Université Paris Cité

Compléments en Programmation Orientée Objet
Projet : Serpents

Projet à réaliser en binôme, à rendre sur Moodle, avant la date indiquée sur Moodle.
Le langage doit être Java (version 17 ou 21).
Le code doit être écrit par vous, mais votre programme peut dépendre de bibliothèques tierces (celles indiquées dans le sujet ou des bibliothèques rendant un service équivalent).

Ce projet consiste en un jeu de serpents en réseau (inspiré de l'antique jeu Snake, revisité à la façon du site slither.io).

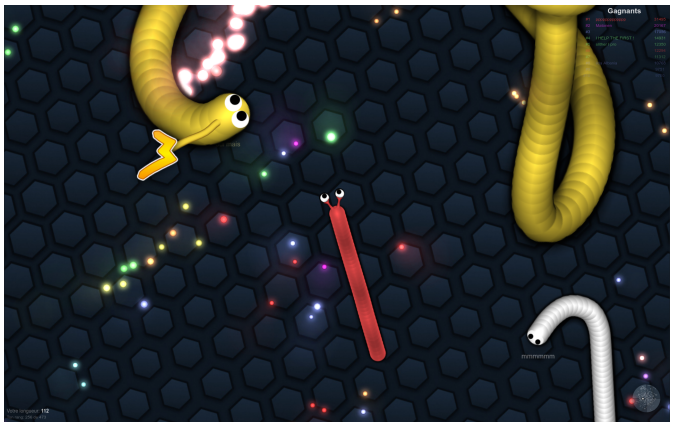

FIGURE 1 – Capture d'écran du site slither.io

1 Déroulement du jeu

Chaque joueur contrôle un serpent, qui se dirige vers le pointeur de la souris [1] a une vitesse constante. En réalité, le serpent se divise en segments, seul le premier segment fait un pas dans la direction indiquée à chaque mise à jour. Le second fait un pas vers le premier, le troisième vers le second, et ainsi de suite. Cela implique que le serpent ne reste pas tout droit, et qu'il y a une certaine inertie dans le déplacement de la queue.
Le serpent est petit au début, mais gagne un segment à chaque fois qu'il mange. Dans slither.io, il s'agit de manger des points lumineux.
Dans le jeu classique, tel qu'il se présentait sur les téléphones portables à la fin des années 90, la partie en solo se déroule sur un terrain relativement exigu (de la taille de l'écran). L'objectif est de survivre jusqu'à obtenir le plus grand serpent possible, sachant qu'il est de plus en plus inévitable de finir par s'écraser dans un mur ou dans sa propre queue mesure que le serpent grandit. Les deux cas de collision impliquent la mort immédiate du serpent.
Dans le jeu multijoueur, l'idée c'est de faire des queue de poisson aux serpents des autres joueurs, afin de survivre plus longtemps. Dans slither.io, la mort d'un concurrent provoque l'apparition d'un nuage de points lumineux que l'on peut manger pour grandir.
Le jeu multijoueur classique se joue généralement à 2 joueurs (ou en tout cas un petit nombre), sur la même machine. Une manche se termine lorsqu'il ne reste qu'un seul survivant. Dans slither.io, par contre, le jeu est est massivement multi-joueur et la partie, qui ne s'arrête jamais (monde permanent), peut être rejointe à tout moment par un nouveau joueur (ou un ancien joueur éliminé qui veut tenter sa chance à nouveau).

2 Jeux à programmer

La consigne est simple : en utilisant un maximum de principes de POO vus en cours cette année, programmer un jeu de serpent.
Les fonctionnalités à implémenter (en ordre de priorité) :

1. déplacement d'un seul segment dans une grille à maillage rectangulaire, coordonnées entières, avec pas de 1 unité vers le nord, l'est, le sud ou l'ouest. Le déplacement est tour par tour.

1. ou dans une direction indiquée autrement, on peut envisager plusieurs modes de contrôle

2. « IA » : un joueur automatique plus ou moins intelligent, surtout là pour vous aider à tester ; cette « IA » devra être adaptée au fur et à mesure qu'on ajoute des fonctionnalités
3. déplacement d'un serpent de multiples segments se déplaçant comme décrit plus haut
4. gestion de la gestion de la mort : collision avec un obstacle tiers et auto-collision
5. gestion de la croissance : quand on passe sur un morceau de nourriture, il disparaît, un autre apparaît à une position aléatoire sur le terrain et le serpent s'allonge d'un segment ;
6. déplacement en temps réel dans une fenêtre graphique : on se donne une « unité » de temps qui correspond au temps nécessaire pour faire un pas entier ; on fixe une nouvelle direction toutes les « unités » de temps ; celle-ci est donnée par la dernière flèche de direction appuyée par le joueur
7. 2 joueurs au clavier
8. déplacement fluide : les coordonnées sont maintenant des `double` et la taille du pas doit être égale à la vitesse fois le temps écoulé depuis le dernier pas ; on met à jour la direction à chaque fois que c'est possible sans attendre une unité de temps fixée
9. déplacement libre : la direction n'est plus donnée par la dernière touche appuyée, mais par la position du pointeur par rapport à la tête du serpent ; cette direction peut suivre un angle quelconque, représenté par un `double` ; le pointeur lui-même peut être déplacé à la souris et/ou au clavier
10. terrain sans bords : le terrain est bien plus grand que la zone d'affichage (gérer défilement) ; traverser un bord fait ré-apparaître le serpent sur le bord opposé (faire en sorte qu'on ne se rende pas compte de cela dans l'UI qui doit donner l'illusion d'un terrain infini)
11. règles personnalisées : inventez d'autres règles. Un exemple : différentes natures de segment aux propriétés différentes. On peut imaginer des segments :
    - « faibles », qui au lieu de causer la mort du serpent qui rentre dedans, cassent la queue du serpent à ce niveau
    - « boucliers », évitant une mort immédiate lors d'une collision, si ce segment est en tête
    - « empoisonnés » qui tueraient le serpent qui mangerait le morceau de nourriture créé après la mort de notre serpent (dans le cadre des règles slither.io)
    - « aérodynamiques », qui augmentent la vitesse (ou la réduisent)
    - « fouisseurs » permettant, si placé en tête, de passer sous les obstacles....

    C'est le type de nourriture absorbé qui détermine quel type de segment est créé.
12. mode réseau classique : en mode client/serveur ; le serveur fait foi pour ce qui est de l'état du jeu (ce qui n'empêche pas au client d'interpoler les déplacements pour donner une impression de fluidité) ; le jeu n'est plus limité à 2 joueurs
13. monde permanent : comme dans slither.io, la partie n'est jamais terminée, et on peut la rejoindre n'importe quand.
14. règles slither.io : la nourriture provient de la mort d'un serpent ; la vitesse dépend de la taille ; on peut « brûler » un segment pour augmenter temporairement sa vitesse... Au point où vous en êtes, ces modifications devraient très peu changer votre base de code.

En général. En passant d'une version à l'autre vous vous assurez que le mode de jeu de la version précédente et le nouveau mode de jeu partagent un maximum de code. Cela peut nécessiter de refactoriser la version précédente. Quand c'est le cas, vérifiez que le mode de jeu précédent fonctionne toujours aussi bien avec la nouvelle base de code (on doit pouvoir le lancer soit par un autre `main`, soit par des options du jeu).

Jusqu'où faut-il aller ?

- on veut voir implémentés plusieurs types de segments, pour que vous puissiez montrer votre maîtrise du polymorphisme ;
- on veut un mode réseau pour que vous montriez que vous gérez la concurrence correctement.

Évidemment, plus il y aura de fonctionnalités correctement implémentées, meilleure sera votre note.

3 Contraintes techniques

Réseau L'architecture sera client-serveur. Le serveur est un programme séparé du client (ou peut être lancé comme tel).
On aimerait que les clients et serveurs soient interopérables entre les binômes (c'est bon pour la qualité), mais il ne faut pas que cela vous fasse perdre trop de temps. Cela dit n'hésitez pas à partager votre protocole sur le forum Moodle du projet si vous arrivez au mode réseau.
Une implémentation basique de client et de serveur TCP vous seront fournis à titre d'exemple.

Concurrence Le logiciel en mode solo s'exécute entièrement dans le *thread* applicatif du toolkit graphique choisi, c'est-à-dire sans création explicite de *thread* supplémentaire. En revanche, en mode réseau, il faudra que le client créée un *thread* séparé pour gérer la connexion au serveur.
Le serveur, lui, utilisera un *thread* pool (1 *thread* par connexion client).
Dans tous les cas, prenez soin de vérifier la bonne synchronisation. Privilégiez les structures immuables et limitez le nombre de variables partagées.

Paramétrabilité et réutilisabilité Le jeu comporte de multiples options qui peuvent se combiner de différentes façons. Privilégiez une architecture permettant cela et favorisant la réutilisation de code. Quelques éléments :

- préférer composition à héritage (en effet, l'héritage ne permet pas les combinaisons arbitraires)
- prévoir une classe pour stocker et centraliser la configuration, les objets du jeu s'instancient en lisant ces paramètres (pensez aux fabriques statiques)
- une instance d'une telle configuration est fabriquée notamment grâce à l'écran de configuration du jeu
- une instance de configuration doit pouvoir être sérialisée/désérialisée (en JSON ?) ; cela permet d'enregistrer les paramètres d'une session sur l'autre, de les stocker, se les échanger ser le réseau, etc.

Testabilité Dans tous les modes, tout ce que vous écrirez devra avoir été programmé de telle sorte à ce que le comportement soit testable.
Le découpage en méthodes doit être suffisamment fin pour permettre des tests unitaires pertinents.
Le découpage en classes et la programmation « à l'interface » doivent permettre d'effectuer facilement des tests d'intégration. Cela peut être fait en remplaçant n'importe quel composant par un composant factice implémentant la même interface.
Les tests devront être écrits, exécutables, et fournis avec le rendu.

4 Aide technique

- Utilisez un système de compilation tel que Maven ou Gradle. Vous faciliterez la gestion des dépendances.
- Interface graphique : Swing, JavaFX [2] (vous pouvez aussi consulter les transparents du cours sur les IG, qui sont basés sur JavaFX) ou autre. Si JavaFX, utilisez le plugin JavaFX

2. `https://openjfx.io/`

pour Maven ou Gradle, vous vous simplifierez la vie. Pour tout autre toolkit que Swing, l'installation des dépendances doit se faire automatiquement (bref, utilisez Maven ou Gradle).
- Réseau : vous pouvez utiliser une connexion TCP. Regardez la documentation des classes `java.net.Socket` et `java.net.ServerSocket`. Vous trouverez aussi sur votre moteur de recherche préféré plusieurs cours d'enseignants de l'UPC qui expliquent ces classes.
- Les objets passés sur le réseau peuvent être sérialisés en utilisant un format standard tel que JSON. Il est utile pour cela d'utiliser un bibliothèque telle GSON [3].
- Les tests peuvent être écrits dans un *framework* tel que JUnit5 [4], mais ce n'est pas indispensable. En tout cas, il faut des tests.

Un exemple minimal utilisant tout cela sera mis à votre disposition sur Moodle.

5 Critères d'évaluation

- Le projet est rendu dans une archive `.zip`.
- Le projet doit contenir un fichier `README.md` indiquant comment compiler, exécuter et utiliser votre programme et ses tests, décrivant les fonctionnalités effectivement implémentées et expliquant vos choix techniques originaux, le cas échéant.
- Joignez aussi un ou des diagrammes de classe.
- La commande pour compiler ou exécuter doit être simple (s'aider de Gradle, Maven... voire de Makefile).
- La compilation selon ces instructions doit terminer sans erreur ni avertissement.
- L'exécution doit être correcte : elle respecte le cahier des charges et ne quitte pas de façon non contrôlée.
  Notamment, en mode graphique, les exceptions doivent être rattrapées. Dans tous les cas, si le problème n'est pas réparable, un message d'erreur doit être présenté à l'utilisateur. Aucune de ces erreurs ne doit être due à un problème de programmation (comme le fameux `NullPointerException`...).
- Le code respecte les conventions de codage.
- Le code est architecturé intelligemment. Notamment, il faut utiliser des patrons de conception vus en cours ainsi que les constructions les plus appropriées fournies par Java.
- Les classes et méthodes sont documentées (javadoc).
- Des commentaires expliquent les portions de codes qui ne sont pas évidentes.
- Les tests fournis doivent être aussi exhaustifs que possible.
- Un projet qui tenterait de tout faire mais dans lequel rien ne marche bien sera moins bien noté qu'une projet qui ne remplit que la première moitié des objectifs sans avoir de bug.
  Les objectifs sont incrémentaux. Il est toujours possible de rendre un projet cohérent qui n'aurait atteint que les premiers objectifs, donc profitez-en !

3. `https://github.com/google/gson`
4. `https://junit.org/junit5/`