Rapport de projet n°1

Nathan Monsoro

Introduction :

L'objectif technique de ce projet est de nous familiariser avec la structure MVC. Nous avons donc pour but de créer un mini jeu semblable au bien connu Flappy Bird. Le principe de notre projet sera le suivant : un ovale qui monte et descend, devra rester sur une ligne brisée qui défile. Pour faire monter l'ovale, l'utilisateur n'aura qu'à cliquer avec la souris dans la fenêtre du jeu. L'ovale redescendra tout seul au cours du temps. Pour réaliser cela, on commencera par se familiariser avec les classes JFrame et JPanel d'abord créer une fenêtre vide, puis avec un ovale statique. On devra ensuite utiliser la classe MouseListener pour que notre programme réagisse aux clics souris. Grâce à cela, nous pourront faire monter notre ovale avec la souris. Il nous faudra ensuite automatiser la redescente de l'ovale ainsi que la création et le défilement de la ligne brisée. Pour cela, on utilisera les bibliothèques Thread et Point. Grâce à ces bibliothèques, nous pourrons calculer le score de l'utilisateur, et lui donner l'impression que la ligne brisée est infinie. Finalement on détectera quand l'ovale sera hors de la ligne afin d'afficher l'écran de défaite.

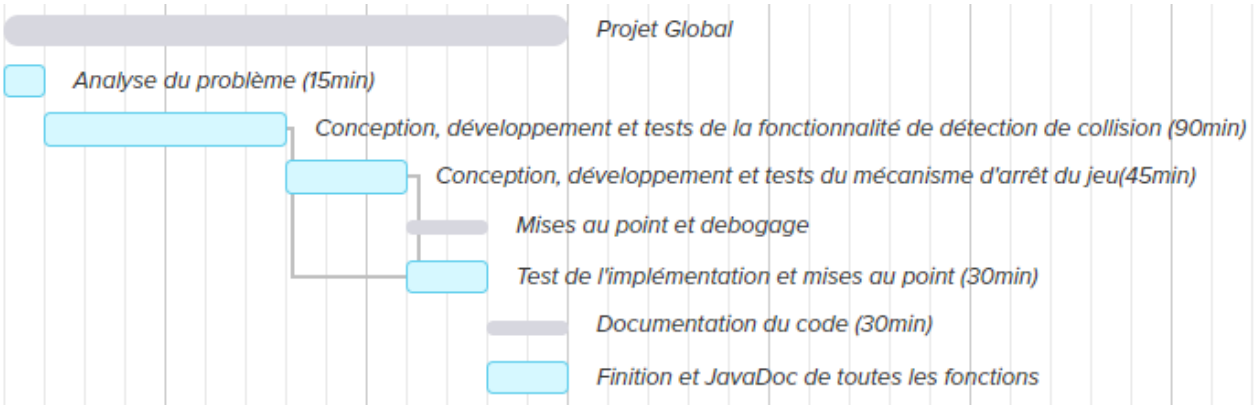
Analyse globale :

Nous pouvons identifier trois groupes principaux de fonctionnalités dans ce projet de mini-jeu : d'abord l'interface graphique qui affichera l'ovale ainsi que la ligne brisée que ce dernier devra essayer de suivre dans la fenêtre, puis le défilement automatique de la ligne brisée, et enfin les déplacements de l'ovale. Les déplacements de l'ovale sont soit des réactions aux clics souris de l'utilisateurs pour le faire monter, soit une descente automatique lorsqu'on ne le fait plus monter pendant un temps. Dans un premier temps on s'est occupé de la création de la fenêtre avec le dessin de l'ovale, puis le déplacement de l'ovale vers le haut lorsque l'utilisateur fait un clic à la souris dans la fenêtre du jeu. Ces deux fonctionnalités sont sûrement les plus simples du mini-jeu, ce qui en font de bonnes premières implémentations pour découvrir le modèle MVC. Évidemment, l'étape de création de la fenêtre bien qui pas spécialement compliquée est capitale dans notre projet, puisque sans cela, aucun espace pour afficher notre jeu, ni pour interagir avec l'utilisateur. Il nous faut ensuite dans cette fenêtre dessiner l'ovale. Cette étape est aussi très simple, mais importante aussi puisque l'ovale est l'élément que le joueur va le plus regarder, puisque c'est sa position par rapport à la ligne brisée qui va définir s'il est en train de gagner ou en mauvaise posture. Cet ovale représente le joueur. La fonctionnalité qui fait « sauter » l'ovale est aussi très importante, puisque faire monter l'ovale est la seule interaction que

l'utilisateur peut avoir avec le jeu. C'est donc par la seule commande du jeu, le clic souris, que l'utilisateur devra se déplacer de son mieux pour gagner. Lors de cette séance, on ne s'est occupé que des sous fonctionnalités qui suivent. On s'est ensuite occupé de la génération et affichage de la ligne brisée ainsi que son défilement, et la descente de l'ovale. Pour la génération infinie de la ligne, on s'est servi des bibliothèques Random et Point. En effet on génère des points atteignables depuis le point précédent. Pour cela, on a dû calculer le coefficient de chute de l'ovale en chute libre. Après cela, on a juste eu à s'assurer que le point crée n'était pas sous la droite de même coefficient passant par le point précédent. Pour ce qui est du défilement, on a utilisé la bibliothèque Thread. Grâce à cette bibliothèque, à intervalle régulier, on incrémente la variable position de la classe parcours qui représente la distance par rapport au départ ainsi que le score de l'utilisateur. En incrémentant cette variable, cela modifie les coordonnées des points que l'on envoie à l'affichage, ainsi on voit la ligne brisée bouger puisque l'on faisait mettre à jour l'affichage à chaque passage. Lorsque l'on voit que le dernier point que l'on utilise pour le tracer des segments est bientôt visible dans la fenêtre, on ajoute un point après lui afin de créer un autre segment et ainsi ne pas interrompre la ligne brisée. Pour ce qui est de la chute de l'ovale, on utilise aussi la bibliothèque Thread. À chaque intervalle de temps, on modifie la hauteur de l'ovale pour donner une impression de chute. Lors de cette séance, on a travaillé la détection de collision, afin d'implémenter la seule condition de défaite du joueur. Pour cela on a créé une fonction testPerdu dans la classe Etat, qui sera appelé dans les classes Voler et Avancer et fait en sorte qu'une fois qu'il a perdu, le joueur ne peux plus bouger l'ovale.

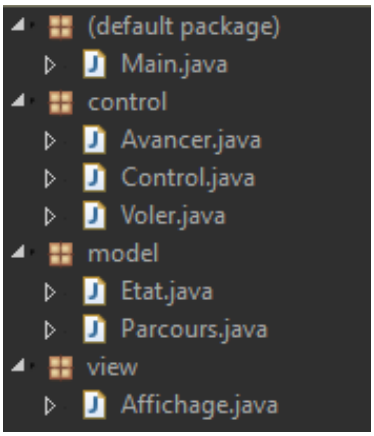
Plan de développement :

Mon temps de travail s'est à peu près réparti comme suit : dans un premier temps j'ai pris environ 15 minutes à analyser le problème, les compétences que j'avais à acquérir et à réutiliser et ce que donnerait le projet final. J'ai ensuite réfléchi à la conception et au développement de la fonctionnalité que nous avons implanté, qui est la détection de collision. J'ai aussi fait plusieurs tests avec différentes équations de la ligne et de la représentation de l'ovale. Cette étape m'a pris environ 1h et 30 minutes. Après cela, j'ai passé environ 45 minutes à la création de l'écran de fin et aux tests du mécanisme de défaite. Ces fonctionnalités sont importantes, et m'ont paru assez difficile à mettre en place car elles impliquaient des mathématiques, et faire une séquence d'arrêt « propre » n'est pas chose aisée. J'ai donc pris mon temps au débogage pour que l'arrêt soit le plus efficace possible. Finalement, j'ai mis environ autant de temps à documenter les modifications que j'ai apporté projet. La documentation fut moins longue qu'à la séance précédente j'y ai passé environ 30 minutes. J'ai passé autant de temps au débogage qu'à la séance précédente, environ 30 minutes en tout.



Ci-dessus le diagramme de Gantt qui représente la répartition de mon temps sur les 3h30 de la séance

Conception générale :



Lors de ce projet, nous avons dû nous familiariser avec une architecture de code toute nouvelle pour nous, le modèle MVC. (Vous trouverez ci-contre l'organisation en package des différentes classes du projet.) Nous avons donc organisé notre code dans 3 packages différents comme ci-contre, et laissé la classe Main dans le package de base. La classe Etat représente l'état du jeu à afficher, elle contrôle la position de l'ovale et le parcours de la ligne brisée, elle gère aussi les conditions de défaites du joueur. La classe Control gère les entrées de l'utilisateur, elle appelle la fonction de saut de la classe état quand l'utilisateur clique avec la souris, et appelle la fonction qui met à jour l'affichage dans la fenêtre. On trouve dans le même package, les classes héritées de Thread, Avancer et Voler. Ces classes gèrent respectivement le défilement automatique de la ligne brisée et la chute de l'ovale. Enfin la classe Affichage s'occupe d'afficher l'état du jeu. Elle crée la fenêtre, dessine l'ovale et la ligne brisée et affiche le score de l'utilisateur. Cette classe affiche aussi l'écran de fin avec le score final de l'utilisateur.

Conception détaillée :

Nous allons dans cette partie parler de façon plus détaillée de l'implémentation des fonctionnalités.

Lors des séances précédentes, voila ce que nous avons implémenté.

D'abord l'affichage de la fenêtre et de l'ovale. Pour cette fonctionnalité on utilise l'API Swing. On a aussi besoin de la classe Affichage qui hérite de JPanel, et de la classe Etat. Les dimensions de la fenêtre que l'on veut créer sont stockées dans les constantes xS et yS de la classe Affichage. Les dimensions de l'ovale sont, elles, dans les constantes xC, yC, w et h de la classe Etat. Pour afficher la fenêtre on crée directement dans la classe Main une JFrame auquel on ajoute l'affichage. Grâce à l'attribut etat de la classe, on a relié l'état du jeu à notre affichage.

Nous avons aussi implémenté le déplacement de l'ovale sur l'axe horizontale dans la fenêtre. La hauteur d'un saut est définie dans la classe Etat par la constante jumpH. La fonction jump() qui fait monter notre ovale est appelé dans la classe Control qui implémente la classe MouseListener, lorsque l'utilisateur fait un clic de souris dans la fenêtre. Après cela, le contrôleur demande à la classe Affichage de mettre à jour la fenêtre avec la fonction repaint(). Dans la fonction jump(), on vérifie si l'ovale ne sortirait pas du cadre après le saut, si c'est le cas, on le colle simplement à la bordure supérieure de la fenêtre, sinon on le fait monter de la valeur de jumpH.

Ensuite la chute de l'ovale. Pour cette fonctionnalité on utilise la bibliothèque Thread. On a aussi besoin de la classe Affichage qui hérite de JPanel, et de la classe Etat. L'intervalle de temps est donné en paramètre, après chaque intervalle de temps, on appelle la méthode moveDown de la classe Etat afin de faire descendre l'ovale. On donne la hauteur de la fenêtre en paramètre pour éviter de devoir include la classe Affichage dans la classe Etat. Cela sert à ce que l'ovale reste dans la fenêtre. Après avoir fait descendre l'ovale, on demande à l'affichage de mettre à jour.

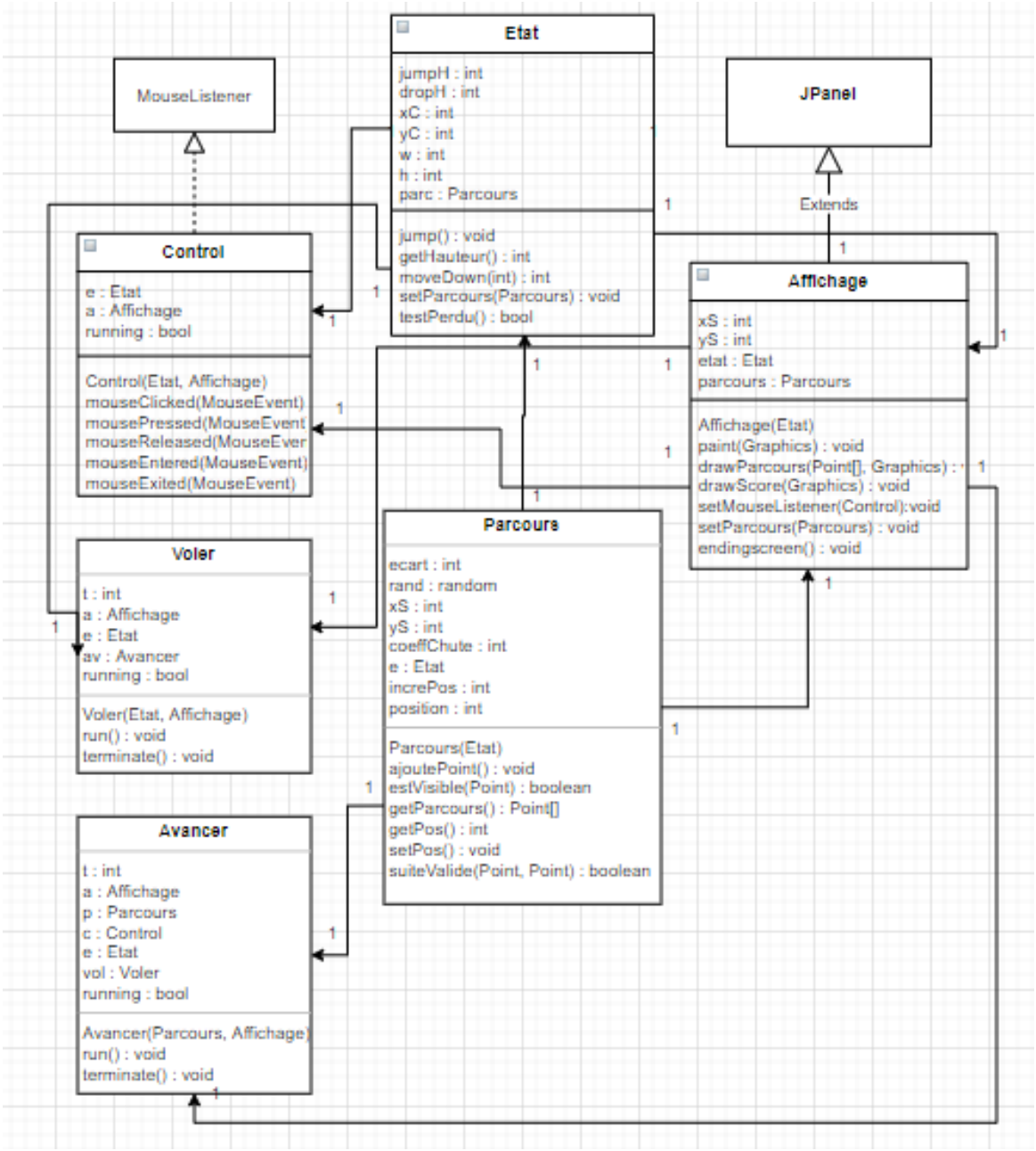
Nous avons aussi implémenté le défilement de la ligne brisée et sa génération infinie. Pour cela on a utilisé la bibliothèque Random et le type Point[]. On utilise aussi la bibliothèque Thread comme précédemment. Tous les intervalles de temps, on incrément la distance par rapport au départ d'un montant fixe en attribut de la classe. Puis on demande à l'affichage de mettre à jour l'affichage. À ce moment-là, l'affichage va demander à la classe Parcours la liste de points pour les segments à afficher. Alors on va passer en revue tous les points de sa liste, et passer en à l'affichage les points impliqués dans des segments visibles. Lorsque l'on voit que le point le plus a droite hors de la fenêtre pour afficher le segment le plus à droite, est bientôt visible, on en crée un autre à sa suite pour créer un autre segment et ainsi donner l'impression d'une ligne brisée infinie. À chaque fois que l'on rajoute un point, on s'assure que le point est atteignable avec le coefficient de la droite de chute libre de l'ovale.

Voici ce que nous avons implémenté lors de cette séance.

Dans un premier temps la détection de collision. Cette fonction est celle qui a nécessité le plus de calcul. En effet on a dû trouver un moyen de calculer l'ordonnée de points du segment entre les points du parcours que l'ovale vient de passer et celui qu'il va passer ensuite, et ce avec les positions relatives à l'ovale. Pour cela on devait dans un premier temps calculer le coefficient de la droite qui passe par ces deux points avec la formule suivante : (y2-y1) /2, y1et y2 étant respectivement l'ordonnées du point que l'on a déjà passé et celle de celui que l'on va passer. Avec ce coefficient, on a calculé l'ordonnée de la droite aux points d'abscisse 0 à w la largeur de l'ovale, par rapport à l'ovale. Pour cela on utilise le reste de la division entière de la position par rapport au début du parcours par l'écart entre chaque point. On a ainsi l'abscisse sur le parcours où se trouve le centre de l'ovale. Avec cela, on parcourt les abscisses de -w/2 à w/2 a partir du centre de l'ovale pour trouver l'ordonnée de la droite à toutes les abscisses ou est l'ovale. Pour simplifier, on a considéré l'ovale comme un rectangle dans cette partie. Ainsi si l'ordonnée de la ligne brisée est entre l'ordonnée de l'ovale et sa coordonnée + sa hauteur à au moins une des abscisses, on la considère dans l'ovale, et l'utilisateur ne perd pas. Lorsque que ce n'est pas le cas, l'utilisateur perd.

Le mécanisme de défaite et d'arrêt du jeu. La fonction qui test si l'utilisateur perd ou pas est appelé après chaque itération de repaint, et lorsque l'on détecte que le joueur a perdu, on appelle une méthode qui arrête tous les Thread et désactive les clics souris, et demande l'affichage de la fenêtre de fin. Il a fallu tester plusieurs organisations pour que cela cette partie marche correcte.

Ci-dessous le diagramme des classes du projet.



Résultat :

Voici ce que donne notre projet dans l'état actuel. On voit bien l'ovale et la ligne brisée, ainsi que le score. L'ovale ne sort pas ni quand il monte, ni quand il descend. On voit que lorsque l'ovale sort de la ligne le jeu s'arrête et affiche la fenêtre de défaite avec le score de l'utilisateur.

Documentation utilisateur :

Pour exécuter le code, il vous faudra un IDE avec Java pour l'exécuter à partir du code, ou bien Java pour l'exécuter à partir d'un fichier .jar.

À partir d'un IDE, importez simplement le projet, et à partir de la classe Main faites « Run as Java Application ». La fenêtre avec l'ovale s'ouvrira alors, vous pourrez faire monter ce dernier en cliquant dans la fenêtre.

À partir du fichier .jar, double cliquez dessus pour l'exécuter et ouvrir la fenêtre. Vous pourrez faire monter l'ovale de la même façon en cliquant dans la fenêtre.

Documentation développeur :

Pour continuer ce projet, il nous faudrait ajouter les textures pour le décor et une image animée pour l'ovale et le transformer en autre chose (un oiseau par exemple). On pourrait aussi arrondir les angles afin de les rendre moins abruptes et plus facile à manœuvrer, ça serait aussi sûrement plus esthétique.

Conclusion et perspectives :

Jusqu'ici, nous avons codé la fenêtre, l'ovale qui s'y déplace et la ligne brisée qui défile à l'infini. Nous avons utilisé le format MVC, les bibliothèques Swing, Thread et Random. Nous pouvons détecter les collisions et afficher un écran de fin avec le score. On pourrait ajouter des textures pour aller plus loin.