Rapport: Projet Tutoriel de PCII

I- Introduction

L'objectif du projet tutoriel de PCII est de réaliser un mini-jeu s'inspirant d'un célèbre jeu mobile d'obstacles, *Flappy Bird* sorti en 2013. Le but du jeu est de faire sauter un oiseau qui chute constamment pour lui permettre d'esquiver des tuyaux. Pour notre projet nous utiliserons un ovale à la place de l'oiseau et le parcours à suivre ne sera pas délimité par des tuyaux mais par une ligne brisée sur laquelle l'ovale devra se déplacer sans en sortir. Ses sauts se feront par l'intermédiaire de clics du joueur sur l'interface graphique. Ce mini-jeu sera réalisé en java. Voici une première comparaison entre le jeu initial et l'interface graphique actuelle de notre jeu :

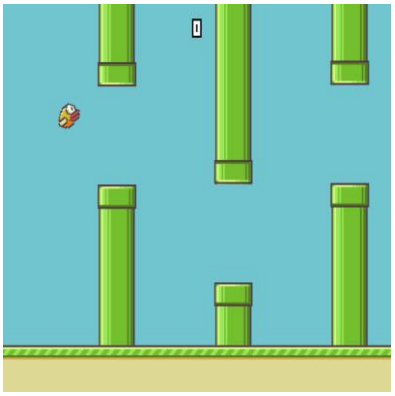

Figure 1 : Flappy Bird

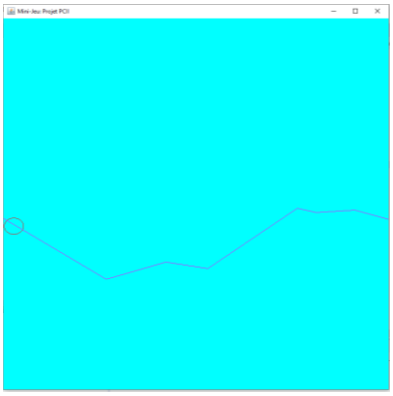

Figure 2 : Interface graphique actuelle

II- Analyse globale

Lors de la première séance nous avons travaillé sur la création d'une fenêtre représentant l'interface graphique de notre jeu et sur les déplacements verticaux de l'ovale quand le joueur clique sur celle-ci.

Au cours de la deuxième séance notre attention s'est portée sur la chute progressive de l'ovale, la mise au point d'un algorithme de génération d'une ligne brisée qui sera détaillé dans la section V et à l'avancée de cette ligne dans la fenêtre. Pour les premiers et troisièmes points de cette séance nous avons utilisé des threads. L'idée générale est de produire l'illusion que l'ovale se déplace, pour cela nous allons simplement faire se déplacer la ligne brisée de la droite vers la gauche.

Enfin pour la troisième et dernière séance nous avons travaillé sur le dernier point du jeu, la détection des collisions entre l'ovale et la ligne. Cependant nous avons dû modifier notre algorithme de génération de ligne brisée car il compliquait le processus de détection de collision. Le nouvel algorithme est également plus simple et plus intuitif.

III- Plan de développement

Voici la liste des différentes tâches que nous avons réalisées lors des séances de TP :

Séance 1 :

- Lecture et analyse du sujet *(25 mn)*
- Conception, développement et test d'une fenêtre avec un ovale *(40 mn)*

- Conception, développement et test des fonctionnalités de déplacement de l'ovale *(105 mn)*
- Documentation du code *(40 mn)*

Séance 2 :

- Lecture et analyse du sujet *(10 mn)*
- Conception, développement et test de la chute progressive de l'ovale et documentation du code *(30 mn)*
- Conception, développement et test de l'algorithme de génération de la ligne brisée et documentation du code *(125 mn)*
- Animation de la ligne brisée et documentation du code *(45 mn)*

Séance 3 :

- Lecture et analyse du sujet (*5 mn*)
- Modification de l'algorithme de génération de la ligne brisée et documentation du code (*60 mn*)
- Conception, développement et test de la détection des collisions et documentation du code (*145 mn*)

Voici les diagrammes de Gant associés :

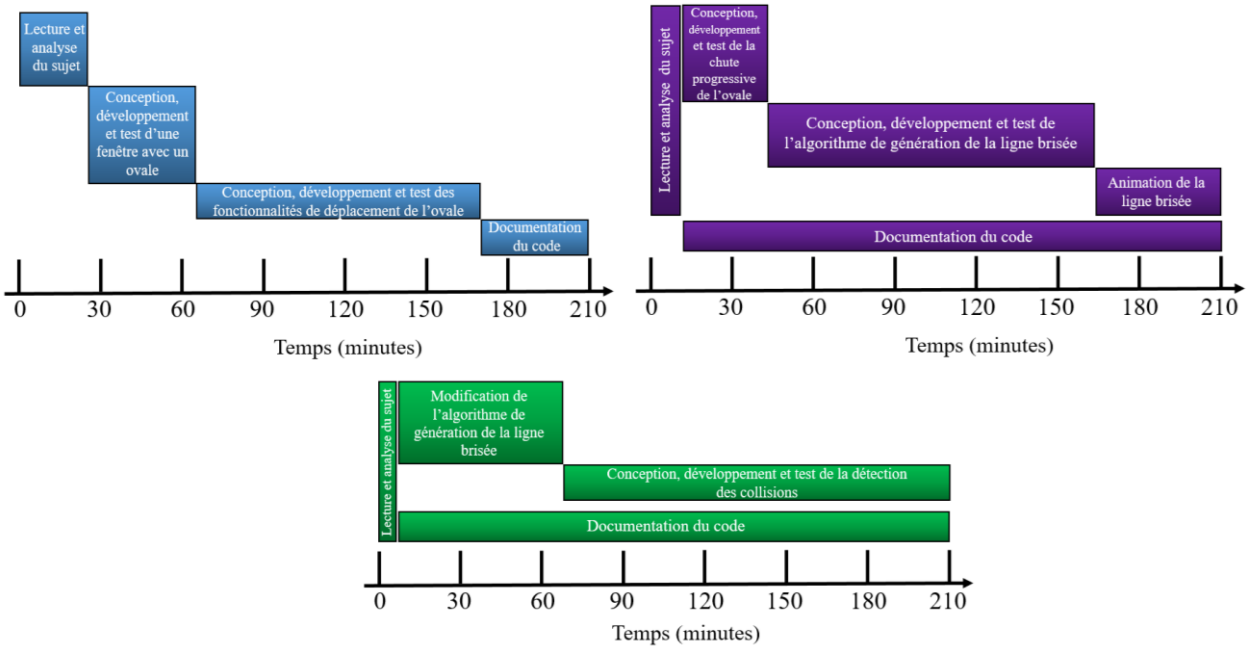

Figure 3 : Diagrammes de Gant des séances 1, 2 et 3

IV- Conception générale

Nous utilisons le motif Modèle Vue Contrôleur (MVC) pour le développement de l'interface graphique pour structurer notre code. Comme précisé précédemment les deux fonctionnalités que nous avons étudiées jusqu'ici sont la création d'une fenêtre pour l'interface graphique et les déplacements de l'ovale. Le schéma suivant montre comment ces fonctionnalités rentrent dans le motif MVC.

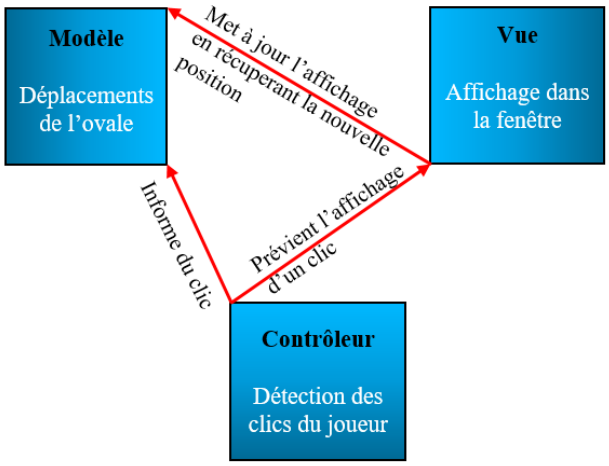

Figure 4 : Représentation du motif MVC dans le cadre de notre projet

V- Conception détaillée

V.1- Mise au point de la fenêtre

Pour la fenêtre avec un ovale, nous utilisons l'API *Swing* et la classe *JPanel*. Les dimensions de l'ovale et de la fenêtre sont définies dans les constantes suivantes :

- *LARGEUR_FENETRE* : largeur de la fenêtre
- *HAUTEUR_FENTRE* : hauteur de la fenêtre
- X : abscisse de l'ovale
- *y* : ordonnée de l'ovale
- *LARGEUR_OVALE* : largeur de l'ovale
- *HAUTEUR_OVALE* : hauteur de l'ovale

V.2- Déplacement de l'ovale

Pour le déplacement de l'ovale, nous utilisons la programmation événementielle avec la classe *MouseListener* et la hauteur est définie dans une constante. Le diagramme de la classe *Control* est présenté ci-dessous.

V.3- Chute de l'ovale et avancée de la ligne

Concernant la gestion de la chute de l'ovale et l'avancée de la ligne brisée nous utilisons la programmation concurrente à l'aide de threads, l'objectif est que l'ovale perde de l'altitude lorsque le joueur ne clic pas sur la fenêtre. Pour cela nous avons implémenté la classe *Voler* qui hérite de *Thread*, quelques modifications ont également été apportées à la classe *Etat* dans laquelle nous avons ajouté la méthode *moveDown* qui est appelée toutes les 100 millisecondes pour modifier l'ordonnée de l'ovale d'une constante nommée *SAUT_DOWN*. Quant à la ligne brisée, elle est avancée toutes les 80 millisecondes.

V.4- Conception de la ligne brisée

Il a également fallu mettre au point un algorithme permettant de générer une ligne brisée. Elle est représentée par une ArrayList d'élements de type Point. Cette partie a été la plus longue de la deuxième séance car nous avons dû modifier plusieurs fois notre méthode pour le faire fonctionner. Mais comme précisé précédemment nous avons changé d'algorithme par la suite. Ces deux algorithmes sont décrits ci-dessous.

1ere version de l'algorithme (supprimée du code) :

Description du choix d'itinéraire :

- On peut soit monter, descendre ou stagner. Ce choix se fait en fonction de la dernière direction choisie, la prochaine direction doit être différente de la précédente. Pour choisir entre les 2 possibles, on tire aléatoirement un entier dans l'intervalle [0 ; 1] et on attribue une direction à fonction du résultat obtenue
- On tire aléatoirement un entier dans un intervalle [*tailleMin* ; *tailleMax*] représentant la longueur de la ligne sur l'axe des abscisses

Description de l'algorithme :

- Si on est au premier point de la ligne
  - On peut monter ou descendre. Pour cela on tire aléatoirement un entier dans l'intervalle [0 ; 1]
    - Si on obtient 0, on choisit de descendre
    - Sinon on choisit de monter
  - On tire un aléatoirement un entier dans un intervalle [*tailleMin* ; *tailleMax*] représentant la longueur de la ligne sur l'axe des abscisses
- Sinon
  - Si aucune ligne n'est en court de construction
    - On choisit un nouvel itinéraire
  - Boucle finie (fin fixée par un entier, sa valeur n'est pas importante):
    - Si la ligne en cours de traitement est finie
      - On choisit un nouvel itinéraire
    - Sinon, en fonction de l'itinéraire actuel on modifie l'ordonnée du point
    - On incrémente son abscisse et on ajoute le nouveau Point à la liste

Pseudo-code :

```
choix_Itineraire()
        choix ← random([|0 ;1|])
        si self.itineraire := 1 # 1 correspond à monter
                si choix := 1
                        self.itineraire ← 2
                sinon
                        self.itineraire ← 0
        sinon si self.itineraire := 0 # 0 correspond à descendre
                si choix := 0
                        self.itineraire ← 2
                sinon
                        self.itineraire ← 1
        sinon                   # correspond à stagner
                si choix := 0
                        self.itineraire ← 1
                sinon
                        self.itineraire ← 1
        self.taille_Ligne ← random([|TAILLE_MIN ; TAILLE_MAX|])
```

```
createLigne(bool debut)
        si debut
                self.itineraire ← random([|0 ; 1|])
                self.taille_Ligne ← random([|TAILLE_MIN ; TAILLE_MAX|])
        sinon si self.cpt := 0
                self.choix_Itineraire()
        Pour i allant de 0 à stop # la valeur de stop n'est pas très importante
                si self.cpt := self.taille_Ligne
                        self.choix_Itineraire()
                        self.cpt ← 0
                sinon
                        si self.itineraire := 0
                                si self.y + 1 > HAUTEUR_FENETRE
                                        self.cpt := self.taille_Ligne – 1
                                sinon self.y++
                        si self.itineraire := 1
                                si self.y - 1 < 0
                                        self.cpt := self.taille_Ligne – 1
                                sinon self.y--
                self.x++
                self.cpt++
                self.pointList.add(Point(self.x, self.y))
```

2ere version de l'algorithme (utilisée dans le code) :

Description de l'algorithme :

- Si la dernière ligne montait, alors la suivante descend et inversement
- On tire les coordonnées du prochain point dans des intervalles spécifiques pour l'abscisse et l'ordonnée définis par des constantes
- Si la pente du segment obtenu entre le point actuel et celui qui vient d'être calculé n'est pas dans un intervalle défini auparavant, on tire de nouvelles coordonnées pour ce point, on continue ce processus tant que la pente n'est pas valable

Pseudo-code :

```
createLigne() :
        x ← random([|self.x + X_MIN ; self.x + X_MAX|])
        Faire {
                si self.itineraire := 0
                        y ← random([|Y_MIN; self.y  |])
                sinon
                        y ← random([|self.y; Y_MAX |])
                pente ← (y – self.y)/(x – self.x)
        } Tant que
                pente > maxPente ou pente < minPente
        si self.itineraire := 0
                self.itineraire ← 1
        sinon self.itineraire ← 0
        self. pointList.add(Point(x, y))
        self.x ← x
        self.y ← y
```

La deuxième version de l'algorithme est beaucoup plus simple, plus intuitive et également plus simple et plus rapide à coder. Elle ne présente que des avantages par rapport à la précédente, ce qui explique notre choix.

V.5- Implémentation de la détection de collision

La dernière étape est l'ajout de la détection de collision pour savoir quand l'ovale sort de la ligne, ce qui doit stopper la partie. Pour être capable de détecter une collision il faut connaître l'ordonnée y(t) de la ligne au point d'abscisse correspondant à la position 0 de l'ovale. Pour cela nous avons utilisé la formule suivante :

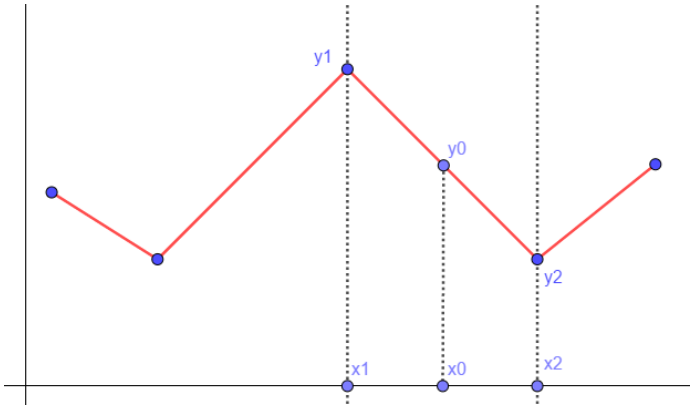

Figure 5 : Graphique illustrant les formules de p et y0

$$p = \frac{y_2 - y_1}{x_2 - x_1}$$

$$y_0 = y_1 - p * (x_1 - x_0)$$

Cette formule nous donne l'ordonnée du point visible de la ligne le plus à gauche dans la fenêtre, cependant pour détecter une collision avec l'ovale il ne faut pas comparer la position de l'ovale avec y_0 mais avec y_{shift} comme montré sur le schéma suivant.

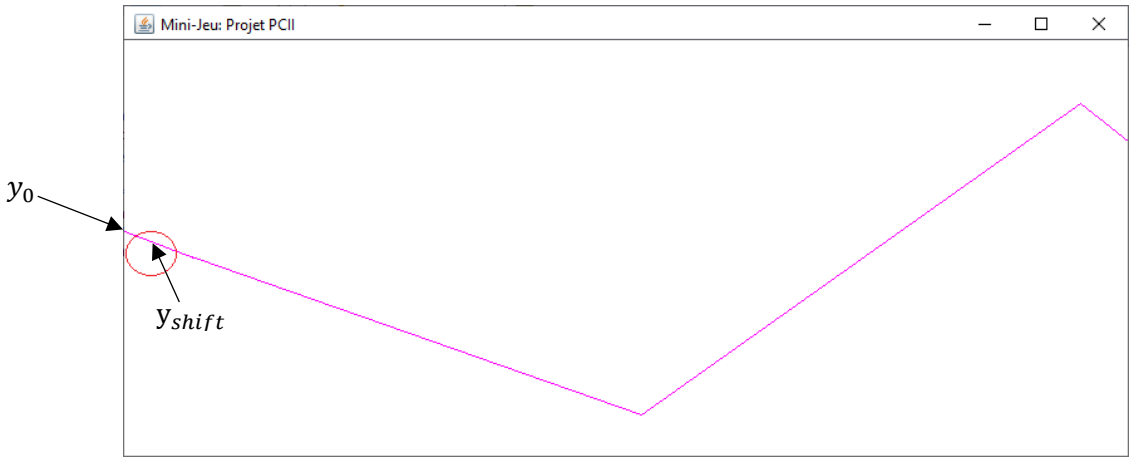

Figure 6 : Figure illustrant la différence entre y0 et y_shift

Voici la formule finale que nous utilisons pour obtenir l'ordonnée du point de la ligne où la collision pourrait s'effectuer (*X* et *LARGEUR_OVALE* correspondent à des constantes définies dans la classe *Affichage*) :

$$y_{shift} = y_0 + \left(X + \frac{LARGEUR\_OVALE}{2}\right) * p$$

Lorsque l'ovale sort de la ligne, les threads sont stoppés et un message indique la défaite à l'écran.

Voici les diagrammes de classes de toutes les classes que nous avons implémentées :

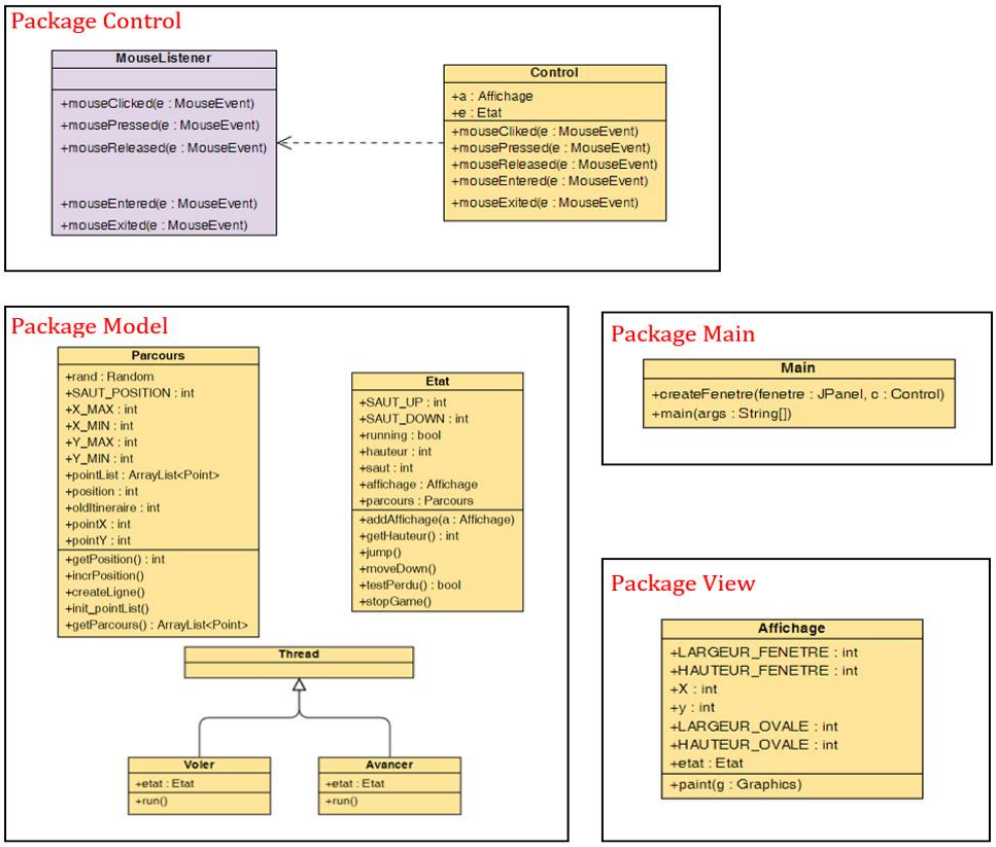

Figure 5 : Diagrammes de classes du projet

VI- Documentation utilisateur

Les éléments suivants sont la documentation utilisateur :

- Prérequis : Java avec un IDE (ou Java tout seul si vous avez fait un export en `.jar` exécutable)
- Mode d'emploi (cas IDE) : Importez le projet dans votre IDE, sélectionnez la classe Main à la racine du projet puis « Run as Java Application ». Cliquez sur la fenêtre pour faire monter l'ovale.
- Mode d'emploi (cas `.jar` exécutable) : double-cliquez sur l'icône du fichier `.jar`. Cliquez sur la fenêtre pour faire monter l'ovale.

VII- Documentation développeur

Le code est séparé en 7 classes, la principale étant *Main*. Les constantes importantes permettant de modifier le rendu sont pour la plupart dans la classe *Affichage*. Les

constantes permettant de modifier la structure de la ligne brisée sont dans la classe *Parcours*.

VII- Conclusion et perspectives

La partie sur la gestion des déplacements de l'ovale lors d'un clic du joueur a été la partie la plus difficile et longue de la première séance mais nous avons fini par comprendre comment cela fonctionne. Le fait de devoir changé notre algorithme de génération de la ligne brisée lors de la dernière séance nous a permis de revenir sur beaucoup de choses de la deuxième séance et de finalement rendre le code plus lisible et clair. Nous avons rencontré beaucoup de difficultés pour la partie sur la détection des collisions car nous n'utilisions pas la bonne formule au départ.

Nous avons appris beaucoup de choses au cours de ce projet tutoriel, notamment l'importance de la documentation qui permet de comprendre rapidement ce que fais le code lorsqu'on revient dessus, mais également comment utilisé des threads ou encore le motif MVC qui permet de structurer le code.