ESIPE – IR1

Projet de Java

Curvy Snake

COLLOMB Jérémie – GRISET Valentin
2015

Sommaire

1. Présentation

Le principe du jeu est très simple : il faut diriger la tête du serpent. Le serpent est dirigé par l'utilisateur grâce au clavier. **L'objectif est de rester vivant le plus longtemps possible.** C'est-à-dire que dès que la tête du serpent heurte un bord du jeu, ou que celle-ci heurte son propre corps, le jeu est terminé.

Le serpent est dirigé avec les touches : **ARROW LEFT** (gauche) et **ARROW RIGHT** (droite). La direction sur laquelle l'utilisateur influe est celle perçue par la tête du serpent.

Exemple : si le serpent va de haut en bas, et que l'utilisateur utilise la touche pour la direction de droite, le serpent va tourner sur la gauche (du point de vue de l'utilisateur – mais sur la droite, de son point de vue).

En tant que développeur, l'objectif du projet est de mettre en pratique et de revoir les connaissances vues en cours de programmation en Java. Ce projet permet également d'évaluer la répartition des tâches, de gérer le versionning, et de découvrir d'autre aspect de la programmation en Java.

2. Architecture logiciel et Organisation

La représentation du découpage est la suivante :

```
 Root Folder
|
|--> fr.upem.ir1.curvysnake
|    |
|    |--> controller
|    |    |
|    |    |--> exception
|    |    |    |
|    |    |    | BonusException.java
|    |    |    | CollisionException.java
|    |    |    | GameSizeException.java
|    |    |
|    |    | Bonus.java
|    |    | BonusAvailable.java
|    |    | BonusListInGame.java
|    |    | Entry.java
|    |    | Movement.java
|    |    | MoveTo.java
|    |    | Snake.
|    |
|    |--> view
|    |    |
|    |    | MultiPlayer.java
|    |    | SinglePlayer.java
|    |
|    | Main.java
```

Le développement a été découpé en deux parties : le cœur et le graphisme. L'intérêt de réaliser d'abord le cœur permet de structurer complètement le fonctionnement de l'application. Ainsi, nous avons réalisé un API la plus intuitive et simple possible. Une fois la partie cœur terminée, nous nous sommes attelés à réaliser une partie graphique utilisant les objets. Cette implémentation graphique correspond uniquement à l'utilisation de l'API avec la librairie Zen.

Dès le début, nous nous sommes penchés vers quelques choses de multijoueur. C'est donc pour cette raison que les collisions sont détectées entre les différentes instances de Snake. C'est donc pour cette raison que la collision est détectée en interne à la méthode move().

Pour la partie Bonus, nous nous sommes inspirés des différents Bonus présents dans CurveFever.

La totalité du projet a été versionée grâce à git (via GitHub). Le projet est dès à présent disponible en version open source (MIT) sur GitHub.

Nous ne sommes pas parties tête baissée dans le code. C'est pour cette raison que nous avons réalisé un digramme de classe complet, permettant de mieux comprendre et cadrer le projet.

Sous le conseil de nos professeurs, nous avons essayé de penser complètement objet. Et cette pour cette raison que l'API est complètement transparente et répond aux critères objet :

- Une classe, un élément, des méthodes associées (à l'exception de la détection de la collision des Bonus)
- Abstraction le plus possible (exemple : utilisation de List au lieu de montrer l'utilisation d'ArrayList).

La librairie qui nous a été mise à disposition n'est pas des performantes, ni des plus complètes. C'est pour cette raison que nous n'avons pas pu réaliser des fonctionnalités très poussées.

La totalité du projet a été réalisée en « Work in Pair ». C'est-à-dire que nous étions deux sur le même clavier. Cela nous a permis de maitriser complètement le projet, ainsi que d'améliorer notre qualité de code.

3. Structure des données

La classe Entry est une représentation d'un couple clé/valeur.

Les Exceptions ne représentent que des situations d'exception pour des classes développées. Par conséquent, aucun traitement n'est réalisé par ces classes. Il est donc envisageable de mettre en place une gestion plus complète de celles-ci. Elles sont attrapées que lors de leurs gestions : par exemple, la collision est gérée au tout dernier moment.

Le corps du serpent est représenté par une liste de cercle. Il est ainsi stocké dans la classe Movement. Celle-ci n'est accessible que grâce à une classe présente dans le package controller. Il est donc ainsi nécessaire d'implémenter une classe utilisant le Movement.

Celle-ci prend en compte :

- les collisions avec : un mur et un corps de serpent
- la récupération de la tête et de la queue
- le déplacement du corps
- la traversée de mur
- l'effacement quasi complet du corps.

La collision est répartie en deux types : mur et autre corps. La collision a été pensée pour que l'on puisse détecter la collision entre plusieurs Snake. La méthode de déplacement prend en paramètre 2 listes : une contenant les positions ajoutées et une autre pour les positions supprimées. Le corps du Snake est rallongé une fois sur deux. De plus, la méthode met en application les bonus du Snake.

Le Snake est un stockage d'un Movement, avec une gestion de la direction et des bonus.

La direction est représentée avec un angle alpha variant sur 360 degrés.

Les bonus sont décrémentés tous les tours (1 tour = 1 ms~). Ils sont tous interprétés pour être passés à Movement sauf EraseAll (effacé tout), qui lui est exécuté que lors de sa récupération.

Le serpent a été pensé pour du multijoueur. C'est-à-dire que dans la classe, en static, on stocke une référence sur toutes les instances de Snake. Par conséquent, dès que l'on veut supprimer une instance, il faut utiliser la méthode destroy(). Celle-ci a pour effet de libérer l'élément de la liste d'instance.

Le fait d'avoir ce système permet de détecter automatiquement les collisions entre les serpents sans aucune contrainte, et ce très simplement. Ce traitement est totalement transparent. En effet, c'est la méthode Movement::move() qui gère cet aspect. D'où la gestion masquée et simplifiée en static.

Cette partie peut très largement être exportée et simplifiée.

Les bonus possèdent des caractéristiques. Les attributs des bonus ne peuvent pas être modifiés. Par conséquent, dès que l'on modifie un attribut, une nouvelle instance est retournée. Cela a pour effet de créer une nouvelle instance d'un bonus très facilement. De plus, une méthode pour décrémenter le temps est en place. Celle-ci modifie l'attribut de temps. Ainsi, dans ce cas uniquement, aucune instance n'est nouvellement créée.

Une liste de bonus possible (en Enum) est mise en place. Cette liste permet de générer automatiquement un bonus parmi la liste, et permet aussi de regrouper et simplifier l'utilisation des différents types de bonus. De plus cette classe, offre directement un système de duplication pour éviter que l'on utilise le bonus en direct.

En revanche, si l'on ne passe pas par le système de duplication, le temps peut être modifié, directement dans l'Enum...

Une liste de bonus actuellement en jeu est aussi mise en place. Cette liste permet de simplifier complètement le système des bonus dans le jeu. Cette classe permet de détecter les collisions entre un Snake et un bonus, ainsi que gérer le système d'apparition des bonus.

4. Possibilité d'évolutions

Le projet est pensé pour obtenir des évolutions le plus simple et le plus rapidement possible. La classe « Player » nous permet d'intégrer un nouveau joueur facilement et donc une évolution possible est le Multiplayer (2 joueurs) que nous avons commencé à implémenter, mais celui-ci pourrait s'ouvrir à un nombre supérieur de joueurs. Egalement dans la classe « Player » un attribut score peut être initialisé et augmenté au fil d'une partie. Celle-ci n'est pas utilisée, car nous étions obligés de dessiner le score. En plus d'une partie en multijoueur en local, nous voulions insérer le multijoueur en réseaux pour apprendre à programmer en java et en réseaux.

Notre classe « Snake » représentant notre serpent peut être retravaillée afin de l'optimiser et de la séparer en différentes classes (exemple : détection de collisions, gestion de la direction).

La classe Bonus peut aussi être améliorée également en évitant le parcours de notre Snake pour ajouter un nouveau bonus. Par exemple, que chaque bonus est disponible qu'un certain temps sur le plateau et qu'il disparaisse avec le temps.

Nous voulions créer une interface avec un menu proposant les différents types de partie (1 joueur, plusieurs jouer). Afficher la liste des bonus actifs sur notre serpent avec le temps restant pour chaque bonus. Et afficher un mur contant durant la partie et qu'il se supprime dans le cas du bonus « Wallthrough ».

5. Difficulté rencontrée

Pour débuter le projet, nous sommes parties sur une architecture avec un serpent (classe « Snake ») composé d'une liste de classe « Body » qui représentait un cercle avec comme attribut une position, avec un radius. Cette première architecture pour notre Snake était bonne, mais pouvais être simplifié avec l'utilisation de la classe « Ellipse2D » qui génère une ellipse avec une position « x, y » et une taille « width » et « height ».

Nous avons appris qu'une classe « Point » représentant un couple « x, y » était disponible dans la bibliothèque (java.awt). Ces deux bibliothèques nous ont permis de nous simplifier le développement en n'ayant pas à redéfinir des méthodes nécessaires.

Le cœur de notre projet étant créé, nous avons décidé d'implémenter l'interface graphique en nous appuyant sur la démo fournie. Une première compréhension des lambda était nécessaire afin que de comprendre l'interface.

Une fois l'interface graphique implémentée ne nous sommes retrouvées face à des problèmes de latence sur l'affichage. Sois le serpent avançait de manière saccadée, sois la réponse des touches de direction était lente surtout en cas d'appuie continue sur celles-ci.

Après plusieurs corrections de bug ou d'erreurs sur l'effet des bonus nous sommes parvenus à obtenir un jeu fluide sur une machine. Cependant après différents tests nous n'obtenons pas la même fluidité sur différente machine Windows, aucune réponse des touches de direction sur les machines Linux. Par manque de temps nous n'avons pas pu aller plus loin dans le projet comme nous l'aurions souhaité.

6. Conclusion

Ce projet fut pour nous un moyen de mettre en pratique les connaissances acquises sur ce second semestre en Java. De plus il a permis à Valentin de revoir certaines notions de la programmation-objet qu'il ne maitrisait pas, avec l'aide de Jérémie.

Ce projet nous a apporté certaines compétences d'organisation et de travail collaboratif. Avec l'utilisation de Git, nous avons pu travailler chacun sur des classes différentes après avoir travaillé ensemble sur les classes mère (exemple : Snake, Bonus). Le projet a aussi permis de mettre en pratique la répartition des tâches, l'écoute de chacun : comment la personne voit le projet, et comment mettre en relation les différentes propositions, pour avoir la meilleure solution. Nous aurions aimé aller plus dans ce projet afin d'en apprendre plus et de voir davantage de moyen pour optimiser celui-ci. Nous avons trouvé le projet amusant dans son ensemble, surtout lors des parties jouées où chacun essayait malgré certains bonus assez aléatoires de rester en vie.