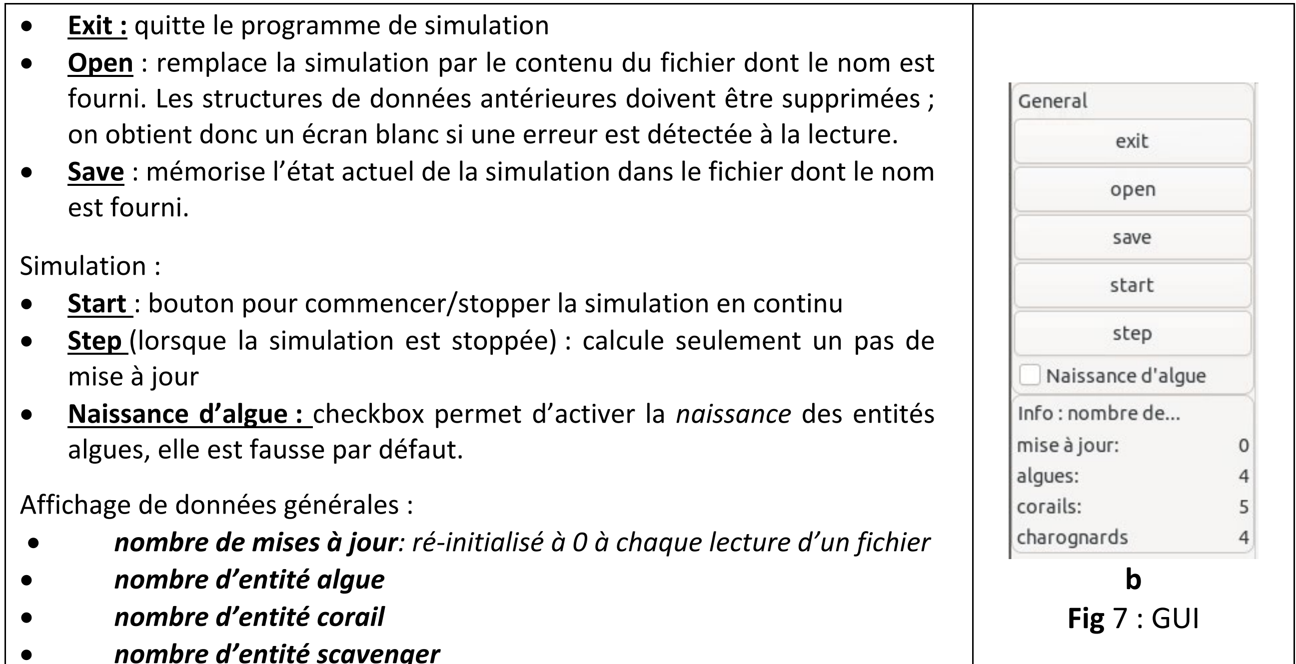
- Les boutons des actions ou affichage de données (Fig 7a partie gauche, et Fig 7b)
- le dessin de l'état courant de la simulation dans un *canvas* (Fig 1 et Fig 7a partie droite). Il s'agit du dessin du monde dans **[0, max]**.



a

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. Les structures de données antérieures doivent être supprimées ; on obtient donc un écran blanc si une erreur est détectée à la lecture.
- **Save** : mémorise l'état actuel de la simulation dans le fichier dont le nom est fourni.

Simulation :
- **Start** : bouton pour commencer/stopper la simulation en continu
- **Step** (lorsque la simulation est stoppée) : calcule seulement un pas de mise à jour
- **Naissance d'algue :** checkbox permet d'activer la *naissance* des entités algues, elle est fausse par défaut.

Affichage de données générales :
- ***nombre de mises à jour**: ré-initialisé à 0 à chaque lecture d'un fichier*
- ***nombre d'entité algue***
- ***nombre d'entité corail***
- ***nombre d'entité scavenger***

b

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, max]** selon X et Y (Fig 1). On demande de matérialiser le domaine par une fine bordure.

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 light mode = le fond du dessin est blanc.
- La bordure du carré 2D du monde est grise
- Une algue est un cercle vide de couleur verte (Fig 1 et 3)**.**
- Un scavenger est un cercle vide de couleur rouge (Fig1, 4 & 5).
- Un corail est un ensemble d'au moins un segment de couleur bleue (ALIVE) ou noire (DEAD) ; sa base fixe est un carré vide de coté **d_cor** de même couleur que les segments.

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

7. Architecture logicielle

7.1 Décomposition en sous-systèmes

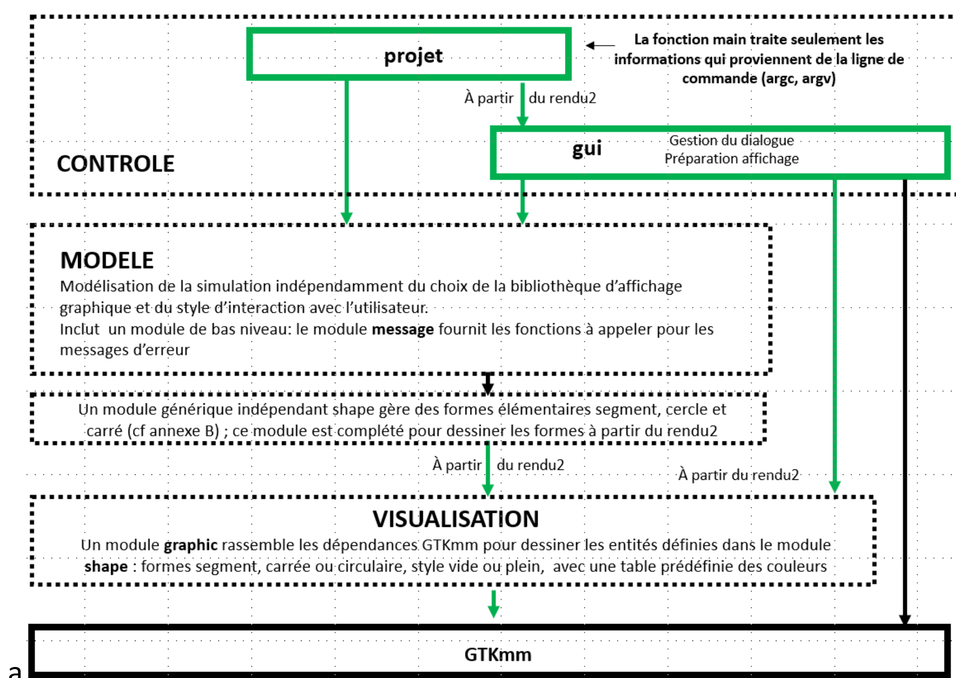
L'architecture logicielle de la figure 8 décrit l'organisation minimum du projet en sous-systèmes avec leur responsabilité (Principe de Séparation des Fonctionnalités) :

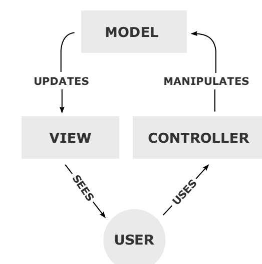
- **Sous-système de Contrôle : son but est de gérer le dialogue avec l'utilisateur (Fig 8b).** Si une action de l'utilisateur impose un changement de l'état de la simulation, ce sous-système doit appeler une fonction du **sous-système du Modèle** qui est le seul responsable de gérer les structures de 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.

- **<u>Sous-système du Modèle</u>**: 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).
- **<u>Utilitaire générique indépendant du Modèle :</u>** un module **shape** gère des formes élémentaires circulaires, segments ou carrées (cf annexe B) ; c'est l'équivalent d'une bibliothèque mathématique.
- **<u>Sous-Système de Visualisation</u> :** le module **graphic** dessine l'état courant de la simulation à l'aide des entités élémentaires gérées par le module **shape**. Le module **graphic** rassemble les dépendances vis-à-vis de GTKmm pour faire les dessins. On autorise l'inclusion de son interface dans l'interface **shape.h** pour que le Modèle puisse choisir les couleur prédéfinies dans **graphic.h** .





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)[1].

- Niveau intermédiaire: il faut au moins considérer un module distinct pour les entités de la simulation. Le module **lifeform** doit être conçu comme une hiérarchie de classes pour intégrer les 3 sortes d'entités (nous accepterons une version simplifiée, sans hiérarchie de classe, pour le **rendu1** seulement).

---

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

- ***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 shape pour les calculs d'inclusion ou de collision

7.3.1 Indépendance de shape

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 notre projet. Pour cette raison **AUCUN** des noms de types/concepts du niveau supérieur MODELE ne doit apparaître dans **shape.** On y trouvera des fonctions effectuant les tests d'inclusion et de collision utiles pour les niveaux supérieurs (cf Rendu1).

7.3.2 Relation « Possède-un » entre la hiérarchie de classes du module lifeform et les types de shape

Les classes du MODELE devront utiliser les types mis à disposition dans l'interface de **shape** MAIS SEULEMENT en utilisant la relation « ***possède-un*** » plutôt que la relation « ***est-un*** ». C'est-à-dire que, par exemple, une entité corail possède un ou plusieurs attributs Segment pour ses calculs géométriques et l'affichage mais, en vertu du principe de *séparation des fonctionnalités* le module **shape** n'a pas vocation à servir de classe parente pour l'ensemble des applications qui utilisent ce module.

7.3.3 Type concret S2d et autres types de shape

Nous demandons d'implémenter le type **S2d** qui permet de modéliser une position et/ou un vecteur soit avec cette approche :

```
struct S2d {double x=0.; double y=0.;}; //plus robuste aux bugs
```

ou avec cette approche :

```
typedef array<double,2> S2d;
enum {X,Y}// quand on veut accéder explicitement à une coordonnée
```

Pour les entités de segment, cercle ou carré, nous considérons qu'ils sont suffisamment simples et de bas-niveau pour vous autoriser à les créer à l'aide de **struct**. Alternativement, vous pouvez aussi les dériver d'une classe parente possédant un attribut **S2d** pour représenter la position d'un point.

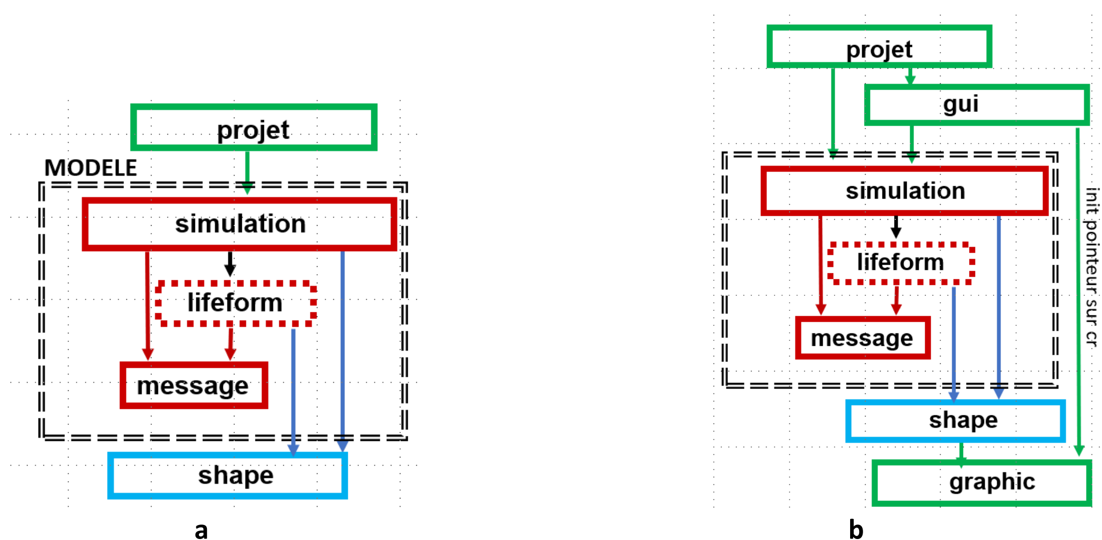


Figure 9 : (a) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (rendu1); (b) modules et dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendus 2 et 3)

7.4 Module graphique de bas-niveau (graphic)

A partir du rendu2 le module **shape** va aussi offrir des fonctions de dessin pour les types qui y sont définis dont **S2d**. Cependant le module **shape** 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 ; son interface **graphic.h** met à disposition des symboles pour définir une table de couleurs prédéfinies. Comme indiqué en section 6, on autorise **shape.h** d'inclure **graphic.h** pour avoir accès à ces symboles de style/couleur dans le MODELE.

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 première version de vos 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.

La simulation devra gérer seulement la génération conditionnelle des algues au cours du temps pour le rendu2.

Un rapport devra décrire les choix de structures de donnée et préciser l'ordre de complexité de la détection de collision.

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 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 compléter celui du rendu2 et illustrer quelques simulations.

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.

#include "shape.h " // nécessaire pour utiliser **epsil_zero** et disposer des symboles de **graphic.h**
enum Statut_cor {**DEAD, ALIVE**} ;
enum Dir_rot_cor {**TRIGO, INVTRIGO**} ;
enum Statut_sca {**LIBRE, MANGE**} ;

constexpr double **max**(256.) ;

constexpr unsigned **r_alg**(1) ;
constexpr unsigned **d_cor**(3) ;
constexpr unsigned **r_sca**(3) ;
constexpr unsigned **r_sca_repro**(10) ;
constexpr unsigned **delta_r_sca**(1) ;
constexpr unsigned **delta_l** (4) ;
constexpr unsigned **l_repro** (40) ;
constexpr unsigned **l_seg_interne** (28) ;
constexpr double **delta_rot**(0.0625) ; // en rd = env. 3.5°

constexpr double **alg_birth_rate**(0.5) ;

constexpr unsigned **max_life_alg**(500) ;
constexpr unsigned **max_life_cor(**1500) ;
constexpr unsigned **max_life_sca(**2000) ;

ANNEXE B : constantes génériques définies dans shape.h

constexpr double **epsil_zero**(0.5) ;

ANNEXE C : constante destinée au sous-système de Contrôle

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