Inass Ech-Chouini G43
Wilfrid Ait Amer
Guillaume G
Thomas Delmas

Compte Rendu du projet d'algorithmique

I. Cahier des charges

Le projet consiste à réaliser un jeu vidéo inspiré du jeu Flappy Bird (2013) en y introduisant de nouveaux éléments. Le joueur contrôle un cochon qui est soumis aux effet de la gravité et qu'il fait sauter grâce à la touche espace du clavier afin d'éviter les obstacles en forme de tubes, qui se déplacent de droite à gauche. Pour différencier notre version de l'originale, on intègre des bonus/malus tels que l'inversion de gravité et l'invincibilité, ainsi que des niveaux de difficultés. Le joueur échoue si le cochon rencontre un obstacle (tuyaux, sol). L'idée derrière ce projet est de réaliser une application interactive amusante mêlant la programmation orientée objet, la gestion d'interfaces homme-machine et des formules de mathématiques appliquées à la mécanique (gestion des collisions, calcul des vitesses, lois de Newton, etc). Au cours de l'écriture du code, nous avons décidé de changer le nom de notre jeu en : Flappig boy (en référence à la pièce des théâtre-études, *Pig Boy*, qui joue à la Rotonde du 29 Avril au 2 Mai).

II. Description du problème

Tout d'abord, pour faire fonctionner notre jeu, nous avons dû réaliser une étude mécanique pour le mouvement du système {cochon}. On pose le principe fondamental de la dynamique pour obtenir l'équation du mouvement du cochon selon sa variable y en fonction du temps. Le cochon est dans une situation de chute libre, donc soumis seulement à son propre poids selon l'axe y, il n'y a pas d'équation pour la variable x. On veut que le joueur puisse contrôler la position verticale du cochon pour passer entre les tuyaux. Il faut donc que nous introduisons une force ponctuelle que l'on appelle f, opposée au poids afin de faire remonter le cochon. Cependant si f est présente trop souvent, le cochon acquiert une accélération très importante. Il deviendrait alors incontrôlable et risque de sortir de la fenêtre du jeu.

Ensuite, nous devons gérer le système de collision du cochon avec son environnement. Ce dernier est composé de tuyaux que le joueur doit éviter au cours du jeu, et de bonus (ou malus) qui se déplacent sinusoïdalement. Il faut également gérer les limites de la fenêtre pour éviter que le cochon sorte de celle-ci.

Nous souhaitons que le programme conserve les scores des différents utilisateurs, afin de donner envie aux joueurs d'obtenir le meilleur score, à la manière d'un jeu d'arcade.

Enfin, pour rendre le jeu plus amusant, nous voulons donner la possibilité aux joueurs de choisir le niveau de difficulté avant de commencer la partie.

III. Principe de de l'algorithme

L'algorithme final est composé de dix classes distinctes, régissant chacune une fonction bien particulière du jeu.

La classe **Boule** possède comme attributs les coordonnées de position x et y de la boule (le cochon dans le jeu) ainsi que la gravité lui étant appliquée, sa masse, sa vitesse limite et sa taille indiquée par la taille d'un côté, la boule étant en réalité un carré que l'on couvrira par une image. Cette classe est dotée d'une unique méthode nommée **bouge**, qui comme son nom l'indique gère le déplacement de la boule. Cette méthode prend en entrée l'entier f, qui représente la force que l'on applique sur la boule pour la faire "sauter". L'accélération, la vitesse et la position verticale de la boule sont donc exprimées en fonction de la constante de gravité g, de la force f et de la masse m de la boule selon les lois de la dynamique. On impose également une vitesse limite ascendante et descendante que la boule ne pourra pas dépasser lors de son déplacement. On a décidé d'effectuer des calculs en prenant 1 pixel = 1 cm. Pour le déplacement, l'accélération et la constance de gravité sont donc en cm/s^2, la vitesse en cm/s, que l'on multiplie par l'intervalle de temps entre chaque réveil du timer en milliseconde, pour obtenir un déplacement en cm.

La classe **Objet** concerne les bonus et malus que le joueur pourra récolter en passant dessus avec la boule. Ces objets sont animés d'un mouvement sinusoïdal fonction du temps grâce à la méthode **bouge**, qui prend en entrée l'entier tps (temps). De plus, les objets de type Objet possèdent une méthode intitulée **collision**, qui établit si un objet est entré en collision avec la boule en retournant un booléen true si les conditions de collision en fonction des côtes de l'objet sont vérifiées, et false si elles ne le sont pas. Finalement la méthode **paint** appelant Graphics g de la classe **fenetreJeu** permet de dessiner ces objets à partir de l'image objet.png.

Nous avons par ailleurs décidé d'ajouter une classe **Sol** ayant une visée purement graphique. En effet celle-ci implémente une unique méthode **dessine** en fonction de Graphics g qui dessine de sol comme étant un rectangle vert remplissant le bas de la fenêtre en fonction de l'attribut hauteur que l'on définit selon notre goût.

Les tuyaux à travers lesquels le joueur doit faire sauter la "boule" sont gérés par la classe **Obstacle** ayant pour attributs la position x du tuyau, sa vitesse de déplacement, ses dimensions (hauteur et largeur), ainsi que les longueurs L1 et L2 respectivement des tuyaux du haut et du bas. Les valeurs L1 et donc L2 prennent pour chaque **Obstacle** des valeurs aléatoires, tout en gardant un espacement constant entre le tuyau du bas et du haut. On obtient alors un enchaînement de trous de positions différentes à travers lesquels le joueur devra faire passer la boule.

Comme dans les classes évoquées précédemment la méthode **dessine** permet de créer la représentation graphique du tuyau en dessinant un rectangle vert muni de contours noirs. Ces tuyaux se déplacent également constamment vers la gauche grâce à la méthode **bouge** qui décrémente à chaque appel la position x du tuyau selon la vitesse constante des tuyaux. En accord avec les règles du jeu Flappy Birds, le joueur perd si la "boule" entre en collision avec un des tuyaux. La méthode **collision** ayant comme entrée un objet b de type

Boule permet donc de déterminer si la boule et le tuyau sont entrés en collision à l'aide d'un code similaire à celui de la méthode **collision** de la classe **Objet**. On remarque également que la première ligne de cette méthode est: *if(!b.invincible)*, ce qui permet d'anticiper les différents avantages et désavantages apportés par les objets bonus/malus. En effet, si un joueur percute un **Objet** bonus qui confère à la "boule" une invincibilité le temps de quelques secondes, celle-ci pourra donc passer par dessus les tuyaux sans perdre, et il faut donc prendre en compte ce cas de figure dans la méthode **collision**.

Nous en venons à présent à la classe **fenetreJeu**, classe centrale du projet qui fait elle même appel à de nombreuses classes citées précédemment. On remarque tout d'abord que cette classe possède une multitude d'attributs de types différents.

On introduit aussi en début de programme la création de la fenêtre de jeu qui accueillera la suite de la classe. Dans le constructeur, on affecte les valeurs aux constantes du jeu spécifiques au niveau choisi par le joueur.

Intéressons nous de plus près au choix d'une LinkedList au lieu d'une ArrayList. En se plongeant dans la méthode **ActionPerformed**, qui est appelé à chaque réveil du chrono, on remarque qu'un nouvel obstacle est ajouté à la liste lorsque le temps atteint un multiple de 70ms. Puis l'on parcourt la liste listeObstacle pour appliquer la méthode **Bouge** à chaque obstacle afin de lui inculquer le mouvement voulu. On calcule le score du joueur au fur et à mesure de la partie en ajoutant un point au score à chaque fois qu'il dépasse une paire de tuyau. Cette instruction est codée en regardant à chaque fois la position de la boule par rapport au premier obstacle de la liste (d'où le *listeObstacle.get(0).x*). Par la suite on teste la collision de la boule avec l'obstacle; si celle-ci retourne le boolean true, alors la boule est morte, et le joueur a perdu (*boule.estMort = true*). Finalement afin de ne pas utiliser trop de stockage en gardant à chaque fois les **Obstacles** ayant défilé à travers l'écran, on retire en fin de méthode tous les obstacles se situant en dehors (à gauche) de la fenêtre de jeu. Nous voyons donc qu'au cours de l'utilisation de cette liste nous réitérons en permanence l'accès au premier élément de la liste (de complexité O(1) pour les LinkedList) ainsi que les insertions à la fin (O(N) pour LinkedList et ArrayList) et les suppressions en tête (O(1) pour les LinkedList et O(N) pour les ArrayList). Il paraît donc évident que l'utilisation d'une LinkedList est le choix le plus approprié pour gérer les obstacles.

Dans cette même méthode, on fait appelle aux méthodes **bouge** des obstacles, des nuages, et des objets. On vérifie également les collisions de la boule avec son environnement, et on applique les éventuels bonus ou malus obtenus.

La méthode **paint** permet, à l'aide de commandes usuelles, d'afficher les différents éléments graphiques sur l'écran tels que le ciel, les **Obstacles**, les images du sol et de la boule (dans les cas vivant/mort).

Enfin en dernier lieu nous utilisons la méthode **keyPressed** permettant de lier le clavier au jeu. En effet c'est en appuyant sur la barre espace que le joueur fait sauter la boule. Nous avons fixé un intervalle de temps entre chaque saut, afin d'éviter que la boule n'aille trop vite. La barre espace ne fait donc sauter la boule que pour des intervalles entre deux pressions sur la touche supérieurs à 5*30ms.

La classe **fenetreGameOver** est une classe typique d'interface machine, et il n'est donc pas crucial de rentrer dans les détails de chaque ligne de code utilisée. Il convient cependant de noter qu'elle est appelée dans la classe **fenetreJeu** lorsque la condition

boule.estMort retourne true. Elle apparaît donc pour signaler au joueur qu'il a perdu, soit parce qu'il a percuté un tuyau soit parce qu'il est sorti de la fenêtre de jeu. Cette classe possède deux boutons "Nouvelle Partie" et "Menu" permettant comme leurs noms l'indiquent soit de recommencer une nouvelle partie en ouvrant une nouvelle **fenetreJeu**, soit de retourner au menu principal en ouvrant une **fenetreMenu** que l'on présentera ultérieurement. Ces actions sont rendues possibles grâce à la méthode **actionPerformed**. De plus le score atteint lors de la partie précédente, ainsi que le meilleur score de la session est affiché dans un JLabel.

Dans cette précédente fenêtre, on fait appelle à la méthode de **Score**. Cet objet crée un fichier "historiques_scores", et la méthode **list** écrit dans ce fichier le nom du joueur, son score, et son niveau de difficulté, grâce à la méthode Bufferderwriter. Cependant, notre manière d'écrire dans un fichier n'est pas optimale. En effet, à chaque fois que l'on crée un fichier, celui sauvegardé précédemment est remplacé, et les données perdues. Nous avons donc fait appelle au constructeur de **Score** dans la classe principale, et l'objet **Score "**historique" est envoyé au fur et à mesure de la session dans toutes les fenêtres. De cette manière, les scores d'une même session (une seule ouverture du programme) sont enregistrés dans le fichier. Si on ouvre à nouveau le programme, les scores précédents sont supprimés.

Similairement à la classe **fenetreGameOver**, la classe **fenetreMenu** est une classe d'interface graphique. Celle ci comporte un textfield permettant d'entrer le username du joueur et de le sauvegarder afin de lui attribuer le score qu'il obtiendra lors de la partie. De plus les règles du jeu sont affichées et l'on peut choisir le niveau de difficulté de la partie (facile, moyen, difficile) à l'aide de trois boutons. Finalement l'appui du bouton "Jouer !" déclenche la fermeture de la **fenetreMenu** pour ouvrir une **fenetreJeu** (instruction dans la méthode **actionPerformed**)**.** Dans les attributs figure aussi player de type **Musique** qui fait appel à la classe éponyme. On appelle aussi un InputStream servant à déclencher le lecteur de musique. Ainsi comme on déclenche le lecteur de musique dans les instructions principales de la classe **fenetreMenu**, on a donc une musique de fond présente dès l'ouverture de fenêtre. Celle-ci s'arrête lorsque l'on appuie sur le bouton "Jouer !" grâce à la méthode **actionPerformed**

Logiquement, la classe **classePrincipale** (classe MAIN) a pour unique fonction d'initialiser une **fenetreMenu** et donc de mettre en route le jeu!

IV. Structuration des données (hiérarchie des objets)

En se basant sur la partie précédente ("Principe de l'Algorithme") certains objets apparaissent comme étant le noyau même du programme et sont d'ailleurs caractérisés par de nombreuses lignes de codes par rapport aux objets secondaires ou périphériques. L'objet **fenetreJeu** ressort particulièrement du lot tout d'abord car il fait appel à la quasi-totalité des autres objets crées pour ce programme. Effectivement comme explicité antérieurement l'objet **fenetreJeu** appelle tous les objets à l'exception de **Musique** et **fenetreMenu**, et

utilise à de multiples reprises les méthodes propres aux objets appelés comme montré dans les quelques exemples ci-dessous:

```
//gestion du joueur
boule.bouge(0);
if (boule.y + boule.cote >= 800-sol.hauteur)
    boule.estMort = true;

//dessin des obstacles
for (Obstacle obs : listeObstacle)
{
    obs.dessine(g);
}
```

```
//collision
if (listeObstacle.get(0).collision(boule))
    boule.estMort = true;
```

On remarque par ailleurs que c'est dans cet objet que sont gérées les interactions entre les différents objets secondaires telles que les collisions entre la boule et les obstacles ou encore la boule et les objets bonus/malus. En fin de programme sont aussi introduites les lignes de code permettant l'affichage graphique du jeu ainsi que les effets sonores.

Il va donc de soi que les objets secondaires sont ceux apparaissant dans la classe principale, et dont les méthodes sont presque exclusivement liées à l'objet créé dans la classe secondaire (à part des petites exceptions telle que la méthode **collision** dans la classe objet faisant appel à un autre objet **Boule**). On peut cependant réaliser la dichotomie entre les objets nécessaires au jeu lui-même - c'est-à-dire la partie où l'utilisateur joue réellement et interagit avec la boule) - et les objets crées dans les classes **fenetreMenu** et **fenetreGameOver** participant à l'amélioration de l'expérience ludique du joueur mais ne constituant pas en elles des parties "jouables". On retrouve donc les objets **Boule**, **Objet**, **Sol** et **Obstacle** parmi ces objets inhérents au jeu, et plus en retrait les classes **fenetreGameOver**, **fenetreMenu**, **Musique, Score** et **SomeRunnable** gérant les fenêtres intermédiaires au jeu ainsi que la musique, qui participent à l'enrichissement du programme même si elles ne sont pas nécessaires au fonctionnement du jeu.

V. Sources utilisées

Nous avons utilisé pour réaliser ce projet les cours à notre disposition sur Moodle, ainsi que le site web StackOverflow pour des choses que nous n'avions pas vu en cours comme le son par exemple. En P2i1, nous avons appris à écrire dans des fichiers textes, ce qui nous a aidé dans notre programme.

VI. Améliorations possibles et bugs connus

Quelques améliorations auraient pu être ajoutées si le temps nous le permettait. Nous aurions pu ajouter plus de bonus et de malus pour rendre le jeu plus amusant. Nous aurions aussi pu améliorer quelques bugs graphiques. Par exemple, certaines fois, le cochon que le joueur contrôle et les tuyaux clignotent, sans doute car la méthode paint est trop importante et que le temps entre chaque réveil du timer est trop faible. Certains objets ne disparaissent apas une fois entrés en collision avec le joueur. De plus au début d'une partie, on observe que le timer met du temps avant de démarrer et que le cochon ne réagit pas très bien. Le constructeur de la **fenetreJeu** est peut-être elle aussi surchargé et met du temps avant d'effectuer toutes ses actions.

Nous aurions pu aussi trouver un meilleur conditionnement du score, en effet après plusieurs tests nous avons fini par rajouter un +((int)(vt*30*0.001)+1) au code suivant :

```
if (listeObstacle.get(0).x +listeObstacle.get(0).largeur < boule.x && listeObstacle.get(0).x
+listeObstacle.get(0).largeur +((int)(vt*30*0.001)+1) >  boule.x)
        {
           score++;
           jouerSon("Score.wav");
        }//lignes 251 à 255
```

On vérifie donc si le cochon est passé après un tuyau. Les tuyaux se déplacent selon x de -vt*30*0.001 à chaque ActionPerformed. Le cochon ne doit passer donc qu'une unique fois dans un intervalle [0,vt*30*0.001] après un tuyau. Pourtant comme nous ne travaillons qu'avec des entiers, nous avons dû rajouter +1 pour être sûr que le cochon passe dans l'intervalle. Quand il passe dans cet intervalle on augmente le score de 1. Cependant, comme cette façon de faire est assez approximative, il existe sûrement un moyen plus efficace de gérer le score.

VII. Carnet de route

Nous avons fait attention à ne pas avoir des ambitions trop grandes en début de projet, sachant qu'il vaut mieux commencer par une base solide et pas trop longue à mettre en place tout en sachant que nous pourrions toujours rajouter des éléments ultérieurement. Nous étions partis à la base sur la création d'un Snake amélioré, mais dont la partie scientifique était trop faible. Nous avons ensuite eu du mal à retrouver un nouveau sujet d'étude, jusqu'à arriver à l'idée d'une variante de Flappy Bird. La première séance ne fut donc pas aussi efficace que nous l'aurions souhaité, car nous avons dû penser la hiérarchie de nos classes.

Nous avons tout d'abord consacré les deux premières séances à mettre en place les mécaniques fondamentales du programme, c'est à dire le mouvement du cochon et la gestion des obstacles en créant la fenêtre du jeu, de la boule et des obstacles.

La semaine d'après, nous avons réfléchi à la gestion des collisions, tout en travaillant à corriger les premiers bugs. Puis nous avons réfléchi sur des valeurs que nous pourrions utiliser pour augmenter la difficulté.

Une fois satisfait de cette base, nous nous sommes concentrés la quatrième séance sur des éléments qui donnerait de l'intérêt au jeu comme la gestion du score ainsi que celle des objets (bonus et malus). En parallèle, nous avons mis en place une fenêtre de menu qui se lance au lancement du programme ainsi qu'une fenêtre de game over qui clôture la partie. La plus grande difficulté pour ces fenêtres fut la gestion des images en arrière plan, qui devaient avoir de bonnes dimensions pour ne pas perturber les boutons de la fenêtre.

Après cette séance, nous avons travailler chez nous afin de réussir à écrire dans un fichier à partir de la fenêtre GameOver.

Nous avons utilisé la dernière séance en classe afin d'optimiser le mieux possible la gestion des malus/bonus. Nous avons également vérifier que les parties codées par des personnes différentes du groupe fonctionnaient bien dans le programme.

Enfin la dernière semaine, afin de rendre un programme fonctionnel et ludique, nous avons corrigé les derniers bugs du jeu, et nous avons rajouté de la musique et des nuages.