Image Vision Interaction

SCI

Multi Agent Systems

Quentin BRAULT

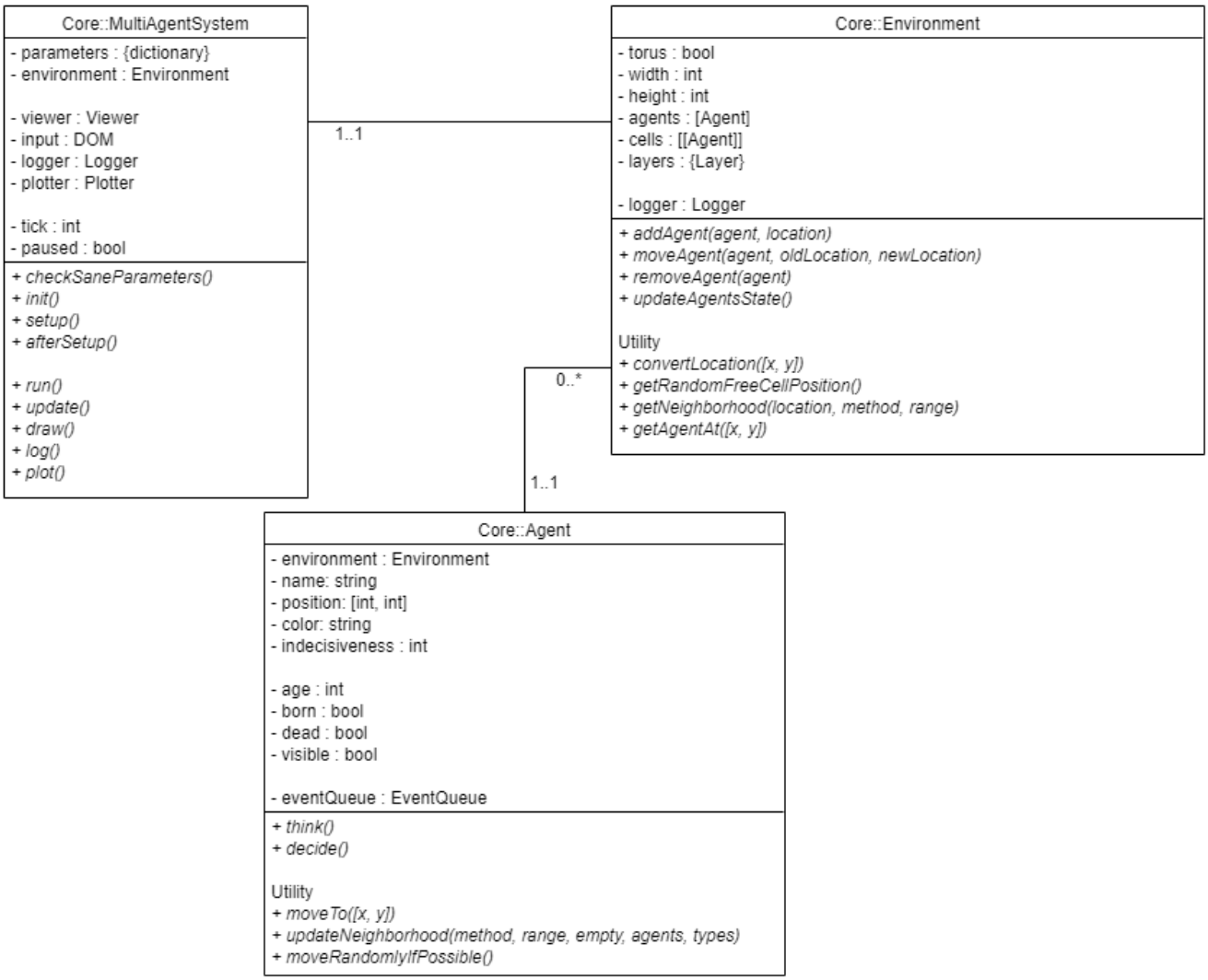
Table des matières

Introduction

L'architecture et les implémentations ont été réalisées en **JavaScript**.
J'ai choisi Javascript car je voulais pouvoir facilement partager le code et le résultat et je voulais des primitives d'affichage facile d'utilisation. Comme **JS** s'exécute dans un naviguateur sur n'importe quelle machine et ne demande aucune installation, cela semblait parfait.
Et au final, malgrès le fait que le language ne soit pas très orienté objet, j'ai réussis à obtenir des dépendances raisonables et un code propre.

Architecture

Architecture avec attributs et méthodes principales



MultiAgentSystem

La classe **MultiAgentSystem** va créer l'environnement et les agents, et gérer la mise à jour et l'affichage.
J'ai fais abstraction de l'affichage dans une classe **Viewer**, du logging dans une classe **Logger** et de l'affichage des courbes dans une classe **Plotter**.

Le **input** est un élément du DOM sur lequel on va placer des listeners afin de gérer les inputs utilisateurs. Les **parameters** qui proviennent du **fichier de configuration** régissent en grande partie les comportements de la classe.

La méthode **checkSaneParameters()** vérifie l'intégrité des paramètres du fichier de configuration. Elle doit être appellée et potentiellement étendue dans chaque implémentation de **MultiAgentSystem**.
La méthode **init()** est appellé une seule fois, à la création. Cette méthode initialise les attributs qui n'ont pas pu être passés en paramètre dû à une dépendance sur le fichier de configuration. Par exemple, elle initialise le **randomizer** (générateur de nombres aléatoires) et le **scheduler** en fonction du **fichier de configuration**.
La méthode **setup()** est appellée à chaque remise à zéro de la simulation. Elle doit être étendue par chaque implémentation afin de créer l'environnement et les agents.
La méthode **afterSetup()** est appellée juste après **setup()** et remet à zéro les attribus qui dépendent sur le fait d'avoir un environnement créé et près à démarrer.

La méthode **run()** est la boucle infinie d'éxécution. Elle s'appelle récursivement en essayant de respecter le délais du fichier de configuration. Elle appelle les méthodes **update()**, **draw()**, **log()**, **plot()** dans cet ordre. La méthode **update()** demande au **scheduler** de s'occuper des agents, puis elle appelle la méthode **updateAgentsState()** de l'environnement afin de régulariser les agents morts ou nés.

Environment

La classe **Environment** contient une grille de cellules 2D qui peuvent être occupées par 0 ou 1 agent chacune. L'environnement peut être torique ou non et sa largeur et sa hauteur sont paramétrables également.
L'environnement peut contenir des **layers** qui sont une abstraction de grilles 2D avec une référence à l'environnement, permettant de calquer sur l'environnement des informations supplémentaires sur ses cellules. Par exemple dans l'implémentation Pacman, il y a une **Layer** qui calcule pour chaque cellule la distance au joueur.

La méthode **addAgent()** ajoute un agent à l'environnement et le marque comme **born** (nouveau-né).
La méthode **removeAgent()** retire un agent de l'environnement et le marque comme **dead** (mort).
La méthode **moveAgent()** déplace un agent d'une cellule à une autre.
La méthode **updateAgentsState()** est appellée à la fin d'un tick et regarde tous les agents. Elle se débarasse définitivement des agents morts et retire la marque des nouveaux-nés.

Agent

La classe **Agent** contient une référence à l'environnement dans lequel l'agent est situé. L'agent possède des attributs **name**, **color** et **position**.
L'attribut **visible** détermine si l'agent est affiché ou non. Il possède optionellement un attribut **eventQueue** qui permet par exemple dans le cas d'un agent controllable par input de lire une pile d'instructions à éxécuter.
Il possède un attribut **indecisiveness** (indécision) qui détermine par le biais de la méthode **think()** combien de ticks séparent les appels à **decide()**. C'est une sorte d'inverse de la vistesse, plus **indecisiveness** est grand, plus l'agent met de ticks à se décider.
La méthode **decide()** contient toute la logique à éxécuter et se fait appeller par la méthode **execute()** du **scheduler** qui elle-même se fait appeller à chaque tick.

```
class RandomSequenceScheduler extends Scheduler {
    execute(agents) {
        RNG.shuffle(agents);
        for (let i = 0; i < agents.length; i++) {
            let agent = agents[i];
            // isActive() returns true if the agent is not born nor dead
            if (agent.isActive()) {
                // think() returns true if the agent is ready to decide
                if (agent.think()) {
                    agent.decide();
                }
            }
        }
    }
}
```

La méthode **moveTo()** met à jours les attributs correspondant à la position et signale le déplacement à l'environnement.
La méthode **updateNeighborhood()** est une méthode utilitaire qui va interroger l'environnement et récupérer les agents voisins (voisinage Moore/Neumann et distance paramétrables) et leurs positions. La méthode permet de trouver des agents quelconques, des cellules vides ou bien des agents spécifiques, qu'elle met dans les attributs **agentsNeighborhood**, **emptyNeighborhood** et **specificNeighborhood** respectivement, ce qui permet ensuite à l'agent de faire les calculs nécessaires à partir de ces données.

Créer son implémentation

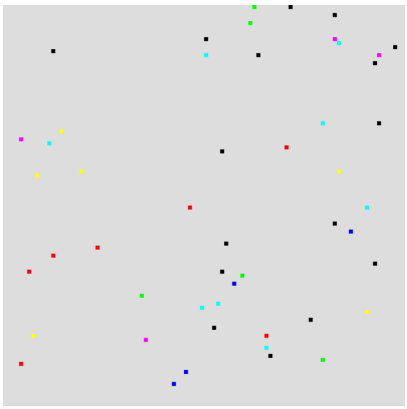
Pour créer une implémentation, il suffit de dupliquer le dossier `Template` et de modifier les fichiers à l'intérieur à sa convenance. Ces fichiers sont :

- Un fichier `TemplateMultiAgentSystem.js` qui contient une classe du même nom qui extends **MultiAgentSystem**. C'est ici que doivent être créés l'environnement et les agents, et que le fichier de configuration est vérifié.
- Un fichier `TemplateEnvironment.js` qui contient une classe du même nom qui extends **Environment**. C'est ici que doit aller le code spécifique à l'environnement, c'est rare qu'il y ait beaucoup de code ici car la classe **Environment** est supposée être très flexible.
- Un fichier `TemplateAgent.js` qui contient au moins une classe du même nom qui extends **Agent**. D'autres classes **XXXTemplateAgent** qui extends **TemplateAgent** peuvent être ajoutées à ce fichier.
- Un fichier `template.config.js` dans lequel sont renseignés tous les paramètres de la simulation. Toutes les valeurs des paramètres peuvent être modifiées, mais c'est dans la variable **extraConfiguration** que devraient être ajoutés les paramètres propres à cette implémentation.
- Un fichier `template.js` qui instancie et lance le système multi-agents.
- Une page `index.html` qui importe tous ces fichiers JavaScript ainsi que les dépendances. C'est ici que vous pouvez modifier la taille du Canvas pour l'affichage, ajouter des instructions ou bien ajouter d'autres Canvas pour afficher des plots.

Lancer les implémentations

Pour lancer une implémentation, il suffit d'ouvrir `index.html` dans un navigateur. Tout devrait fonctionner au moins sous Chrome.

Implémentation 1 : Particles



Configuration

- **nbBalls**: Nombre de balles
- **bounceBehavior**: Comportement des balles quand elles rebondissent entre-elles. SWAP; REVERSE; WAIT
- **colors.balls**: Couleurs aléatoire des balles au départ.
- **colors.hit**: Couleur des balles ayant eu une collision avec une autre.

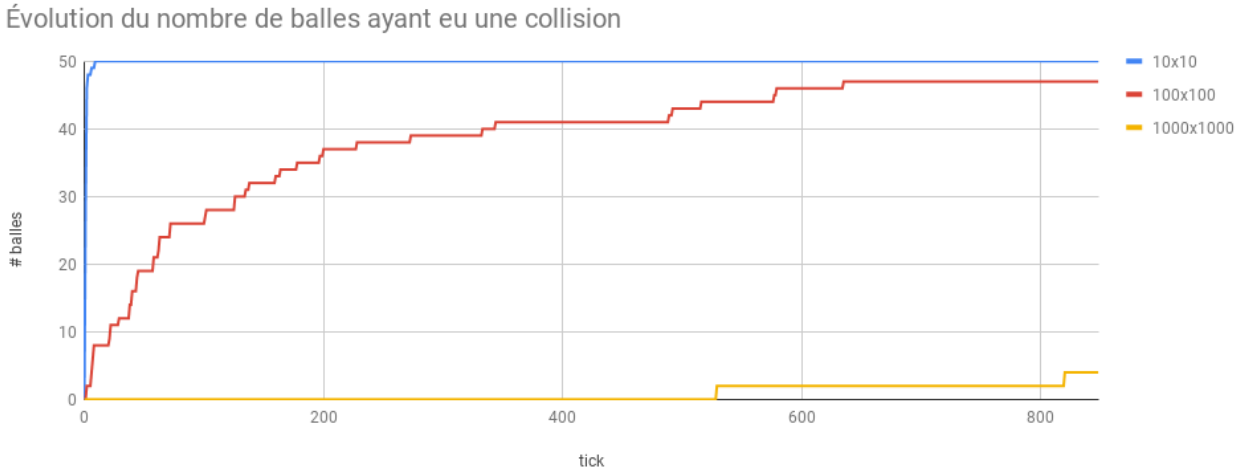
Questions

Question 1

La simulation tourne à 1 fps avec environ 500 000 agents et l'affichage activé sans grille sur mon ordinateur. La performance va dépendre beaucoup de la machine sur laquelle le programme tourne.

Question 2

Environnements non-toriques, de tailles 10x10, 100x100 et 1000x1000 avec 50 balles, scheduler équitable



Question 3

Avec le comportement "Attendre si une balle me bloque" (**bounceBehavior = WAIT**), les balles se déplacent plus lentement et si deux balles avec des directions opposées se rentre dedans elles restent bloquées en attendant l'une l'autre que l'autre bouge.
Avec le comportement "Rebondir en inversant ma direction" (**bounceBehavior = REVERSE**), si une balle va dans la même direction qu'une autre balle qui est dans la case devant elle, elle rebondira si son tour arrive avant celle de devant, mais elle ne rebondira pas sinon, car la balle se sera déplacée d'une case de plus entre temps.
Avec le comportement "Rebondir en inversant ma direction avec l'autre balle" (**bounceBehavior = SWAP**), on obtient une vrai impression de rebond.

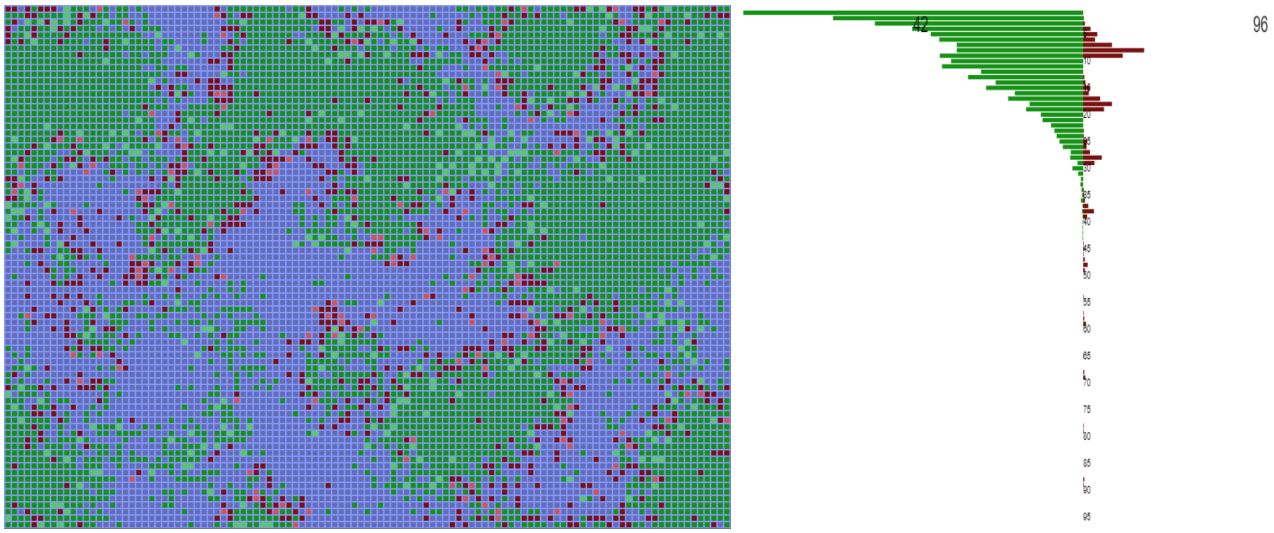
Difficultés techniques & limites

Dans la première version du TP1 il n'y avait presque pas d'architecture, et j'ai eu plein de problèmes de comportements étranges dûs à un code mal organisé. Lorsque j'ai réimplémenté le TP1 avec l'architecture complète cela m'a pris 15 minutes.

Améliorations possibles

Ajout d'un comportement de rebond où chaque balle calcule le vecteur milieu des deux directions comme leur nouvelle direction.
Dans la première version j'avais ajouté un tracé de déplacement derrière les balles, ça devrait être facile à réimplémenter avec les **Layer**.

Implémentation 2 : Wator



Configuration

- **nbFish**: Nombre de poissons
- **nbSharks**: Nombre de requins
- **fishGestation**: Temps (minimum si **randomGestation**) nécessaire pour faire naître un poisson
- **sharkGestation**: Temps (minimum si **randomGestation**) nécessaire pour faire naître un requin
- **sharkStarvation**: Temps (minimum si **randomStarvation**) nécessaire pour qu'un requin meure de faim
- **babySharkFood**: Temps (minimum si **randomStarvation**) nécessaire pour qu'un requin qui n'a encore jamais mangé meure
- **randomGestation**: Si vrai, le temps de gestation est aléatoire
- **fishGestationRange**: Variabilité du temps nécessaire pour faire naître un poisson
- **sharkGestationRange**: Variabilité du temps nécessaire pour faire naître un requin
- **randomStarvation**: Si vrai, le temps pour mourir de faim est aléatoire
- **sharkStarvationRange**: Variabilité du temps nécessaire pour qu'un requin meure de faim
- **babySharkFoodRange**: Variabilité du temps nécessaire pour qu'un requin qui n'a encore jamais mangé meure
- **oneActionPerTurn**: Si vrai, les agents ne peuvent faire d'une action par tour parmis bouger, manger (et bouger si **moveWhenEating**) et se reproduire.
- **sharkReproduceWhenEating**: Si vrai, les requins peuvent se reproduire quand ils mangent
- **sharkReproduceWhenMoving**: Si vrai, les requins peuvent se reproduire quand ils se déplacent sans manger
- **moveWhenEating**: Si vrai, les agents se déplacent sur leur proie quand ils mangent
- **fishSuffocate**: Si vrai, les poissons peuvent mourir par asphyxie
- **fishBreath**: Le nombre de cases voisines libres nécessaires pour qu'un poisson respire
- **explosivePregnancy**: Si vrai, quand un agent fini sa gestation mais n'est pas capable de créer un bébé, il meure et ne laisse pas d'enfant
- **sharkGestateOnlyWhenEating**: Si vrai, les requins ne comptent comme de la gestation que les tours où ils mangent

Questions

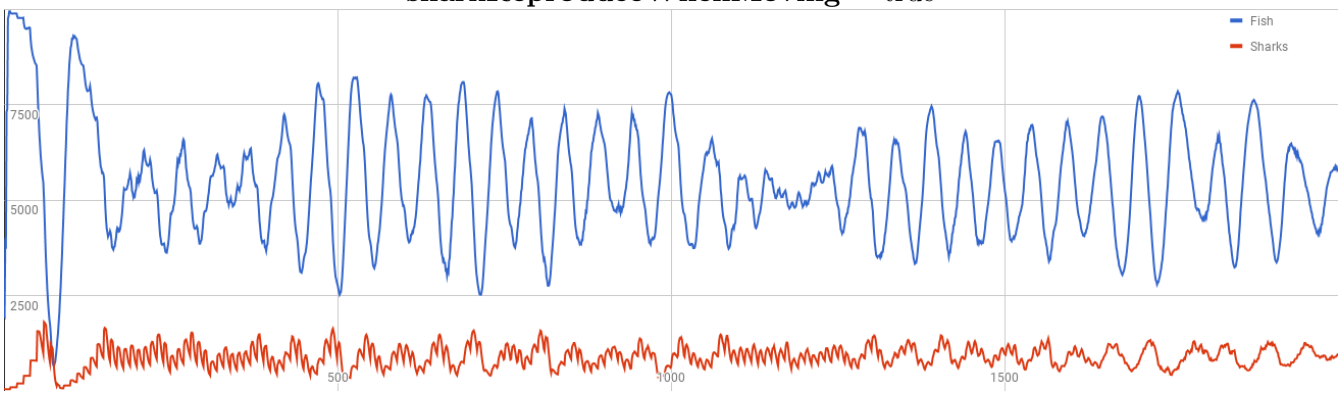
Question 1

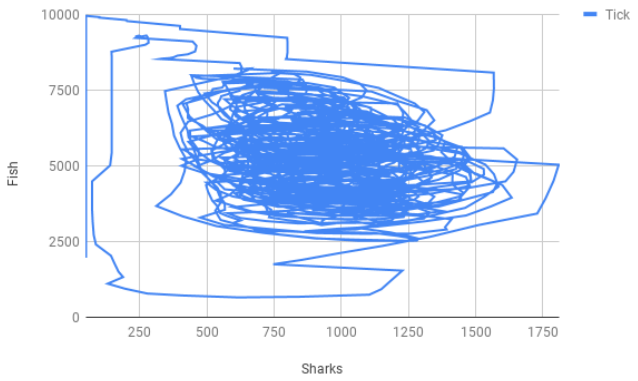
Plus l'aléatoire joue un rôle important, plus le risque qu'une simulation finisse en extermination est grand. Il faut également faire attention à ce que les requins ne naissent pas plus vite qu'ils meurent de faim sinon ils survivront même sans poissons. Pour palier à cela j'ai ajouté le paramètre **sharkGestateOnlyWhenEating** qui transforme le temps de gestation en un nombre de poissons à consommer pour accoucher.

Question 2

Je vais comparer les comportements avec les mêmes paramètres simples. Environnement 100x100 torique, pas d'aléatoire, **fishSuffocate**, **explosivePregnancy**, **sharkGestateOnlyWhenEating** à faux et **moveWhenEating** à vrai.
nbFish = 2000, **nbSharks** = 50, **fishGestation** = 2, **sharkGestation** = 10, **sharkStarvation** = 3, **babySharkFood** = 3

Situation initiale : **oneActionPerTurn** = false, **sharkReproduceWhenEating** = true, **sharkReproduceWhenMoving** = true

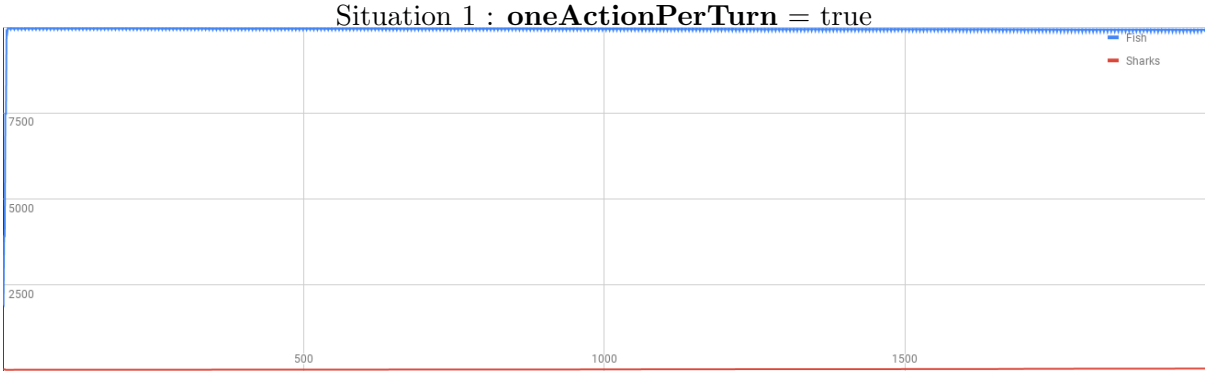




En observant la courbe on remarque qu'il y a une période de stabilisation au début de la simulation dû à une oscillation d'un trop grand nombre de poissons qui nourissent un trop grand nombre de requins qui mangent un trop grand nombre de poissons ce qui affame les requins, etc... Jusqu'à avoir une oscillation stable. En observant la simulation on remarque que les bancs de poissons sont définis par les requins qui les encerclent. Ces bancs rétrécissent au fur et à mesure que les requins les mangent mais les poissons se reproduisent plus vite et un autre banc se créé avant que le premier ne disparaisse.

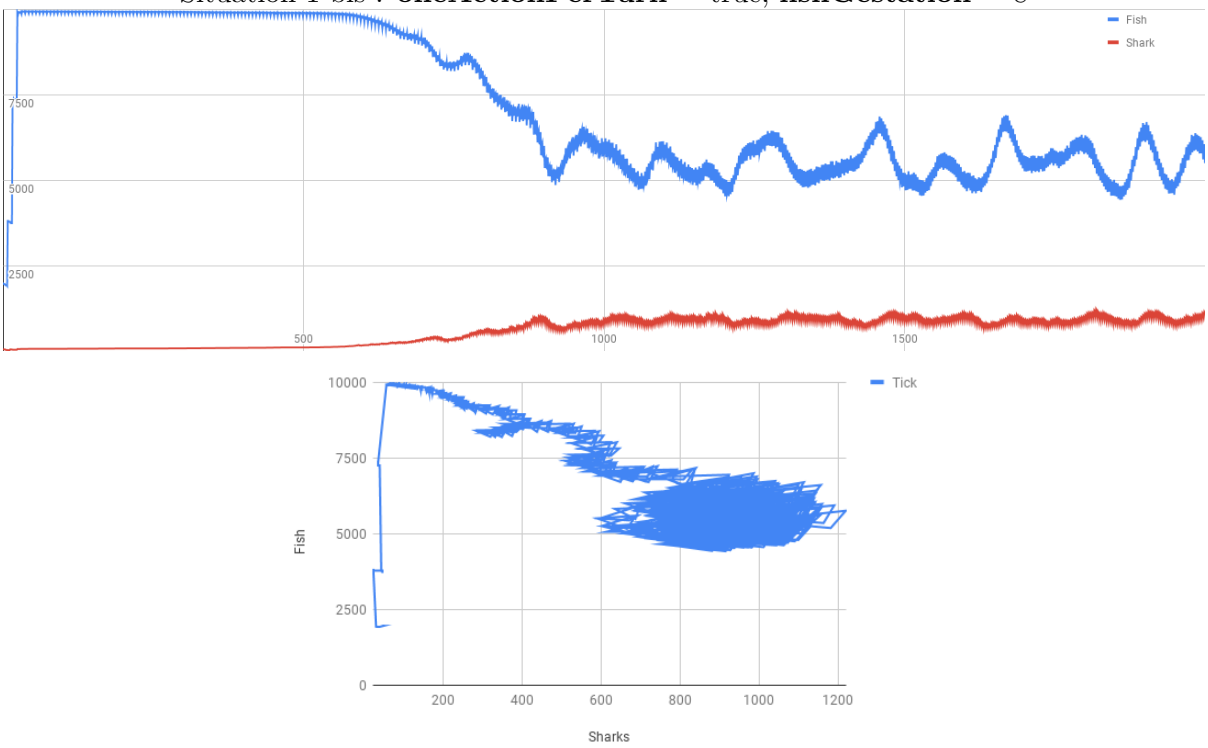
Dans la situation 1, dû au fait que les poissons n'ont que 2 actions et les requins 3, les poissons prennent rapidement tout l'espace disponible, et une fois cela fait, un requin n'a pas l'occasion d'accoucher car il n'y a jamais de place puisque la case qu'il a mangé est déjà remplie.
Les requins ne meurent donc jamais mais ne font aucun enfant.

Situation 1 : **oneActionPerTurn** = true



Si on compense cela en augmentant le temps de gestation des poissons, on arrive à une situation où éventuellement il y a suffisemment de requins les uns à coté des autres pour que certains mangent un poisson et d'autres accouchent, et à un moment donné des bancs de poissons commencent à apparaître. Cependant il faut un certain temps pour arriver à cet état car il faut que les requins soient bien synchronisés.

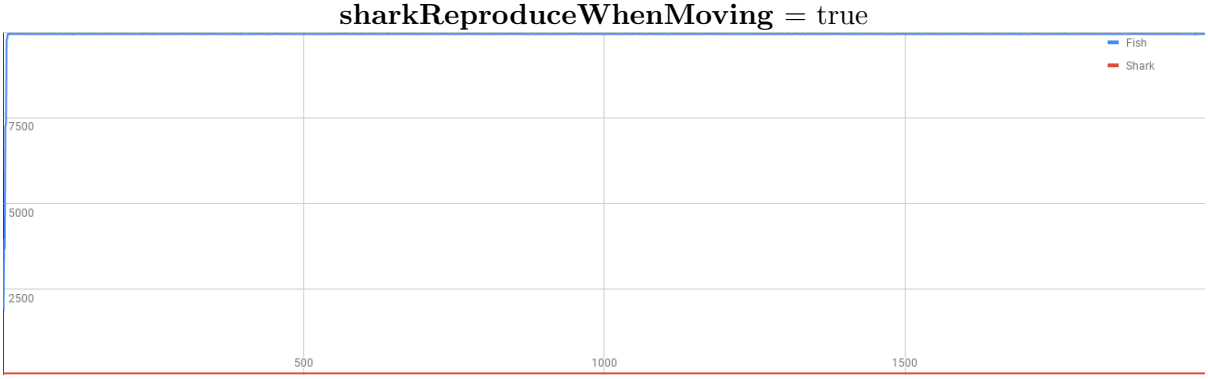
Situation 1 bis : **oneActionPerTurn** = true, **fishGestation** = 8



Dans cet état, on remarque une différence notable, les bancs de poissons se déplacent pendant une longue période sans requins autour, car ceux-ci n'ont pas encore eu le temps de les rattrapper, ce qui donne un côté plus naturel à la simulation par rapport à la situation initiale.

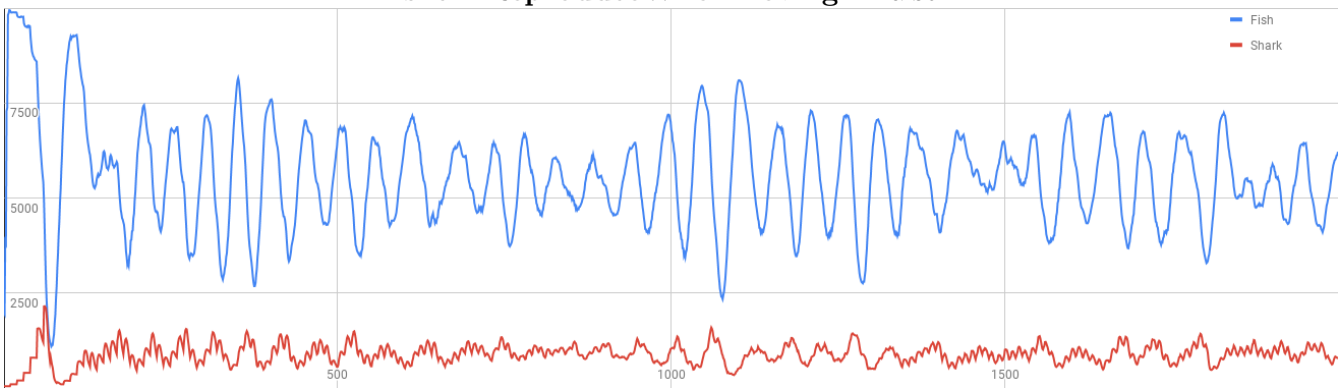
Un effet similaire se produit dans la situation 2, à la différence près que même en augmentant le temps de gestation des poissons, les requins n'arrivent jamais à se reproduire car ils mangent tout le temps et donc ne bougent jamais sans manger.

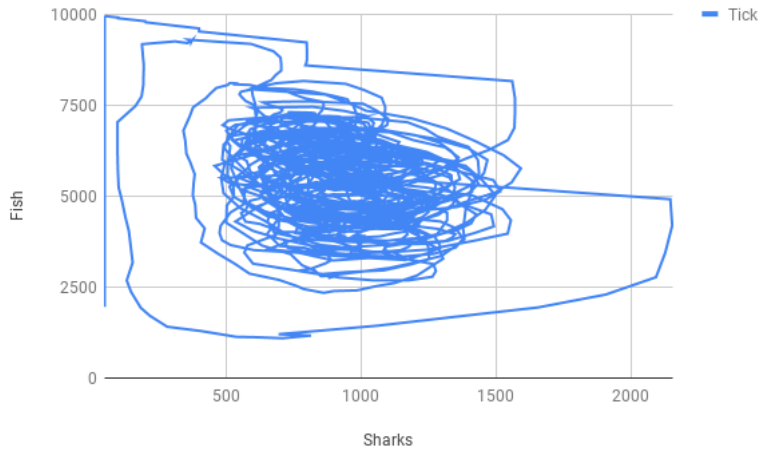
Situation 2 : **oneActionPerTurn** = false, **sharkReproduceWhenEating** = false, **sharkReproduceWhenMoving** = true



Dans la situation 3, les bébés requins naissent uniquement aux abords des bancs de poissons, mais à part cela il n'y a pas de différence notable avec la situation initiale sauf peut-être des variations légèrement plus douces comme le montre l'apparence plus arrondie de la courbe des poissons sur les requins.

Situation 3 : **oneActionPerTurn** = false, **sharkReproduceWhenEating** = true, **sharkReproduceWhenMoving** = false





Difficultés techniques & limites

La possibilité de supprimer et d'ajouter des agents est passée par plusieurs itérations. D'abord avec des listes d'agents séparés, mais les agents devaient à la fois immédiatement disparaître et apparaître dans l'environnement mais également ne pas jouer leur tour si ils sont morts ou nouveaux-nés. J'ai donc mis en place le système actuel où les agents sont marqués à leur mort et leur naissance, et répondent donc false à l'appel de **isActive()**.

Améliorations possibles

Comme il ya beaucoup de paramètres modifiables, les nombre de combinaisons de situations est gigantesque, et il est difficile de les comparer car il faut à chaque fois faire des petites modifications et redémarer la simulation pour voir ce que ça change.
Il faudrait donc pouvoir modifier certains paramètres pendant l'éxécution pour pouvoir plus facilement expérimenter.

Implémentation 3 : Pacman

Configuration

- **wallProbability**: Probabilité de placer un mur sur une case
- **nbGhosts**: Nombre de fantômes
- **nbPowerpellets**: Nombre de pastilles
- **playerSpeed**: Vitesse du joueur (1 - 100)
- **ghostSpeed**: Vitesse des fantômes (1 - 100)
- **powerpelletDurationFactor**: Facteur qui, multiplié par le nombre de cases de l'environnement, donne la durée de vie d'une pastille
- **powerpelletStrength**: Durée d'invincibilité donné par une pastille
- **overload**: Nombre de pastilles à avaler pour ouvrir une sortie

Comportements

J'ai remarqué que lorsque le joueur se déplace trop vite, on est beaucoup moins précis dans nos déplacement et cela rend le jeu beaucoup plus difficile même si les fantômes sont lents. Plus l'espace de jeu est grand, plus c'est facile d'échapper aux fantômes. Un jeu dans un environnement non-torique est presque impossible à gagner.

Difficultés techniques & limites

Lorsqu'il a fallut introduire la notion d'input, j'ai dû complètement repenser l'architecture pour en faire ce qu'elle est aujourd'hui. En JS il n'y a pas de compilation, donc c'est facile d'oublier quelque chose en réfactorant sur une grosse architecture comme celle-ci. Et lorsqu'il y a une erreur, elle n'est pas indiquée, il faut beaucoup creuser pour comprendre pourquoi une variable se retrouve avec la valeur undefined.
J'ai aussi passé beaucoup de temps à trouver une architecture convenable pour gérer les inputs, en effet, en JS on dépend du contexte de la page web pour récupérer les inputs. Or je voulais que le contexte web reste séparé de l'architecture tout en communiquant les events aux agents.
Au final la solution fut de créer la classe **EventQueue** qui est remplie d'event par des listeners placés sur le DOM par **MultiAgentSystem** et dont on passe une référence à l'agent pour qu'il puisse lire les events sur la pile.

Améliorations possibles

J'aurais bien voulu essayer de générer un labyrinthe aléatoire, voire même d'avoir une phase où des **WallAgent** se déplacent en suivant leur logique propre et créent un labyrinthe par émergence à la manière des tas des termites.
Il serait aussi intéressant d'avoir plusieurs niveau d'IA des fantômes. En cours de développement j'ai amélioré l'IA, mais je n'ai pas laissé l'option de jouer avec la version simple.
La version simple fait qu'un fantôme qui cherche la prochaine case parcourera les cases de son environnement toujours dans le même ordre, ce qui fait que si il y a 2 cases avec la même valeur (ce qui est souvent le cas), il prendra toujours la même (par exemple celle avec les x y les plus petits). Dans la version améliorée je mélange l'ordre de parcours des cases voisines, et le résultat est incroyablement imprévisible et cela rend très difficile l'attaque de fantôme en mode invincible.