

Agar.io Simulator

Steven Liatti

Orestis Malaspinas

Vincent Pilloux

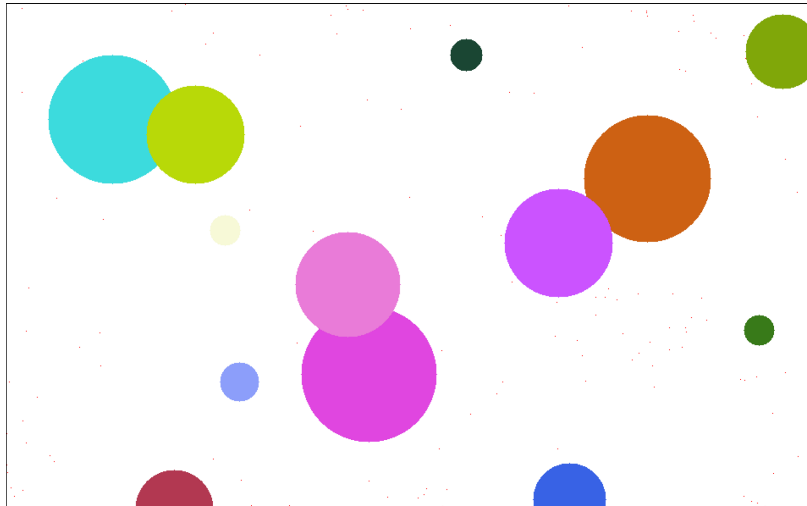


Figure 1: Screenshot du jeu

1 But du travail

Le but de ce travail pratique est de réaliser une simulation du jeu agar.io. Dans ce jeu 2D dans le navigateur, en réseau, chaque joueur contrôle, à la souris, le déplacement d’une cellule, représentée par un disque de couleur. En début de partie, la cellule apparaît sur le domaine de jeu à une position aléatoire. Ce domaine a des dimensions finies, de sorte qu’une cellule ne peut en “sortir”. La “loi de la nature” s’applique aux cellules : une cellule plus grosse peut “gober” une plus petite. En contrepartie, une cellule plus petite se déplace plus rapidement qu’une grosse. Pour gagner en masse, une cellule doit soit manger des cellules plus petites qu’elle-même, soit manger de la nourriture disséminée sur le domaine de jeu de manière aléatoire. Le but étant bien entendu de manger les autres, rester en vie et d’être la plus grosse cellule au classement. L’objectif du TP est de réaliser une simulation simplifiée du jeu original, comme vu en fig. 1. Les cellules seront contrôlées par ordinateur et il faudra gérer l’affichage, la gestion du clavier et la logique du jeu de manière concurrente.

2 Règles du simulateur

- Le domaine de jeu est rectangulaire et défini par une largeur et une hauteur fixes.
- Sur ce domaine, sont présentes des particules de nourriture qui sont caractérisées uniquement par leur position. Si une cellule rencontre une particule de nourriture, la cellule gagne en masse et la particule disparaît du domaine. La quantité de nourriture initiale est définie au début du programme. Selon une probabilité donnée, la nourriture “réapparaît” sur des points aléatoires du domaine.

- Chaque cellule est définie par un état (“morte” ou “vivante”), une couleur (fixe tout au long de la vie de la cellule), une direction de déplacement, une position sur le domaine de jeu et une masse (initialement comprise entre 5 et 8). De cette masse, sont déduits le rayon du disque la représentant graphiquement et sa vitesse de déplacement. Elle se déplace de manière aléatoire dans une direction donnée, selon les huit points cardinaux principaux, et change de direction après un certain temps, de manière aléatoire également. Si son centre entre en collision avec les bords du domaine de jeu, elle doit rebondir (voir explications plus bas). Si elle mange une particule de nourriture, elle gagne une unité de masse, mais perd en vitesse. Elle ne peut augmenter que d’une unité à la fois, c’est à dire manger une seule particule de nourriture à chaque déplacement. Si elle mange une autre cellule, elle récupère la masse de cette dernière. Si elle est mangée par une autre cellule, son état passe à “mort” et au bout d’un certain temps, elle réapparaît sur le domaine, de manière analogue aux conditions initiales. La masse maximale sera définie à 10% de la hauteur du domaine de jeu.

2.1 Relations entre masse, rayon et vitesse des cellules

Pour ne pas faire “grossir” vos cellules trop rapidement, nous vous recommandons de déduire le rayon d’une cellule en fonction de sa masse par la fonction suivante :

$$r = 4 + 6\sqrt{m}$$

où r est le rayon et m la masse.

De manière analogue, pour obtenir une vitesse inversement proportionnelle, mais lissée, en fonction de la masse, votre professeur de maths favori vous recommande d’utiliser la fonction sigmoïde suivante :

$$s = \alpha + \frac{\beta}{1 + e^{\gamma(m-\delta)}}$$

où s est la vitesse et m la masse.

La fig. 2 illustre cette fonction et ses paramètres.

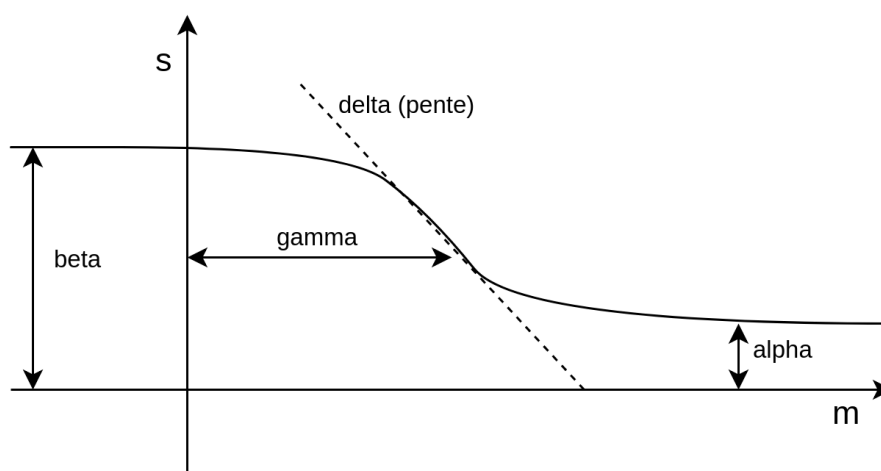


Figure 2: Graphe de la vitesse en fonction de la masse

2.2 Conditions de collisions

Les trois types de collisions sont les suivants :

- Cellule - nourriture : la position d'une particule se trouve en contact avec la surface d'une cellule ou à l'intérieur de celle-ci.
- Cellule - bords du domaine : si le centre d'une cellule entre en collision avec les bords du domaine de jeu, elle doit effectuer une réflexion spéculaire (comme sur la fig. 3). **Important** : dans ce cas de figure, il faut changer uniquement de direction mais pas la position, car il y a un risque de "blocage" de la cellule sur le bord du domaine de jeu.

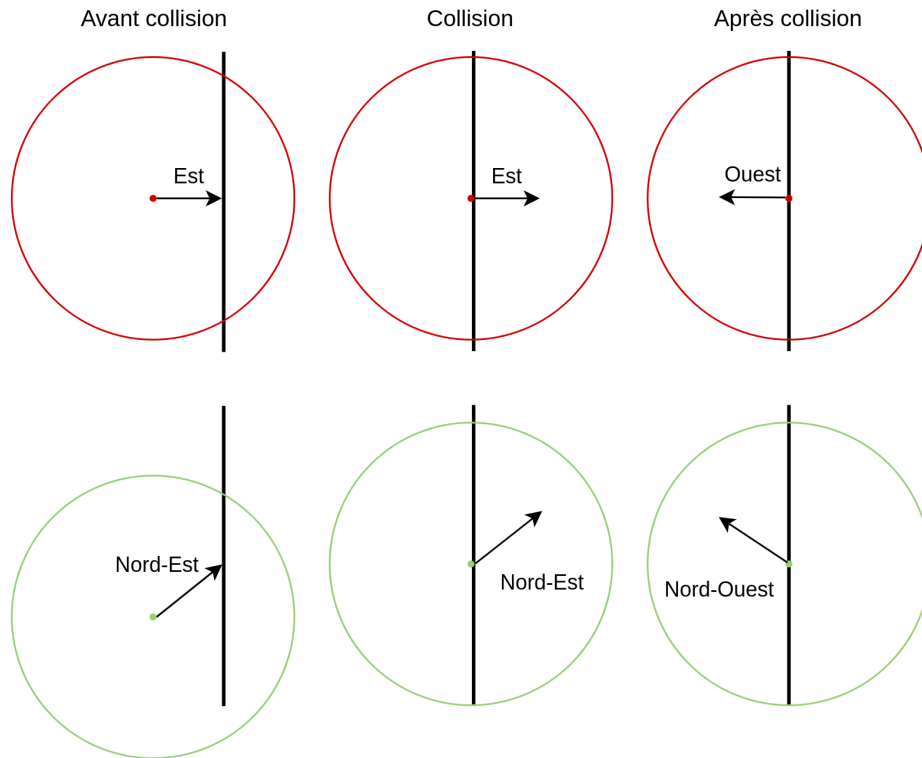


Figure 3: Cas de collisions avec les bords

- Cellule - cellule : la position du centre du disque représentant la plus petite cellule entre en contact avec la surface ou se retrouve à l'intérieur de la plus grosse cellule, comme sur la fig. 4.

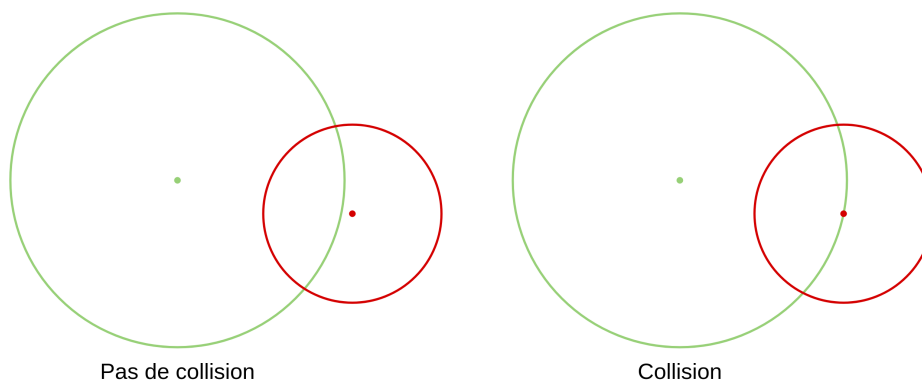


Figure 4: Cas de non collision et de collision entre deux cellules

3 Architecture multi-threadée

Le programme à développer est basé sur une architecture multi-threadée organisée de la manière suivante :

- Le thread principal (fonction `main`) s'occupe seulement de gérer les arguments de la ligne de commande, initialiser les structures de données, et créer les threads.
- Un thread est dédié à la gestion du clavier.
- Un ou plusieurs threads *travailleurs* sont dédiés au calcul du prochain état des cellules : nouvelles positions (en fonction de la vitesse de la cellule) et **détection de collisions** entre certains éléments. Ces événements de collisions potentielles sont stockés dans une seule structure de données concurrente (file, pile, etc.). Il est **important** que tous les threads *travailleurs* déplacent leurs cellules assignées, puis s'attendent **avant** de réaliser la détection de collisions. Un thread traite une ou plusieurs cellules (de manière équitable).
- Un seul thread *collisionneur* est dédié au **traitement des collisions** entre certains éléments (lecture des événements de la liste concurrente) et à **l'affichage**.

Concernant les collisions, deux variantes sont possibles :

1. Chaque thread *travailleur* réalise uniquement la détection de collisions entre cellules et entre cellules et particules de nourriture, puis délègue totalement le traitement de ces collisions à l'unique thread *collisionneur*.
2. Chaque thread *travailleur* détecte toutes les collisions, mais traite les collisions entre cellules et particules de nourriture. L'unique thread *collisionneur* ne traite en conséquences uniquement que les collisions entre cellules.

4 Cahier des charges

- Le programme à développer sera nommé **agario** et sa syntaxe est la suivante :

```
agario <width> <height> <seed> <food> <dir> <res> <nf> <freq> <workers> <cells>  
Exemple : agario 960 600 0 0.001 0.01 0.05 0.1 30 4 10
```

- La taille du domaine à créer est donnée par les arguments **width** et **height** spécifiés sur la ligne de commande, entiers plus grands ou égaux à 100.
- L'état initial du domaine (absence/présence de particules de nourriture) est calculé aléatoirement selon les deux arguments **seed** et **food** spécifiés sur la ligne de commande :
 - **seed** (entier) est la graine à utiliser par le générateur de nombres aléatoires.
 - **food** (valeur décimale dans le *range* [0..1]) représente la proportion de cases du domaine occupées par des particules de nourriture. Par exemple, 0.1 représente une surface occupée de 10% par de la nourriture.
- Les arguments **dir**, **res** et **nf** (valeurs décimales dans le *range* [0..1]) représentent les probabilités de :
 - **dir**: changement de direction d'une cellule (vivante).
 - **res**: pour une cellule morte, réapparaître (testé à chaque frame).
 - **nf** (pour "new food"): faire apparaître une nouvelle particule de nourriture sur une position qui ne comprend pas déjà de nourriture.
- **freq** est un entier (> 0), représentant la fréquence d'affichage en Hz.
- **workers** est un entier (≥ 1) donnant le nombre de threads *travailleurs*.
- **cells** est un entier (≥ 1 et \geq **workers**) représentant le nombre de cellules.
- Un **makefile** devra être présent à la racine et ayant comme première règle la compilation de votre programme produisant l'exécutable.

- Le thread d’affichage s’occupera de créer le contexte graphique (fonction `gfx_create`) et d’afficher le jeu à la fréquence spécifiée (fonction `gfx_present`) par l’argument `freq` sur la ligne de commande. Attention à ce que l’affichage du domaine ne soit réalisé qu’une fois celui-ci **entièrement** mis à jour !
- L’affichage doit être réalisé avec les fonctions fournies (`gfx_create`, `gfx_present`, etc.); cf. section suivante.
- Le thread responsable de la gestion du clavier testera, à la fréquence de 50 Hz, si une touche a été pressée. Le programme se terminera, proprement, une fois la touche d’échappement (ESC) pressée.
- La division du travail entre les threads *travailleurs* doit être la plus équitable possible par rapport au nombre de cellules.
- Seules les primitives de synchronisation vues en cours sont autorisées.
- **Toute attente active est prohibée.**
- Aucune variable globale n’est autorisée. À noter qu’il est permis d’utiliser des constantes globales (déclarées via la directive `#define` ou le mot-clé `const`).
- **Attention** : dans les mesures de temps, n’oubliez pas de tenir compte du temps écoulé entre deux mesures successives. Par exemple, si une routine *R* doit être appelée à la fréquence de 1 Hz, réaliser un `sleep` de 1 seconde est incorrect car il est nécessaire de tenir compte du temps d’exécution de la routine *R*.

5 Informations utiles

5.1 Options de compilation

Nous vous conseillons de compiler votre code avec gcc et les options de compilation suivantes :

```
-g -std=c11 -Wall -Wextra -fsanitize=address -fsanitize=leak -fsanitize=undefined
```

De plus, si vous utilisez des barrières dans votre code, vous devez inclure cette directive :

```
#define _GNU_SOURCE
```

5.2 Affichage graphique

Pour réaliser le rendu graphique de votre programme, vous **devez** utiliser la librairie `gfx` incluse dans le repository du TP. L’entête de la fonction `gfx_drawcircle` vous est fournie dans `gfx`, à vous de l’implémenter.

5.3 Mesure du temps d’exécution

Les mesures du temps d’exécution peuvent être effectuées avec la fonction `clock_gettime` de la librairie `<time.h>` comme montré ci-dessous :

```
struct timespec start, finish;
clock_gettime(CLOCK_MONOTONIC, &start);
... // code à mesurer
clock_gettime(CLOCK_MONOTONIC, &finish);
double elapsed = (finish.tv_sec - start.tv_sec) * 1e6;
elapsed += (finish.tv_nsec - start.tv_nsec) / 1e3;
```

À noter que selon les versions de gcc, le code ci-dessus requiert la librairie `rt`. Pour ce faire, passez `-lrt` à gcc au moment de l’édition des liens.

5.4 Schéma du fil d'exécution et de synchronisation

Pour appuyer vos explications à l'oral, vous devrez produire un schéma du fil d'exécution du code et de la synchronisation entre threads, à rendre avec votre code. La fig. 5 illustre un exemple d'un tel schéma. La syntaxe à utiliser est libre, mais les mécanismes de synchronisation doivent clairement apparaître.

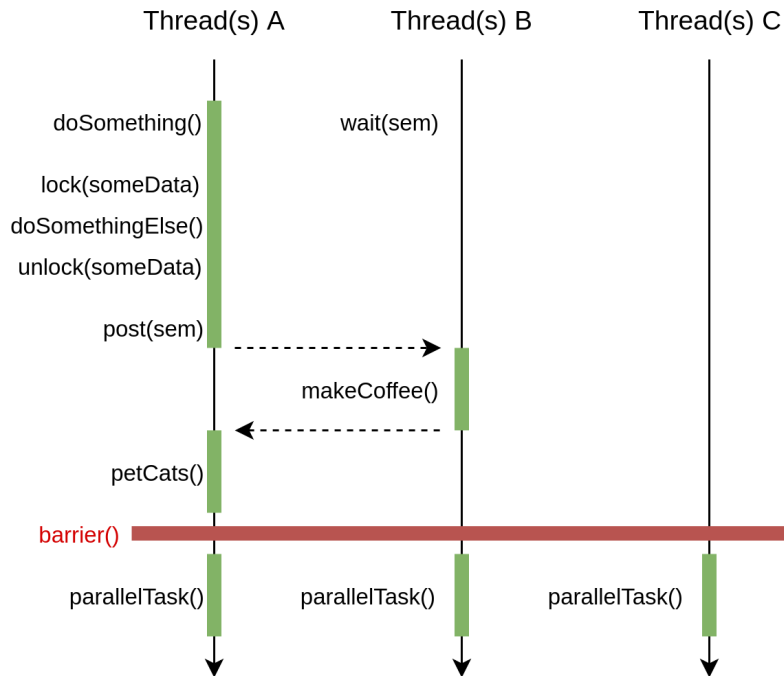


Figure 5: Exemple de schéma du fil d'exécution et de synchronisation d'un programme multi threads

6 Travail à rendre

N'oubliez pas de lire attentivement le document “*Consignes Travaux Pratiques.pdf*” disponible sur la page CyberLearn du cours. Les consignes supplémentaires et spécifiques pour ce travail sont les suivantes :

- Ce travail sera réalisé par **groupes de deux**. Si le nombre d'étudiants est impair, il n'y aura qu'un seul groupe de trois.
- Vous serez forcés d'utiliser **git et gitedu.hesge.ch**. Vous devrez “forker” ce repository et suivre à la lettre la procédure contenue dans le README.md, sous peine de pénalités. Nous exigeons qu'au minimum une version (donc un commit) soit réalisée par séance de TP, et ce jusqu'au rendu du travail.
- Vous devrez produire un schéma du fil d'exécution et synchronisation entre threads (dans un format d'images usuel), comme illustré plus haut, qui vous servira de support pour l'interrogation orale entre autres.
- Le rendu du travail est fixé au **dimanche 2 juin 2019 à 23h30** (la version antérieure ou égale à cette date sera récupérée).
- Suite au rendu, vous devrez effectuer une présentation orale de votre travail le **mardi 4 juin 2019**. La note sera une combinaison du code rendu et de la présentation.