**Livrable du projet Pac-Man en Java**



De Charentenay Henri

Taieb Matthieu

G7

1. Rappel du sujet

Le but du projet est de créer une réplique simple du jeu Pac-Man. Le but du jeu est, pour le joueur dirigeant le pacman symbolisé par un rond jaune, de manger tous les fromages répartis sur les cases du labyrinthe, en mettant le moins de temps possible. Les Fantômes sont des entités qui parcourent le labyrinthe au hasard. Si le joueur sur la même case qu’un des fantômes, alors le joueur doit recommencer à son point de départ et une vie lui est enlevée. La partie s’arrête lorsque le joueur à manger tous les fromages du labyrinthe ou lorsqu’il a perdu ses trois vies.

Le pacman avance automatiquement dans la direction choisie par le joueur avec les flèches du clavier. Lorsque le joueur choisit une direction, le pacman se tourne et avance dans la nouvelle direction. Lorsqu’un mur bloque le passage du pacman, il arrête d’avancer.

Lorsque la partie est terminée, on affiche le score du joueur et on demande si l’utilisateur souhaite rejouer.

1. Modélisation

Pour réaliser cette réplique du jeu, nous avons créé un nouveau projet sur Eclispe dans lequel nous avons rajouté la librairie StdDraw qui nous servira pour la partie graphique du jeu.

Notre projet se décompose en cinq classes différentes servant à gérer le labyrinthe, le joueur, les fantômes et les graphismes du jeu. Nous allons expliquer ces 5 classes une par une.

La classe du labyrinthe :

Notre labyrinthe de jeu est en fait une matrice de 30 lignes et colonnes de nombre entier.

Les cases de notre matrice remplies d’un 0 représentent les cases sur lesquels se trouve un fromage alors que les 8 sont des cases vides sur le labyrinthe. Les 1 représentent les murs du labyrinthe.

Les cases 3 sont des cases spéciales qui ne nous servent uniquement à afficher le score et les vies du joueur.

Le tableau étant modifié en temps réel pendant le jeu, une deuxième version de la matrice qui ne se modifie pas est présente pour permettre de réinitialiser le plateau de jeu lorsque le joueur désire rejouer après la fin d’une partie.

La classe du joueur :

On importe deux bibliothèques que nous allons utiliser pour le deplacement du joueur « java.awt.event.KeyEvent » et « edu.princeton.cs.introcs.StdDraw ».

* Les variables de classe :

On a défini cinq entiers pour cette classe. Les deux premiers, « x » et « y », correspondent aux coordonnées du joueur dans le labyrinthe, deux autres entiers, « ancienX » et « ancienY », qui servent à enregistrer les anciennes coordonnées du joueur pour son mouvement. On utilise des getters et setters pour les coordonnées du joueur.

Le cinquième entier, appelé « mouvement », sert à détecter s’il y a eu un mouvement ou pas.

On utilise aussi quatre booléens qui nous servent dans la fonction de détection des murs, appelés « Nord », « Sud », « Est » et « Ouest ».

* Le constructeur de classe :

On initialise les coordonnées du joueur à (23 ;14). Le tableau représentant le labyrinthe est modifié, un 2 apparaît dans la matrice aux coordonnées du joueur.

* La détection des murs « testMur » :

Le but de cette méthode est de pouvoir détecter les murs afin que le pacman s’arrête s’il ne doit pas continuer à avancer.

Les quatre booléens sont initialisés à faux. On utilise ensuite des boucles if pour tester les quatre cases du tableau à côté du joueur. Si la case est vide, contient un fromage ou un fantôme, le booléen correspondant à la case testée devient vrai, sinon il reste faux et le joueur ne pourra pas avancer.

Un cas particulier se présente dans le cas où le joueur se trouve dans le « tuyau » et qu’il souhaite se téléporter à l’autre côté du labyrinthe. Avant de détecter les murs, on a décidé de tester les coordonnées du joueur, s’il se trouve en (14 ;1) ou en (14 ;29), le booléen correspondant est directement vrai pour autoriser le mouvement, et sa position sera modifiée comme expliquée ci-après.

* Le déplacement du joueur « deplacement » :

Avant de commencer, les variables ancienX et ancienY sont changées par les coordonnées actuelles du joueur, et mouvement revient à 0.

La méthode testMur est appelée avant tout mouvement.

On utilise ensuite des boucles if pour connaître la direction demandée. Dans ces différentes boucles, nous avons rajouté une deuxième condition qui correspond à un des quatre booléens. Si les deux conditions sont remplies, les coordonnées du joueur sont actualisés et la variable mouvement prend la valeur 1.

Comme dans la méthode testMur, deux cas particuliers apparaissent, les deux cases de téléportations à chaque extrémité du labyrinthe. Si le joueur se trouve en (14 ;1) et qu’il souhaite aller à gauche, il est alors déplacé en (14 ;29). Si le joueur se trouve en (14 ;29) et qu’il souhaite aller à droite, il est alors déplacé en (14 ;1).

Si le joueur a pu se déplacer, la matrice du labyrinthe est modifiée et une case vide est laissée derrière le pacman.

La classe des fantômes :

Pour la classe des fantômes, nous avons besoin de la bibliothèque « java.util.Random ».

* Les variables de classe

Pour cette classe, nous aurons besoin de définir six entiers et quatre booléens ainsi qu’un string. Les quatre entiers seront utilisés pour les coordonnées et les anciennes coordonnées de chaque fantôme, « x » et « y », ainsi que pour connaître la dernière direction empruntée par les fantômes, « ancienneDirection », et savoir si le fantôme a pu bouger, « mouvement ». Comme pour la classe du joueur, les booléens servent à la détection des murs, « Nord », « Sud », « Est » et « Ouest ». Le string sert pour définir la couleur du fantôme, « couleur ».

On utilise des getters et setters pour les coordonnées des fantômes ainsi que pour l’ancienne direction.

* Le constructeur de classe :

Le constructeur de classe que nous utilisons dépend d’une variable qui correspond au string « couleur » cité plus haut. Ce string nous sert à définir les différentes coordonnées de base des fantômes, à savoir (13 ;14), (15 ;14) et (17 ;14), qui correspondent à 3 emplacements dans la petite boite au centre du plateau de jeu, qui correspond à la «maison» des fantômes.

L’entier qui définit l’ancienne direction emprunté par le fantôme prend la valeur 0 pour laisser la possibilité au fantôme de se déplacer n’importe où au premier déplacement.

* La détection des murs « testMur » :

La méthode de détection des murs est la même que pour celle du joueur, le cas de téléportation aussi.

* La phase de mouvement random du fantôme « deplacementRandom » :

Pour le déplacement aléatoire du fantôme, nous utilisons la bibliothèque « java.util.Random » pour générer un nombre entre 0 et 3, cette variable s’appelle « dep ». La valeur donnée par la génération est ensuite utilisée dans un switch.

Dans les quatre cas, on vérifie que les fantômes peuvent se déplacer dans la direction choisie grâce à « testMur() » et que l’ancienne direction n’est pas l’opposée (pour ne pas avoir un comportement ridicule du fantôme où il ferait des petits aller-retour dans un couloir). L’utilisation de la variable d’ancienne direction permet d’imposer aux fantômes de ne pas revenir sur leurs pas (pour ne pas avoir un comportement ridicule du fantôme où il ferait des petits aller-retour dans un couloir).

Dans le déplacement aléatoire du fantôme, trois cas particuliers se dessinent, les deux premiers concernent la téléportation quand le fantôme se trouve au bord du labyrinthe. Le dernier cas particulier concerne la sortie de la cage du départ. Les fantômes se déplacent aléatoirement à l’intérieur de la cage jusqu’à se trouver sous les trois cases spéciales correspondant à un 9 dans le tableau. Dans ce cas, le fantôme est immédiatement téléporté une case au-dessus de la cage et ne peut plus y retourner (on notera que le joueur non plus).

* La phase de mouvement intelligent du fantôme

Après plusieurs essais de partie, nous avons décidé d'implémenter une petite condition sur le comportement des fantômes, afin de relever légèrement la difficultée. Pour cela, lorsqu’un fantôme se trouve a des coordonnées suffisamment proche du joueur, il entre dans une phase intelligente. Si la somme de la valeure absolue de la différence des abscisses et des ordonnées du joueur et d’un fantôme passe en-dessous d’un certain seuil, ce fantôme cherchera à se rapprocher du joueur. Si son ordonnée est strictement supérieure à celle du joueur, sa direction deviendra automatiquement « sud » (et inversement : « nord » si son ordonnée est inférieure à celle du joueur). De même pour l’abscisse avec « est » et « ouest ». La difficulté est ainsi un peu augmentée, et la situation où les fantômes sont tous à l’opposé du joueur devient de plus en plus rare.

La classe Main:

Dans notre classe contenant la méthode main, nous avons implémenter 3 autres méthodes, une pour l’affichage d’un menu de démarrage et les deux autres concernant la victoire et la défaite. Lorsqu’une partie est terminée, nous avons permi au joueur de pouvoir recommencer s’il le souhaite.

* Le menu de démarrage “page Accueil”

Notre menu s’affiche en exécutant le projet. Grâce à deux boucles if détectant les entrées au clavier, le joueur peut décider de lancer la partie ou non.

* Le test de victoire “testVictoire”

Nous avions le choix de paramétrer la victoire grâce aux nombres de cases vide sur la carte ou avec le score. Nous avons choisi la première solution car il permettait de pouvoir recommencer plus facilement une partie, le labyrinthe étant réinitialisé. Le score est alors calculé en fonction du nombre de case vide dans le labyrinthe puis affiché.

Lorsque toute les cases sont vides, le booléen de victoire devient vrai est permet de sortir de la phase de jeu.

* Le test de défaite ‘testDefaite”

Pour la défaite, nous avons décidé de comparer les abscisses et ordonnées de chaque fantômes avec celle du joueur. En cas de collision, le joueur et les fantômes retournent à leurs positions initiales et le joueur perd une vie.

Lorsque le joueur a perdu ses 3 vies, le booléen de défaite devient vrai et permet de sortir de la phase de jeu.

Difficultés rencontrées :

En tant que premier projet java, premier travail nettement plus complexe que nos TPs, la première difficulté a été de savoir comment commencer. Nous ne savions pas par quelle classe commencer, comment allions nous construire le terrain, faire le déplacement etc… Nous avons donc commencé par convenir entre nous de passer par une grande matrice 30\*30, composée d'éléments de natures différentes que nous allions modifier avec les actions du joueurs.

Malgré les quelques TP que nous avons fait pendant le semestre, nous avons manqué un peu d'expérience, nous avons donc eu parfois un peu de mal à matérialiser et à mettre sur machine certaines idées ou certains projet que nous avions en tête.

Contrairement à certains autre groupe dont nous avons pu voir le travail, le déplacement de nos joueurs et de nos fantômes se fait case par case, sans animation donc de manière un peu saccadée. C’est visuellement un peu moins agréable mais cela nous demandait de modifier entièrement notre code, nous avons donc choisi de le laisser tel quel. Cela a en revanche le mérite d’être un peu plus léger pour les calculs concernant notre tableau.

L’utilisation d’un tableau pour représenter le labyrinthe nous a posé un autre problème. Au début, nos fantômes se déplaçaient dans le tableau comme le joueur. Lorsque nous avons implémenté les pastilles dans le jeu, le déplacement du joueur se faisait de façon simple. Quelque soit la case où le joueur se déplaçait, le joueur laisser derrière lui une case vide. Pour les fantômes, nous ne sommes pas à arriver à différencier les cases vides ou avec pastille pour les faire réapparaitre après le mouvement. Nous avons donc opté pour une deuxième solution, les fantômes ont été enlevés du tableau et nous les affichons “par dessus” le labyrinthe. Les mouvements et les changements d'abscisses ou d’ordonnées restaient les mêmes mais les pastilles ne disparaissent plus derrière un fantôme.

Le dernier problème concernait la mort du personnage. Lorsque le joueur et le fantômes échangeaient leurs cases (par exemple le fantôme passe de (10:10) à (10:11) et le joueur, en même temps, de (10:11) à (11:10)), le jeu ne détectait donc pas de collision et donc laisser le joueur vivre alors que visuellement, le fantôme passait à travers le joueur et il n’était possible de mourir que lorsque le joueur et un fantôme arrivaient sur une même case au même moment. Le problème s’est résolu lorsque nous avons implanté la phase intelligente des fantômes.

Pour conclure, nous avons abouti à un jeu fonctionnel mais perfectionnable, tant au niveau de l’intelligence des fantômes que pour les graphismes de la fenêtre, et nous nous avons acquis de l'expérience dans la programmation java.