Rapports S.C.I.

Tristan Camus & Arnaud Cojez

TP1 - S.M.A Architecture générale classique

Présentation

Le but de ce TP était de réaliser un système multi-agents permettant à des agents “particules” d’interagir avec les autres à travers des rebonds.

Pour réaliser ce système nous avons mis en place différentes classes dont les suivantes sont communes aux 3 TPs et se situent dans le dossiers core :

- *Core*, qui va s’occuper de créer les différentes instances des classes nécessaires à l'exécution du programme. Elle s’occupera de créer les classes du modèle : L’*Environnement*, qu’elle viendra peupler d’agents, ainsi que la *SMA*. Elle créera aussi la fenêtre ainsi que la vue représentée par la classe *View*. Elle chargera les paramètres stockés dans le fichier properties au format *JSON* et lancera le programme;
- *SMA*, située dans le dossier *core*, s’occupera d’effectuer le tour de parole (séquentiel juste) dans sa méthode *run()*;
- *Environment*, qui représente l’environnement dans lequel les agents interagissent sous forme d’une grille 2D, cette classe possède également la liste des agents présents dans le modèle;
- *Agent*, qui est une classe abstraite contenant le comportement commun à tous les agents manipulés dans les 3 TPs, les méthodes clés de cette classe sont *decide()* et *update()*, qui seront abordées dans la partie Comportement ci-dessous;
- *View*, qui s’occupe de dessiner les éléments graphique représentant le modèle sur la fenêtre. *View* hérite de *Observer* et observe l’objet observable *SMA* afin de dessiner le contenu de l’environnement de la *SMA*;
- *Observer*, qui permet de réagir à un signal envoyé par un *Observable*. La méthode *onReceive()* redirigera les appels de fonctions vers la fonction de callback associée au signal;
- *Observable*, à qui on abonnera des *Observers*, et qui leur enverra un signal spécifique (par exemple : “modelUpdated”) auquel l’*Observer* réagira, ou non.

En plus des classes communes, le TP Particules nécessite deux classes supplémentaires située dans le dossier “particles” :

- *Main*, cette classe hérite de la classe *Core,* elle définit la méthode *populate()* qui servira à peupler correctement l’environnement avec le bon type d’agents ainsi que la méthode *setDefaultProperties()* qui définit les paramètres par défaut du programme;
- *Particle*, qui contient les caractéristiques propres aux agents particules ainsi que le comportement de ceux-ci. Elle hérite de la classe *Agent*.

Choix de *Python*

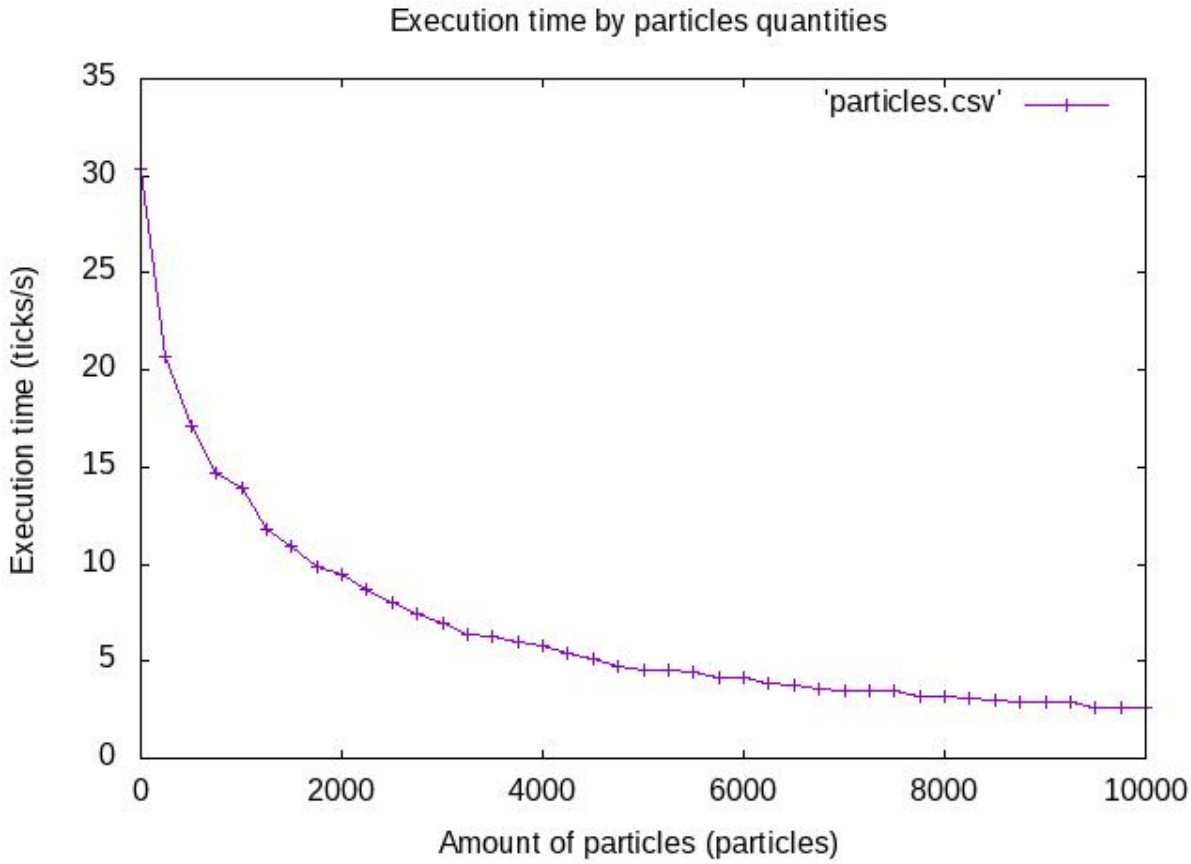
Nous avons choisi d’utiliser le langage *Python* pour développer notre projet, d’une part, pour renforcer notre connaissance sur ce langage que nous avons peu utilisé lors de nos études contrairement à Java et, d’autre part, pour le gain de temps de développement qu’il permet. Nous avons utilisé *Tkinter* pour l’affichage et la gestion des entrées utilisateurs.

Comportement

Le comportement d’une particule est défini grâce à la méthode *decide()*, située dans la classe *Particles*, qui permet à une particule, en fonction de son environnement, de décider de sa case de destination. Une fois une destination décidée, la méthode *update()* est appelée par la méthode *move()* et déplace la particule vers sa destination. Nous avons aussi créé une méthode *findNextCellFromPas()* qui retourne la position de la prochaine case à partir d’une direction.

Ainsi, à chaque tour, chaque particule décide de sa destination en prenant en compte sa direction et son voisinage (les autres particules et les murs) puis se déplace vers la case de destination. Elle décidera de changer de direction, de rebondir, si une autre particule ou un mur se situe devant elle.

Trace


Execution time by particles quantities
'particles.csv'
Execution time (ticks/s)
Amount of particles (particles)
0
5
10
15
20
25
30
35
0
2000
4000
6000
8000
10000

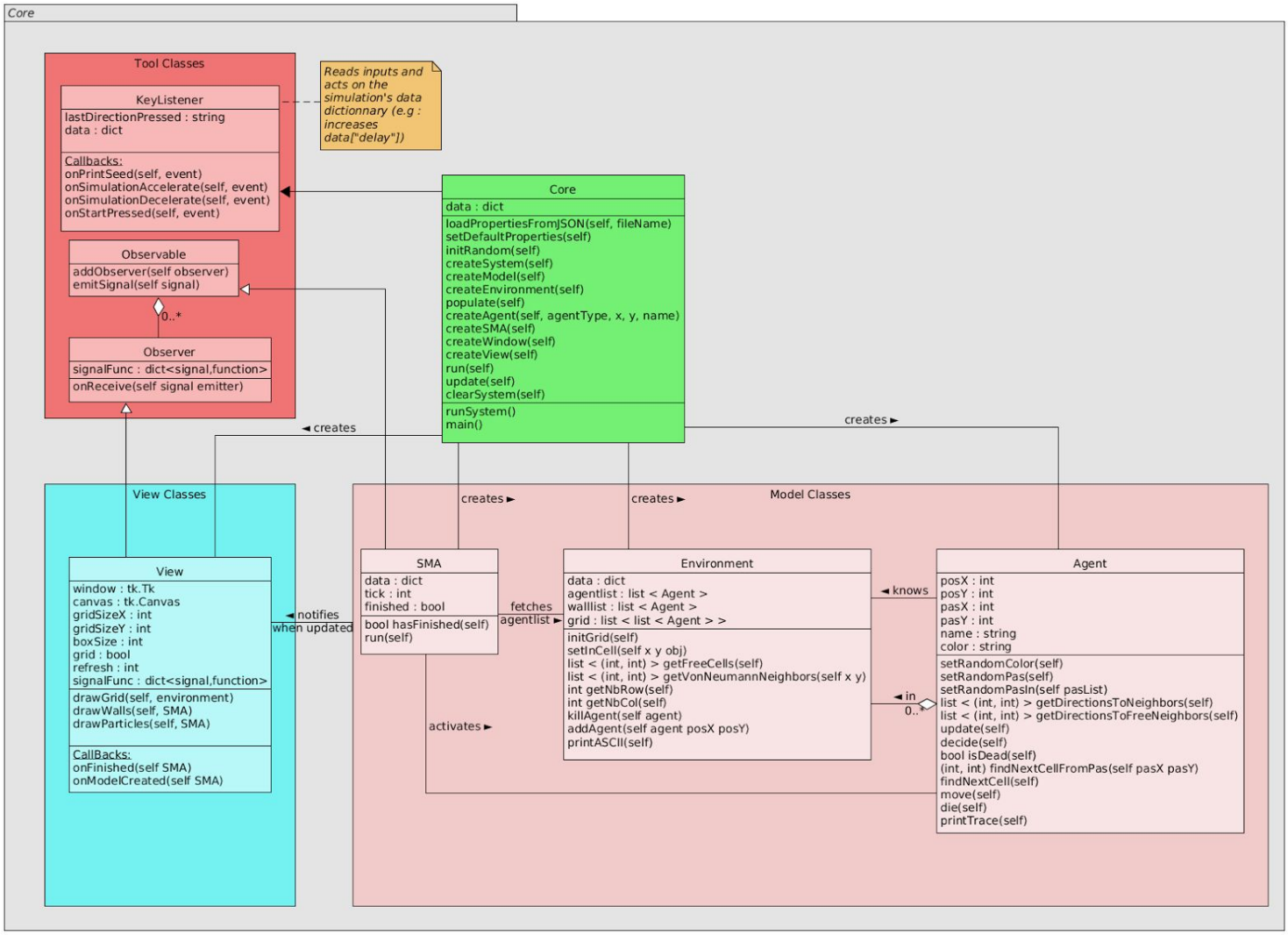
Usage

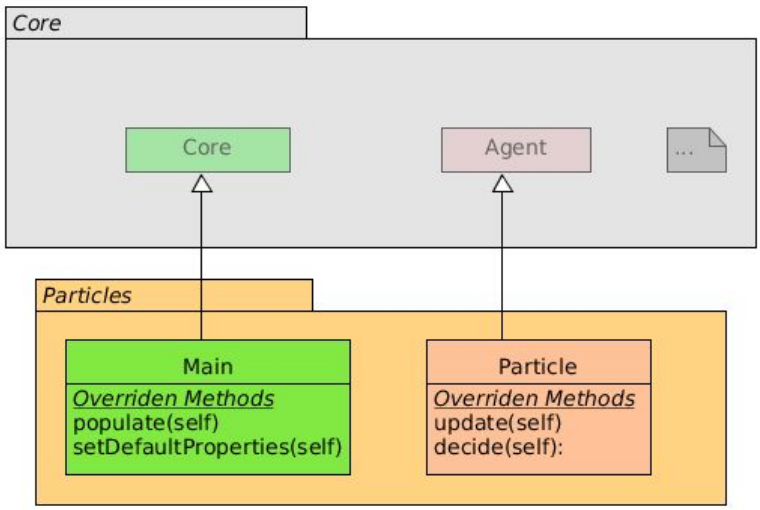
Se déplacer dans le dossier *particles/*
$ python3 Main.py [properties_file.json]
Le fichier properties est en format JSON, prendre exemple sur le fichier properties.json fourni.

Options du fichier JSON

- gridSizeX : Nombre de colonnes de la grille
- gridSizeY : Nombre de lignes de la grille
- torus : *true* si le nombre est torique, *false* sinon
- canvasSizeX : Largeur en pixels de la fenêtre
- canvasSizeY : Hauteur en pixels de la fenêtre
- boxSize : Taille en pixels d'une case de la grille (0 = calculée automatiquement en fonction des options *canvasSize*)
- delay : Délai en millisecondes entre chaque tick
- scheduling : Type de séquenceur utilisé : par défaut séquentiel, *"random"* pour le séquenceur aléatoire
- nbTicks : Nombre de ticks
- grid : *true* pour afficher la grille, *false* sinon
- trace : *true* pour afficher la trace, *false* sinon
- seed : Graine utilisée pour initialiser les générateur de nombres pseudo-aléatoires. 0 pour utiliser une graine aléatoire
- refresh : Nombre de ticks entre chaque rafraîchissement de l'affichage
- nbParticles : Nombre de particules
- autoquit : *true* pour fermer la fenêtre après exécution, *false* sinon

UML


Core
Tool Classes
KeyListener
lastDirectionPressed : string
data : dict
Callbacks:
onPrintSeed(self, event)
onSimulationAccelerate(self, event)
onSimulationDecelerate(self, event)
onStartPressed(self, event)
Reads inputs and acts on the simulation's data dictionnary (e.g : increases data["delay"])
Observable
addObserver(self observer)
emitSignal(self signal)
0..*
Observer
signalFunc : dict<signal,function>
onReceive(self signal emitter)
Core
data : dict
loadPropertiesFromJSON(self, fileName)
setDefaultProperties(self)
initRandom(self)
createSystem(self)
createModel(self)
createEnvironment(self)
populate(self)
createAgent(self, agentType, x, y, name)
createSMA(self)
createWindow(self)
createView(self)
run(self)
update(self)
clearSystem(self)
runSystem()
main()
◄ creates
creates ►
View Classes
View
window : tk.Tk
canvas : tk.Canvas
gridSizeX : int
gridSizeY : int
boxSize : int
grid : bool
refresh : int
signalFunc : dict<signal,function>
drawGrid(self, environment)
drawWalls(self, SMA)
drawParticles(self, SMA)
CallBacks:
onFinished(self SMA)
onModelCreated(self SMA)
◄ notifies when updated
creates ►
creates ►
Model Classes
SMA
data : dict
tick : int
finished : bool
bool hasFinished(self)
run(self)
fetches agentlist ►
activates ►
Environment
data : dict
agentlist : list < Agent >
walllist : list < Agent >
grid : list < list < Agent > >
initGrid(self)
setInCell(self x y obj)
list < (int, int) > getFreeCells(self)
list < (int, int) > getVonNeumannNeighbors(self x y)
int getNbRow(self)
int getNbCol(self)
killAgent(self agent)
addAgent(self agent posX posY)
printASCII(self)
◄ knows
◄ in
0..*
Agent
posX : int
posY : int
pasX : int
pasY : int
name : string
color : string
setRandomColor(self)
setRandomPas(self)
setRandomPasIn(self pasList)
list < (int, int) > getDirectionsToNeighbors(self)
list < (int, int) > getDirectionsToFreeNeighbors(self)
update(self)
decide(self)
bool isDead(self)
(int, int) findNextCellFromPas(self pasX pasY)
findNextCell(self)
move(self)
die(self)
printTrace(self)


Core
Core
Agent
...
Particles
Main
Overriden Methods
populate(self)
setDefaultProperties(self)
Particle
Overriden Methods
update(self)
decide(self):

TP2 - S.M.A Travail sur les comportements

Présentation

L'objectif de ce TP était de réaliser un système multi-agents de type proies-prédateurs avec des requins et des poissons.

Pour réaliser ce système nous avons principalement utilisé et modifié les classes actuellement présentes dans le dossier core.
En plus de celles-ci, nous avons créé deux nouvelles classes ainsi qu'un *Main* propre au projet *wator* :

- Une classe *Fish* définissant le comportement des poissons
- Une classe *Shark* définissant le comportement des requins

Comportement des poissons

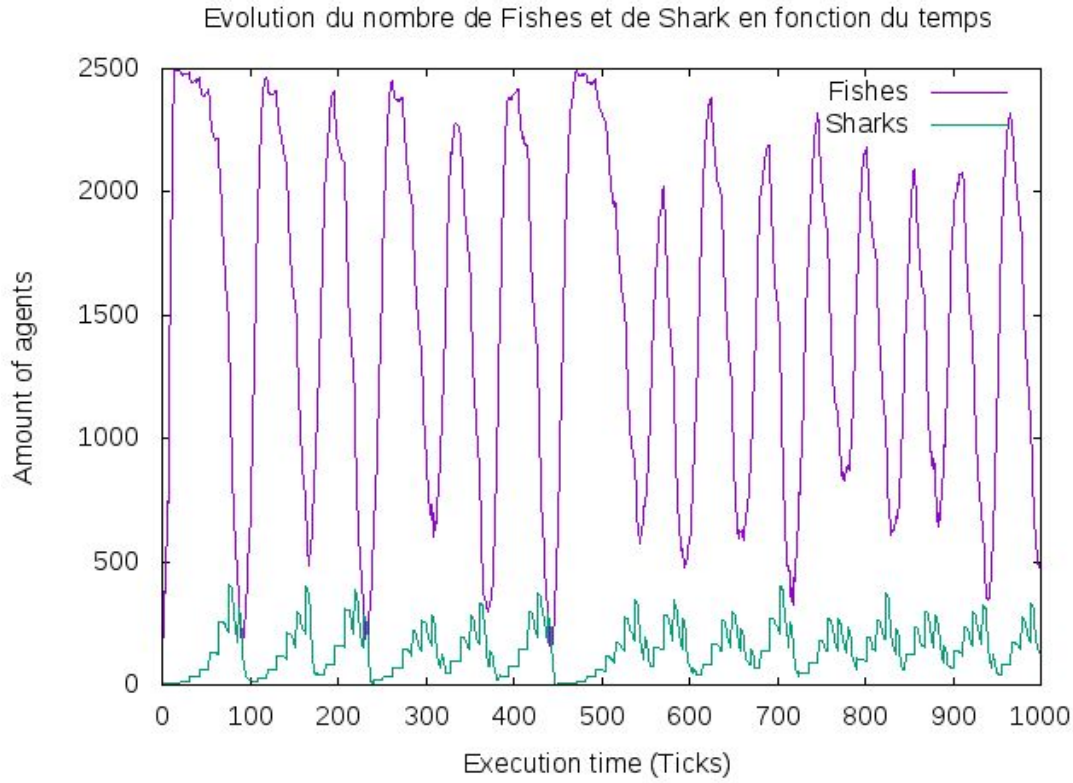
Tout comme pour les particules, la méthode *decide()* des poissons détermine leur destination et choisit donc une case libre aléatoire dans le voisinage du poisson.
Une poisson peut aussi se reproduire grâce à la méthode *breed()* lorsque sa période de gestation est atteinte. Pour ce faire, s'il est en âge de procréer après un déplacement, il créera un autre poisson à son ancienne position. S'il n'est pas capable de se déplacer, il tentera alors de se reproduire en se déplaçant au tour suivant. Finalement, un poisson peut aussi mourir (s'il se fait manger par un requin) via la méthode *die()*.

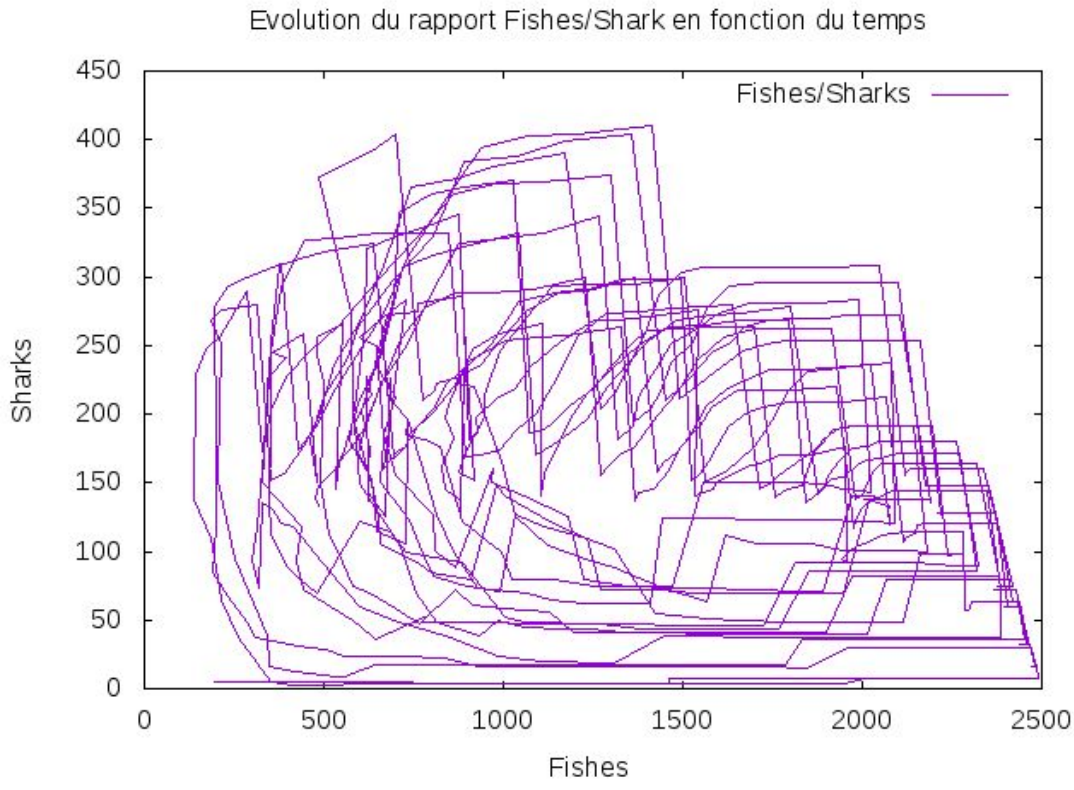
Comportement des requins

Le requin se reproduit d'une manière similaire au poisson.
S'il ne trouve pas de poisson dans son voisinage direct, le requin décidera de son déplacement de la même manière que le poisson. Sinon, il se dirigera vers un poisson aléatoire dans son voisinage et se déplacera à sa place.
La méthode *eat()* permet à un requin de manger. Ils possèdent un compteur qui diminue progressivement avant de tomber à zéro, auquel cas l'agent mourra de faim. Si un requin mange, il fait le plein de nourriture et le compteur reprend sa valeur initiale. La méthode *starve()* diminue ce compteur si le requin n'a pas mangé et peut le tuer le cas échéant.

Ce sont les méthodes *update()* des poissons et des requins qui appellent leurs différentes actions.

Trace


Evolution du nombre de Fishes et de Shark en fonction du temps
Fishes
Sharks
Amount of agents
Execution time (Ticks)
0
500
1000
1500
2000
2500
0
100
200
300
400
500
600
700
800
900
1000


Evolution du rapport Fishes/Shark en fonction du temps
Fishes/Sharks
Sharks
Fishes
0
50
100
150
200
250
300
350
400
450
0
500
1000
1500
2000
2500

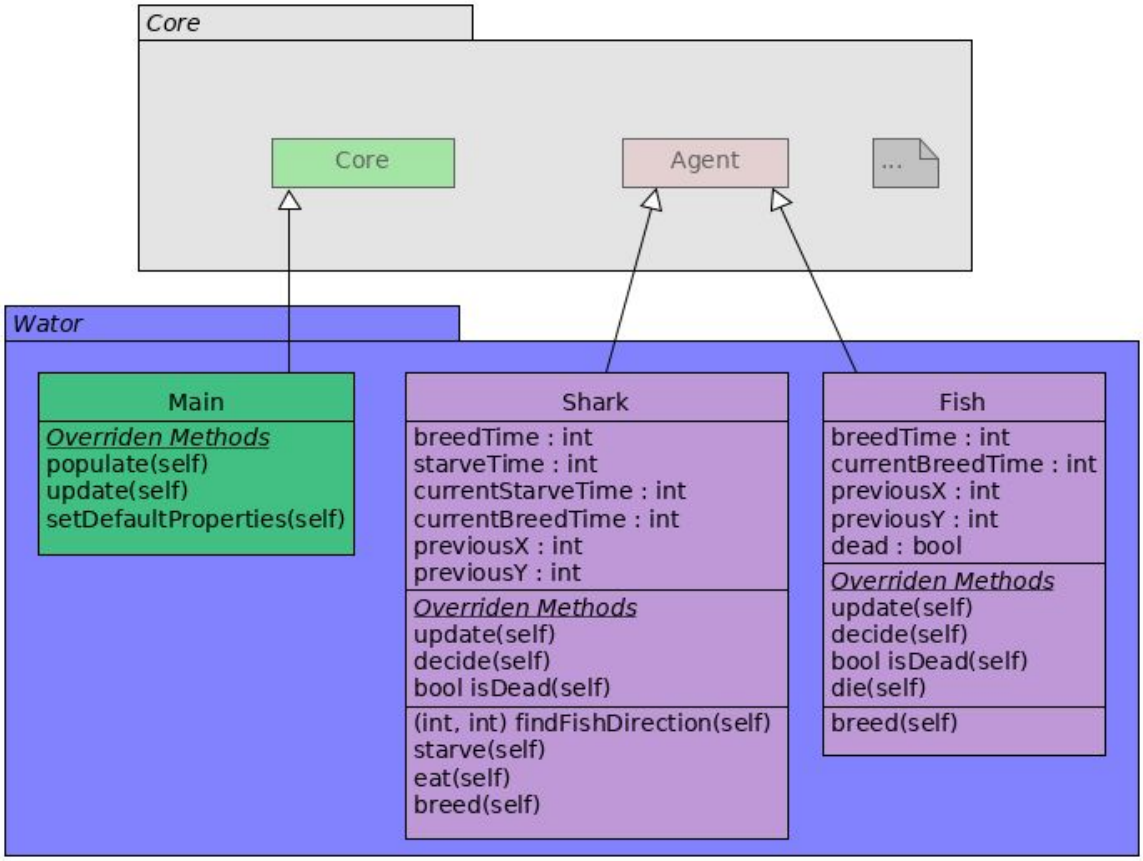
Usage

Se déplacer dans le dossier *wator/*
$ python3 Main.py [properties_file.json]
Le fichier properties est en format *JSON*, prendre exemple sur le fichier properties.json fourni.

Options du fichier JSON

- gridSizeX : Nombre de colonnes de la grille
- gridSizeY : Nombre de lignes de la grille
- torus : *true* si le nombre est torique, *false* sinon
- canvasSizeX : Largeur en pixels de la fenêtre
- canvasSizeY : Hauteur en pixels de la fenêtre
- boxSize : Taille en pixels d'une case de la grille (0 = calculée automatiquement en fonction des options *canvasSize*)
- delay : Délai en millisecondes entre chaque tick
- scheduling : Type de séquenceur utilisé : par défaut séquentiel, *"random"* pour le séquenceur aléatoire
- nbTicks : Nombre de ticks
- grid : *true* pour afficher la grille, *false* sinon
- trace : *true* pour afficher la trace, *false* sinon
- seed : Graine utilisée pour initialiser les générateur de nombres pseudo-aléatoires. 0 pour utiliser une graine aléatoire
- refresh : Nombre de ticks entre chaque rafraîchissement de l'affichage
- autoquit : *true* pour fermer la fenêtre après exécution, *false* sinon
- nbSharks : Nombre de requins
- nbFishes : Nombre de poissons
- sharkBreedTime : Nombre de tours avant qu'un requin puisse se reproduire
- sharkStarveTime : Nombre de tours qu'un requin ne meurt de faim
- fishBreedTime : Nombre de tours avant qu'un poisson puisse se reproduire

UML


Core
Core
Agent
...
Wator
Main
Overriden Methods
populate(self)
update(self)
setDefaultProperties(self)
Shark
breedTime : int
starveTime : int
currentStarveTime : int
currentBreedTime : int
previousX : int
previousY : int
Overriden Methods
update(self)
decide(self)
bool isDead(self)
(int, int) findFishDirection(self)
starve(self)
eat(self)
breed(self)
Fish
breedTime : int
currentBreedTime : int
previousX : int
previousY : int
dead : bool
Overriden Methods
update(self)
decide(self)
bool isDead(self)
die(self)
breed(self)

TP3 - S.M.A Avatars et motion planning

Présentation

L'objectif de ce TP était de réaliser un système multi-agents de type jeu interactif dans lequel un Agent *Avatar* contrôlé par le joueur doit, dans un environnement contenant des murs répartis de manière procédurale, récupérer des bonus nommés *Defender*s tout en évitant de se faire manger par des ennemis appelés *Hunter*s. Les Hunters pourchasseront l'Avatar à l'aide d'un algorithme de PathFinding (Dijkstra) et le tueront au contact. Au bout d'un certain nombre de *Defender*s obtenus, apparaîtra un *Winner* que le joueur devra manger afin de gagner la partie.

Pour réaliser ce système nous avons, en plus des classes présentes dans le dossier *core*, créé les classes suivantes :

- *Avatar*, un agent pouvant être contrôlé au clavier selon le voisinage de Von Neumann, dont le but est de manger des *Defender*s en évitant les *Hunter*s.
- *KeyListener*, réintégré ensuite dans le core, qui associe des touches du clavier à des fonctions décrites plus bas dans ce document;
- *AvatarKeyListener*, qui hérite de *KeyListener* et qui défini les touches du clavier et leurs fonction dans le pour le projet Avatar;
- *Wall*, un simple agent sans comportement, qui empêchera simplement les autres agents de se déplacer à sa position;
- *MapGenerator*, qui permet de générer un environnement de type cave de manière procédurale en utilisant un automate cellulaire (cf. *Création du labyrinthe*);
- *Hunter*, un agent poursuivant l'*Avatar* dans le but de le manger;
- *Defender*, est un agent que l'*Avatar* doit récolter pour faire apparaître le *Winner* et gagner la partie;
- *Winner*, est le dernier agent que l'*Avatar* doit ramasser afin de gagner.

Comportement de l'Avatar

L'*Avatar* est un *Agent* contrôlé par l'utilisateur. Ce contrôle est permis par l'utilisation de la classe *AvatarKeyListener*. À chaque tour, la méthode *decide()* cherchera dans l'*AvatarKeyListener* la dernière direction pressée et changera sa propre direction en fonction de celle-ci.
Si un *Defender* se trouve sur son chemin, il le mangera, ce qui aura pour effet de le rendre invincible et de faire fuir les *Hunters* pendant un temps prédéterminé.
Après chaque mise à jour de position, L'*Avatar* calculera une matrice de scores utilisée par l'algorithme de Dijkstra, dont il sera le centre, et qui sera envoyée aux *Hunter*s en émettant un signal à l'aide d'un composant *Observable* : *AvatarNotifier*.

Comportement du Hunter

Le but du Hunter est de dévorer l'Avatar. Pour atteindre cet objectif, il poursuivra ce dernier à l'aide d'un algorithme de Pathfinding. L'algorithme implémenté est l'algorithme de Dijkstra, en utilisant la matrice de score que leur a communiqué l'*Avatar,* afin de déterminer la case voisine le rapprochant le plus de ce dernier. La classe contient donc un composant *AvatarFollower* qui conserve la dernière copie de la matrice de scores. La méthode *decide()* de la classe récupérera cette matrice pour décider de la direction empruntée par le *Hunter*. Si *l'Avatar* se trouve sur le chemin du *Hunter*, ce dernier pourra le manger, ce qui aura pour effet de mettre fin à la partie.
Lorsque l'*Avatar* est invincible, suite à la prise d'un *Defender*, le *Hunter* cherchera à le fuir. Pour reproduire ce comportement, nous faisons en sorte que la méthode *decide()* du *Hunter* recherche non plus le score minimum dans ses cases voisines, mais le score maximum. Le *Hunter* s'éloignera par conséquent de l'*Avatar*.

Comportement du Defender

Le but de l'Avatar est de récupérer les *Defender*s afin de faire apparaître le *Winner* et ainsi gagner la partie. Lorsque le joueur récolte un *Defender*, ce dernier meurt via la méthode *die()*. Avant de mourir un *Defender* se reproduit avec *breed() afin de faire apparaître un nouveau Defender à ramasser* sur une case aléatoire libre de l'environnement. Si le joueur a ramassé suffisamment de *Defender*s, le *Defender* crée un *Winner* avec *createWinner() avant de mourir*. À chaque tour, comme les requins, le *Defender starve()*. Au bout d'un certain temps, le *Defender* "meurt de faim". Lorsqu'il mange un *Defender*, l'*Avatar* est invincible pendant un temps court et fait fuir les *Hunters.*

Comportement du Winner

Le *Winner* n'a pas de comportement qui lui est propre. Cependant, quand il se fait manger par l'*Avatar*, la partie se termine et le joueur gagne.

Création du labyrinthe

Nous avons dans un premier temps regardé les algorithme de création de labyrinthe proposés pour le TP. Cependant, ces algorithmes créent des mazes qui permettent de n'avoir qu'un seul chemin pour aller d'un point A à un point B. Or, si un *Hunter* se trouve entre l'*Avatar* et son objectif, dans ce cas, l'*Avatar* n'aura pas de moyen de contourner cet *Hunter* puisqu'un seul chemin existe entre lui et son objectif. Dans le jeu *Pacman*, les labyrinthes disposent de nombreux embranchements et de plusieurs chemins qui permettent de contourner les fantômes. Nous avons donc décidé de créer, non pas des labyrinthes, mais des caves générées de manière procédurale afin de laisser la place à l'*Avatar* pour éviter les *Hunter*. Pour ce faire nous avons utilisé un automate cellulaire qui crée tout d'abord des murs aléatoirement répartis dans l'environnement puis qui itère un certain nombre de fois sur cet environnement afin de transformer une cellule en mur ou en vide en fonction de son voisinage. Un mur reste un mur si suffisament de ses voisins sont des murs.

Une case vide devient un mur si suffisament de ses voisins sont des murs. Les paramètres de l'algorithme de génération peuvent être modifiés dans les properties.

Pour plus d'informations :
http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels

Usage

Se déplacer dans le dossier *avatar/*
$ python3 Main.py [properties_file.json]
Le fichier properties est en format JSON, prendre exemple sur le fichier properties.json fourni.

En plus des propriétés paramétrables, nous pouvons changer dynamiquement certains paramètres de la simulation grâce aux touches suivantes :

- A pour augmenter la vitesse des Hunters, Z pour la diminuer;
- O pour augmenter la vitesse de l'Avatar, P pour la diminuer;
- Espace pour mettre la simulation en pause / la reprendre;
- W pour diminuer le délai entre les ticks et accélérer la simulation, X pour l'augmenter;
- S pour imprimer la seed actuellement utilisée pour la génération du modèle.

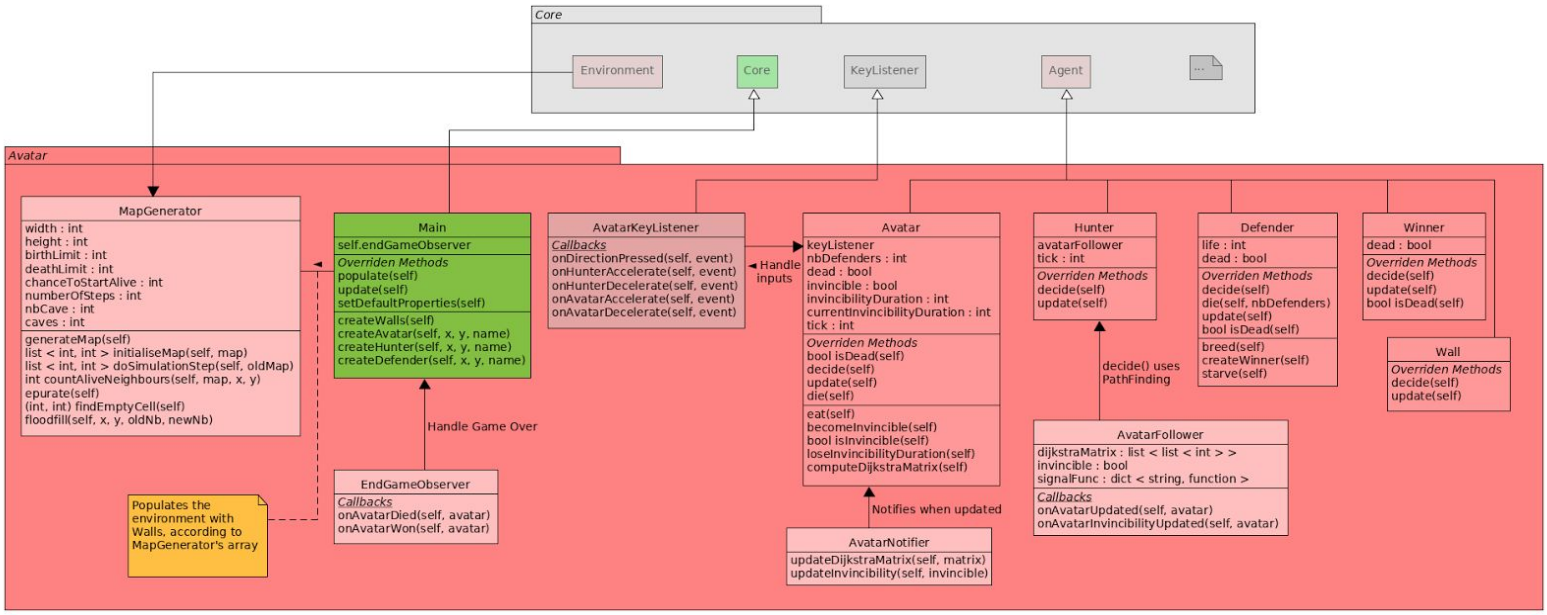
Options du fichier JSON

- gridSizeX : Nombre de colonnes de la grille
- gridSizeY : Nombre de lignes de la grille
- torus : *true* si le nombre est torique, *false* sinon
- canvasSizeX : Largeur en pixels de la fenêtre
- canvasSizeY : Hauteur en pixels de la fenêtre
- boxSize : Taille en pixels d'une case de la grille (0 = calculée automatiquement en fonction des options *canvasSize*)
- delay : Délai en millisecondes entre chaque tick
- scheduling : Type de séquenceur utilisé : par défaut séquentiel, *"random"* pour le séquenceur aléatoire
- nbTicks : Nombre de ticks
- grid : *true* pour afficher la grille, *false* sinon
- trace : *true* pour afficher la trace, *false* sinon
- seed : Graine utilisée pour initialiser les générateur de nombres pseudo-aléatoires. 0 pour utiliser une graine aléatoire
- refresh : Nombre de ticks entre chaque rafraîchissement de l'affichage
- autoquit : *true* pour fermer la fenêtre après exécution, *false* sinon
- nbHunters : Nombre de chasseurs
- nbDefenders : Nombre de défendeurs
- invincibilityDuration : Durée d'effet d'un défendeur en tour
- speedHunter : Vitesse des chasseurs
- speedAvatar : Vitesse du joueur

- defenderLife : Nombre de tour avant qu'un défenseur ne soit détruit et qu'un autre apparaisse ailleurs
- birthLimit : Si le nombre de voisins vivant d'une case morte dépasse cette valeur, la case prend vie
- deathLimit : Si le nombre de voisins vivant d'une case vivante est inférieur à cette valeur, la case meurt
- chanceToStartAlive : Probabilité qu'une case soit un mur à la première étape de l'automate cellulaire
- numberOfSteps : Nombre d'étapes de l'automate cellulaire

Nous n'avons pas explicitement de paramètre WallPercent. Cependant, nous pouvons agir sur le nombre de murs et les propriétés des caves générées en modifiant les paramètres birthLimit, deathLimit, chanceToStartAlive, numberOfSteps du fichier de properties.

UML



(Les UML sont disponibles en format .png dans chaque package de l'archive donnée pour le rendu)