Projet : Serpents

1 Gameplay

Concernant le gameplay, les règles de gameplay sont dans l'ensemble celles de slither.io, avec certaines modifications pour satisfaire le besoin de règles personnalisées. Ainsi, un serpent n'entre jamais en collision avec lui-même, c'est un choix de gameplay qui permet aux joueurs d'encercler un serpent afin de l'attraper.

Concernant la nourriture, il en existe 5 types correspondant aux règles personnalisées demandées:

- Normale: rajoute un segment normal
- Faible: rajoute un segment faible, qui peut se faire couper par les auters segments
- Bouclier: lorsque ce segment est placé en tête, le serpent est protégé lorsqu'il rentre dans un serpent. Il ne peut y avoir qu'un seul segment de ce type-là, pour des raisons d'équilibrage de gameplay.
- Empoisonnée: tue le serpent lorsqu'il la mange
- Aérodynamique: accélère le serpent temporairement

2 Explication des choix d'implémentation

2.1 Affichage: Java Swing

Java Swing a été choisi en raison de sa simplicité et de sa facilité de mise en œuvre pour des jeux 2D relativement simples comme celui-ci. En utilisant les composants graphiques de Swing tels que JPanel et en surchargant la méthode `paint(Graphics)` il est très simple de dessiner le jeu de snake.

Pour ce qui en est de la performance, bien que Swing ne soit peut-être pas aussi performant que d'autres bibliothèques graphiques conçues spécifiquement pour les jeux, pour des jeux moins exigeants en termes de graphismes et de performance (comme Snake), Swing est tout à fait adéquat.

2.2 Terrain sans bords

L'implémentation du terrain sans bords n'est pas simple aux premiers abords. Effectivement, bien qu'il est simple de faire en sorte que les positions restent dans la carte du jeu, en utilisant des comparisons et l'opérateur modulo. Il est plus compliqué de faire en sorte que la transition se passe de manière inaperçue dans l'interface du jeu de snake. L'approche qui a été prise dans ce jeu a été de dessiner plusieurs fois la carte du jeu. Ainsi on a:

Le carré bleu représente la carte du jeu, le fait de dessiner en plusieurs fois le jeu tel que représenté permet le fait que dès que l'on sort de la carte du jeu, par exemple par le bord droit, on puisse voir le le bord gauche de la carte. Ainsi, la transition se fait discrètement et fluidement.

3 Solutions Apportées

3.1 Déplacement fluide du serpent

Pour la question du déplacement fluide du serpent, j'ai dû penser à l'efficacité. Je ne voulais pas que chaque segment fasse un pixel, car cela entrainerait rapidement beaucoup de pression sur la RAM et surtout sur le réseau (il faudrait plus de bande passante pour communiquer les informations sur tous les segments). J'ai donc décidé de maintenir une liste séparée nommée `lastPositions`. Cette liste contient la position de la tête du serpent pour chaque pixel de déplacement.

Ainsi, pour avoir un espacement de 10 pixels entre chaque segment, il suffit ainsi de prendre 1 élément de `lastPositions` sur 10. Lorsque la taille de la liste `lastPositions` dépasse $10 \times$ le nombre de segments, le premier élément de la liste est enlevé et tous les éléments suivants sont donc décalés. C'est cette partie du code qui permet aux segments de continuer à se déplacer au lieu de rester bloqués après un certain nombre de déplacements.

L'avantage de cette solution est qu'en plus d'être simple, elle est également configurable, car en changeant une seule variable on peut changer l'espacement en pixels entre tous les segments.

3.2 IA

L'IA a au départ été implémentée de la manière la plus simple possible. C'est à dire, en se posant la question: quelle est la chose la plus importante dans un jeu de snake ? La réponse est qu'il faut faire grandir son serpent, avoir le plus de segments possible. Et pour ce faire, il faut pouvoir manger la nourriture. C'est ainsi que la première version de l'intelligence artificielle des serpents était simplement un algorithme qui cherchait la nourriture (non-empoisonnée) la plus proche et dirigeait le serpent vers elle, avec de l'interpolation linéaire afin de rendre le mouvement plus fluide.

Il y a rapidement eu besoin d'un algorithme plus sophistiquée, car l'intelligence artificielle des serpents rentrait dans les autres serpents et mourrait, étant donné qu'elle ne prenait pas compte de la position des autres. Il a donc été décidé de rajouter une méthode `getClosestThreat` qui donne au serpent la position (en `Point2D.Double`) de la menace la plus proche. Menace inclut les segments (non-faibles) des autres serpents, et la nourriture empoisonnée. Le serpent va ainsi fuir la menace la plus proche avec une force inversement exponentielle à la distance suivant la formule $f(x) = \min\left(0.2, \exp\left(\frac{\text{distance}}{50}\right)\right)$ où la fonction min est utilisée afin de mettre le maximum de force à 0.2, afin d'éviter que le mouvement soit trop brusque.

3.3 Collision avec les autres serpents

La détection de collision avec les autres serpents se passent juste après le déplacement du serpent, car il est important que les collisions soient dûes au déplacement du serpent et non pas au déplacement des autres serpents. Sinon, un serpent pourrait venir vers le joueur et ce serait le joueur qui mourrerait, ce qui ne serait pas logique. C'est donc pour ça qu'il y a la méthode `moveAndCheckCollisions` qui déplace le serpent et vérifie les collisions immédiatement après.

La vérification de collision en elle-même se base sur le fait que chaque segment est un cercle. Or il est assez simple de vérifier la collision entre un disuqe et un autre disuqe. Il suffit de savoir si l'intersection entre l'intérieur du disque du cercle C_1 et l'intérieur du disque du cercle C_2 n'est pas vide. Or, un disque est définit comme l'ensemble des points à une distance du centre inférieure au rayon. Il en convient donc que pour qu'il y ait intersection, il faut que la distance entre les deux cercles soit inférieur à la somme de leur rayon, c'est à dire le diamètre (dans le cas où ils ont le même rayon).

3.4 Déplacement libre

Le déplacement libre avec les touches du clavier fut la partie la plus simple à coder, la flèche gauche diminue l'angle et la flèche droite augmente l'angle. Ainsi, lorsque l'on appuie sur la flèche gauche, le serpent tourne à gauche, et lorsque l'on appuie sur la flèche droite, le serpent tourne à droite.

Pour le déplacement libre avec la souris il faut d'abord s'attaquer à la trigonométrie. Comme nous le savons, la fonction tangente est la fonction qui associe à un angle son côté opposé divisé par son côté adjacent dans un triangle rectangle. Si on dessine un triangle rectangle pour la position de la souris, on trouve que la longueur du côté opposé est égal à la position verticale de la souris et que la longueur du côté adjacent est égal à la position horizontale de la souris. On en convient donc que la valeur de la tangente est $\frac{y}{x}$. Cette information devient utile par le fait qu'on puisse aussi associer à chaque tangente la valeur de l'angle qui le donne, avec la fonction arctan (des fois notée $\tan^{-1}$). Il faudrait donc faire $\arctan\left(\frac{y}{x}\right)$? Sauf que cela ne fonctionnera pas (totalement) car par exemple, dans le cas d'un angle de 90°, $\frac{y}{x}$ n'existe pas car on aurait $x = 0$. Il faut donc pouvoir s'occuper de tous ces cas particuliers, et pour ça Java fournit la fonction `Math.atan2`.

Donc, pour trouver l'angle que doit suivre le serpent, il suffit donc de faire `Math.atan2(y, x)` où `y` et `x` sont les coordonnées de la souris relatives à la position de la tête du serpent.

3.5 Multijoueur

Pour le multijoueur, il a été décidé d'utiliser un système de paquets. C'est un système très commun parmi les jeux multijoueurs, que ce soit sur les connexions TCP ou UDP. La raison à ça est que ce système se prêtent bien à tout type de jeu, étant donné qu'un jeu est souvent une succession d'évènements, et donc pour les représenter, par des paquets. De plus, pour ce genre de systèmes, utiliser TCP représente un avantage considérable car ce protocole va s'occuper du fait que les paquets arrivent (il, que UDP oublie des quelques bouts), qu'ils arrivent dans l'ordre (UDP parle ne dans l'ordre pas), et qu'ils ne sont pas modifiés (avec UDP on peut envoyer une pizza et recevoir un burger).

Afin de simplifier la gestion du jeu et l'implémentation du protocole réseau, seul quelques types de paquets on été retenus:

- `ServerSettingsPacket`: utilisé par le serveur pour envoyer les paramètres à utiliser au client
- `SnakeSpawnedPacket`: envoyé par le serveur pour signaler qu'un serpent est apparu, ce que paquet sert aussi à indiquer au client lequel des serpents est son joueur.
- `SnakeRemovedPacket`: envoyé par le serveur pour signaler qu'un serpent a été enlevé (soit il est mort, soit le joueur a quitté la partie)
- `SetFoodPacket`: envoyé par le serveur pour mettre à jour / synchroniser la nourriture présente sur la carte
- `JoinPacket`: envoyé par le client pour demander de rejoindre la partie ou pour ressuciter.
- `ChangeSnakePacket`: envoyé par le client pour indiquer les changements de propriétés de son serpent, ou envoyé par le serveur pour indiquer les changements de propriétés des autres serpents.

Un autre avantage du système de paquet est qu'en plus de promouvoir l'encapsulation, il permet de bâtir une abstraction par-dessus la couche réseau. Ainsi, au lieu de directement utiliser les méthodes de `java.net.Socket`, chaque paquet doit implémenter 2 méthodes: une pour écrire le paquet et une pour lire le paquet. Ils prennent un `DataOutputStream` et un `DataInputStream` respectivement.

Le fait de les isoler ainsi de la couche réseau permet de simplifier le code, de le rendre plus extensible, et de gérer plus facilement des opérations commune à tous les paquets (comme une méthode `sendPacket`). Cela signifie aussi que la boucle du thread en charge de la communication avec le serveur ou avec le client n'a qu'à faire deux choses: lire un paquet avec `Packet.read()` et le traiter. Encore une fois, c'est une majeure simplification par rapport à l'alternative.

Enfin, chaque paquet transmet ses informations en JSON et quelques une sont envoyés en format binaire directement (grâce aux classes `DataOutputStream` et `DataInputStream`), pour prendre moins de place. Pour sérialiser et désérialiser en JSON, c'est la bibliothèque Gson qui est utilisée.

3.6 Performance

Dans le développement du jeu vidéo, un des problèmes encontrés a été celui de la performance graphique. Java Swing n'est déjà pas connu comme ayant de bonnes performances, mais en plus, comme montré dans le Chapitre 2.2, le jeu est dessiné 9 fois. Il y a donc besoin de faire des optimisations afin de ne pas surcharger la carte graphique des joueurs.

3.6.1 Occlusion des objets

La première optimisation qui a été implémentée, une optimisation rudement efficace car elle fut en plus la seule nécessaire pour pouvoir tourner sur tous les ordinateurs testés, c'est l'occlusion. Ce que cela veut dire est que lorsqu'on peut détecter qu'un objet se trouvera entièrement en dehors de la fenêtre, on ne le dessine pas. Ainsi, on évite de faire travailler l'ordinateur pour lui faire afficher quelque chose que le joueur ne verra pas. Cette optimisation a été suffisante pour atteindre les 30 images par secondes initialement visés.

4 Utilisation des Patrons de Conception

4.1 Observer/Observable

Le patron de conception `Observer/Observable` a été utilisé pour la classe `FoodManager` de par son utilité pour le multijoueur. En effet, les instances de la classe `FoodManager` peuvent subir une modification sous différentes formes: soit de la nourriture a été rajoutée, soit de la nourriture a été retirée. Chaque nourriture étant considérée comme immuable, il est impossible que ses propriétés changent, et donc ce cas n'est pas à prendre en compte.

Utiliser ce patron de conception permet donc à ce qu'en multijoueur, le serveur puisse s'enregistrer comme `Observer` du `FoodManager` et ainsi envoyer un `SetFoodPacket` à tous les clients dès qu'il y a une modification de l'état de `FoodManager`. Cela permet ainsi que l'état du serveur et l'état du client soient tous deux synchronisés.

4.2 Fabriques statiques

Les fabriques statiques sont utilisés à de maintes reprises dans le code source du jeu, que ce soit pour la classe `Snake`, pour `FoodManager`, ou pour `MultiplayerGameState`.

Pour la classe `Snake`, la fabrique statique est utile comme patron de conception car elle permet au `Snake` d'utiliser l'objet `Settings` afin de s'instancier. Elle permet également d'utiliser une seule classe pour instancier les deux types de serpent: serpents joueurs et serpents IA, avec les méthodes `Snake.createPlayerSnake(Settings)` et `Snake.createAISnake(Settings)` respectivement.

Pour la classe `MultiplayerGameState`, l'utilisation est évidemment similaire, mais ici elle permet aussi un avantage en plus par rapport à l'utilisation dans d'autres classes. Notamment,

`MultiplayerGameState.connect`, la fabrique statique de `MultiplayerGameState`, se connecte au serveur et s'assure de l'état valide de la connexion. Le cas échéant, cette fonction renvoie une exception `IOException`. Cela est pratique car, contrairement à un constructeur, en cas d'erreur il est possible d'empêcher l'instantiation de la classe, ce qui n'aurait pas été possible si le code avait été dans le constructeur directement. Enfin, il y a aussi l'avantage de l'encapsulation, le code est non seulement facile à changer mais l'intention est également claire. Lorsqu'on appelle la méthode `connect` on sait qu'une connection à un serveur distant va avoir lieu (`MultiplayerGameState.connect`), ce qui n'aurait pas forcément été évident à voir s'il y avait juste une instantiation (`new MultiplayerGameState()`).

4.3 État

Suivant le livre du Gang of Four, le patron d'état est un patron de conception très utile dans de nombreux programmes. Ici, pour le jeu de Snake, ce patron est utilisé pour représenter l'état actuel du jeu. Il y a donc un état `MainMenuState` pour le menu principal, un état `GameState` pour tout état relatif à une partie de jeu en cours, et `SoloGameState` pour l'état du jeu lorsqu'il y a une partie solo.

L'utilisation des états exacerbe un bon cas d'usage pour la POO (c.f. les diagrammes de classe pour les états) et encourage une bonne encapsulation. Ainsi, `GameState` est assez bien encapsulée pour que l'un de ses sous-types, `ServerGameState`, soit utilisé pour représenter l'état du jeu du serveur, alors que l'autre sous-type, `SoloGameState`, soit utilisé pour représenter l'état du jeu du client en partie solo.

Les avantages de ce patron de conception sont donc, comme vu précédemment, multiples et cela explique aussi pourquoi ce patron de conception est plus largement utilisé dans beaucoup de jeux viédos et même beaucoup de programmes plus généraux (comme des éditeurs d'images, d'audio, etc.) car les GUI et autres IHM se prêtent en général très bien à la notion d'état.

4.4 Memento

Dans l'implémentation de notre menu principal pour un jeu multijoueur, nous avons rencontré un défi crucial lié à la préservation des états lors de la transition entre le menu principal et le jeu multijoueur. Plus précisément, lorsqu'un joueur se connecte à un serveur pour rejoindre une session de jeu, les paramètres spécifiques au jeu, encapsulés dans l'état de jeu `MultiplayerGameState`, doivent être chargés depuis le serveur. Cependant, ce contexte de jeu `MultiplayerGameState` ne conserve pas de manière inhérente les paramètres initiaux du client.

Afin de remédier à cette lacune et d'assurer la préservation des paramètres d'origine du menu principal, le patron de conception Memento a été appliqué. Ce patron a permis de concevoir un objet Memento associé à MainMenuState qui agit comme une capsule temporelle, stockant précisément les paramètres initiaux du menu principal.

Lorsque le joueur termine sa session multijoueur et retourne au menu principal, cet objet Memento a une importance capitale. Il est alors utilisé pour restaurer l'état initial du MainMenuState, garantissant ainsi la continuité des paramètres définis par le joueur.

5 Raisons pour l'Utilisation de Certaines Classes

Les classes les plus importantes créées pour ce projet sont:

5.1 `Snake`

Cette classe s'occupe de la gestion du serpent dans le jeu de snake. Elle a pour attributs la position, la vitesse, les segments du serpent, le skin, etc. La classe `Snake` représente à elle seule

le joueur d'un serpent, sans IA. C'est pourquoi cette classe a un sous-type `SnakeAutonome` qui implémente la méthode `updateAi()` qui s'occupe de déplacer la tête du serpent et le reste du corps du serpent. Le choix a été fait d'utiliser une méthode `updateAi()` qui est vide dans `Snake` (on parle de stub) et qui puisse être surchargé par une classe héritante (`@Override`).

5.2 `Skin`

Cette classe est implémentée en tant qu'énumération, permettant ainsi un choix fini de skins à supporter. L'implémentation de la classe de cette manière permet également d'alléger l'espace mémoire que prend la sérialisation, ainsi, plutôt que de devoir sérialiser 3 images entières (la tête, le corps et la queue), il n'y a que le nom de l'énumération à sérialiser. Ça a donc un grand intérêt à la sauvegarde dans `settings.json` et à la mise sur réseau.

5.3 `Segment`

La classe `Segment` et ses sous-types sont un bon exemple de l'utilisation de la POO dans ce projet. `Segment` est une classe abstraite, avec 3 méthodes: `causesCollision()`, `onCollision()` et `onCollided()`. Ainsi, chaque sous-type (c'est à dire `ShieldSegment`, `DefaultSegment`, `WeakSegment`) réimplémente ses méthodes.

5.4 `Server`

La classe `Server` est la classe qui implémente le serveur multijoueur. Pour faire le multi-threading nécessaire à cela, a été utilisée la classe de Java `ExecutorService`. C'est une classe qui permet d'exécuter tout objet `Runnable` en multi-threading. La classe `Executors` est utilisé pour instancier les objets `ExecutorService` en fonction des spécifications de multi-threading voulu. Pour le serveur du jeu de Snake il a été décidé d'utiliser la fabrique statique `Executors.newCachedThreadPool()` qui crée un `ExecutorService` qui exécute les tâches sur un nouveau thread mais essaie de réutiliser les threads précédemment créés mais qui sont actuellement inutilisés. Cela permet donc d'économiser sur le nombre de Threads créés sur la JVM.

5.5 `State`

Comme évoqué plutôt dans le Chapitre 4.3, c'est la mise en classe du patron de conception État. Chaque État peut implémenter plusieurs méthodes:

- `void update()`: méthode appelée 30 fois par seconde et qui doit s'occuper de la mise à jour de l'état.
- `void paint(Graphics)`: méthode qui doit afficher l'état à l'écran (par exemple, dans le cas de `GameState`, afficher le jeu)
- `void onMouseMove(int, int)`: déplacement de la souris
- `void onMouseClick(int, int)`: clic de la souris
- `void onWindowResize(int, int)`: redimensionnement de la fenêtre
- `void onKeyPressed(int)`: pression d'une touche
- `void onKeyReleased(int)`: relachement d'une touche