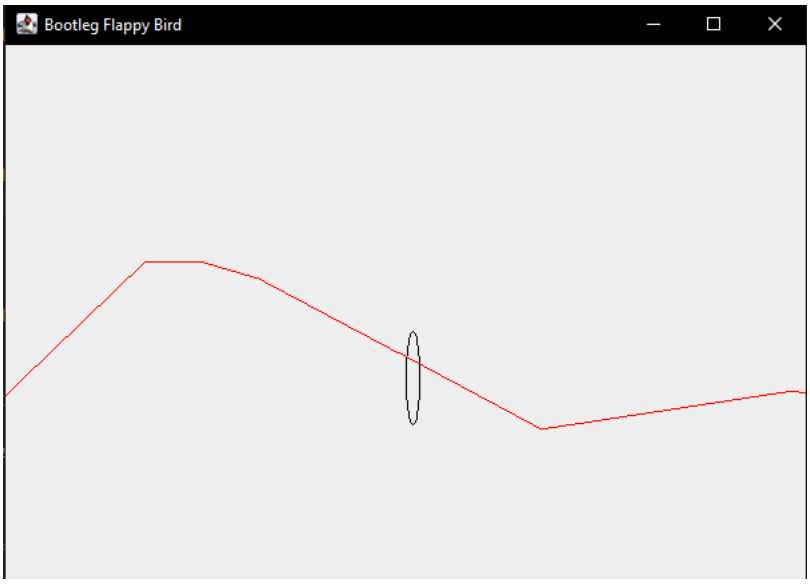
1 Introduction

L'objectif de ce tutoriel est de réaliser un mini-jeu qui reprend le principe de *Flappy Bird* de manière simplifiée. Le joueur sera un anneau autour d'un fil qu'il ne doit pas toucher. Pour ce faire, le joueur peut cliquer sur l'écran pour faire monter l'anneau un peu, et après celui-ci redescend tout seul.
Dans la première partie, nous avons dû réaliser uniquement les sauts de l'anneau lorsque le joueur clique avec la souris. Dans la seconde partie du projet, nous avons introduit le défilement de la ligne brisée ainsi que la génération de celle-ci. Voici à quoi ressemble l'interface graphique du jeu :



2 Analyse Globale

L'ensemble du projet peut être découpé en 3 fonctionnalités :

- L'interface graphique (ligne brisée et ovale)
- Défilement automatique de la ligne
- "Sauts" de l'ovale lorsque le joueur clique avec la souris

La première séance a été consacré à la réalisation de plusieurs sous-fonctionnalités :

- La création d'une fenêtre
- Affichage de l'ovale

Ainsi que de la fonctionnalité des sauts de l'ovale lorsque le joueur clique avec la souris.
La seconde séance, elle, a été consacrée à la réalisation de l'affichage de la ligne brisée, ainsi que le défilement automatique de celle-ci

3 Plan de développement

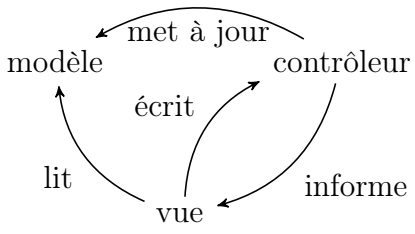
Liste des tâches :

- Analyse du problème (20 minutes)
- Conception, développement et test d'une fenêtre avec un ovale (20 minutes)
- Conception, développement et test d'un mécanisme de déplacement de l'ovale (40 minutes)
- Conception, développement et test de la création et l'affichage de la ligne brisée (60 minutes)
- Conception, développement et test du défilement automatique de la ligne brisée (20 minutes)
- Documentation du projet (150 minutes)

//DIAGRAMM DE GANTT

4 Conception générale

Ce projet est créé avec le motif MVC.
Ainsi, nous avons la classe *etat* pour le modèle, la classe *controle* pour le contrôleur et la classe *affichage* pour la vue.



5 Conception détaillée

La fenêtre est créée à l'aide de l'API swing et la classe JPanel. Les dimensions de la fenêtre ainsi que les dimensions de l'ovale sont données par des constantes :

- *width* : qui définit la largeur de la fenêtre
- *height* : qui définit la hauteur de la fenêtre
- *ovalWidth* : qui définit la largeur de l'ovale
- *ovalHeight* : qui définit la hauteur de l'ovale
- x : qui définit la position de départ de l'ovale sur l'axe des abscisses
- y : qui définit la position de départ de l'ovale sur l'axe des ordonnées

Le déplacement de l'ovale est lui géré avec de la programmation évènementielle et la classe *MouseListener*. Le saut, lui, est défini par une constante. Lorsque le joueur clique sur la fenêtre, le contrôleur augmente la position y de la constante de saut, et appelle la fonction repaint pour que l'affichage se mette à jour avec la nouvelle hauteur.

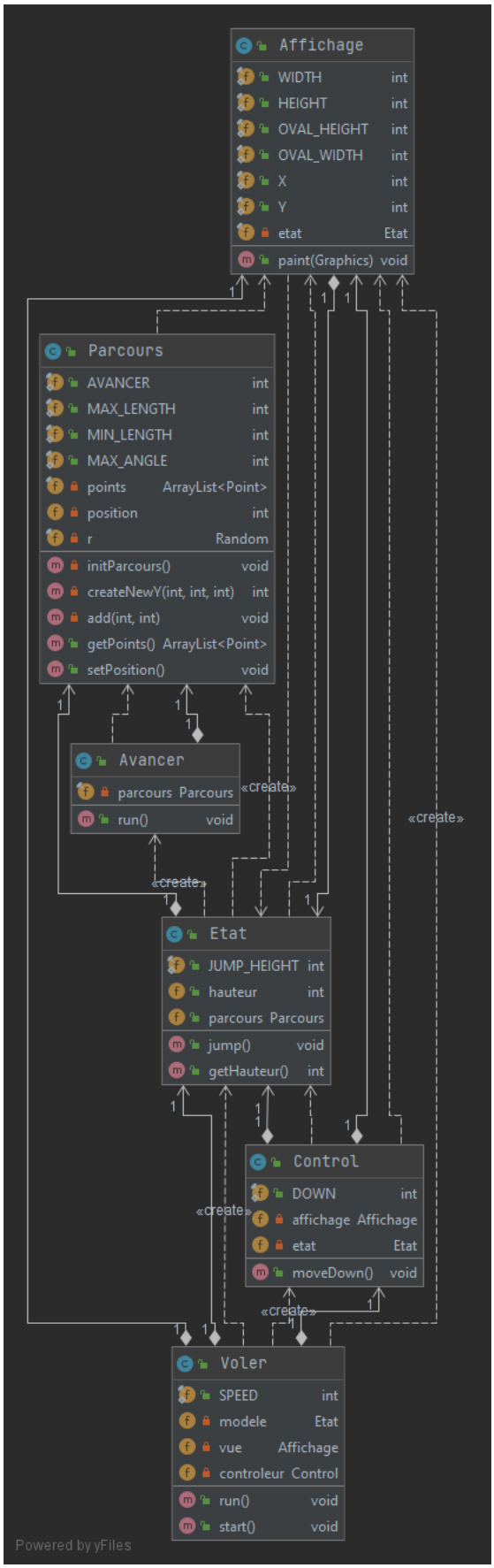
La ligne brisée est créée à l'aide d'une suite de points via la génération suivante :

```
fonction : initParcours () : void
    x <- largeur_fenetre + largeur_ovale/2
    y <- hauteur_fenetre + hauteur_ovale/2
    Point p <- new point(x, y)
    parcours.add(p)

    tant que(x <= largeur_fenetre)
        x = random entre MAX_LENGTH + p.x et MIN_LENGTH + p.x
        y = createNewY(x, p.x, p.y)
        p = new point(x, y)
    fin tant que
fin fonction

fonction : createNewY(x, oldX, oldY : x) : int
    y <- 0
    faire
        hyponetnus = valeur absolue de x - oldX
        sinus = sinus(random entre 0 et MAX_ANGLE)
        hauteur = hypotenus * sinus
        si random entre 0 et 1 == 0
        alors hauteur = hauteur * -1
        fin si
        y = valeur absolue de hauteur + oldY
    tant que y < hauteur_ovale OU y > hauteur_fenetre - hauteur_ovale
    renvoyer y
fin fonction
```

Le calcule de la coordonnée y se fait en fonction de la nouvelle coordonnée x, afin d'éviter d'avoir des angles trop optus et que le jeu soit injouable. Pour ce faire, les règles de trigonométrie sont utilisées : $sin(angle) = \frac{oppos}{hypotenus}$. On nombre aléatoire est pris entre 0 et l'angle maximal possible, défini dans la classe *Parcours* et la hauteur est ainsi calculée. Pour savoir si on va vers le haut ou vers le bas, un entier aléatoire est pris entre 0 et 1 et on multiplie par -1 si c'est 0 sinon rien.
Quand il n'y a plus assez de point, un nouveau est généré aléatoirement. De plus, quand il y a 2 points hors de l'écran (à gauche) le dernier point est supprimé.

Pour créer l'effet de défilement, on soustrait une constante à l'ensemble des coordonnées x des points du parcours à chaque tick du thread *Avance*.

A ce stade du projet, nous avons le diagramme de classe suivant :


Affichage
WIDTH int
HEIGHT int
OVAL_HEIGHT int
OVAL_WIDTH int
X int
Y int
etat Etat
paint(Graphics) void
Parcours
AVANCER int
MAX_LENGTH int
MIN_LENGTH int
MAX_ANGLE int
points ArrayList<Point>
position int
r Random
initParcours() void
createNewY(int, int, int) int
add(int, int) void
getPoints() ArrayList<Point>
setPosition() void
Avancer
parcours Parcours
run() void
«create»
Etat
JUMP_HEIGHT int
hauteur int
parcours Parcours
jump() void
getHauteur() int
Control
DOWN int
affichage Affichage
etat Etat
moveDown() void
Voler
SPEED int
modele Etat
vue Affichage
controleur Control
run() void
start() void
Powered by yFiles

6 Documentation utilisateur

Prérequis : Java avec un IDE.
Mode d'emploi : importer le projet dans votre IDE, sélectionner la classe Main puis "Run as Java Application". Cliquez sur la fenêtre pour faire monter l'ovale.

7 Documentation développeur

La prochaine fonctionnalité à implémenter est la "mort" du joueur : la détéction de collision entre l'anneau et la ligne brisée.

8 Conclusion et perspectives

La plus part des fonctionnalitées majeurs ont été implémentées, il ne reste qu'à créer la détéction de collision. Il serait possible de faire un ajout de l'affichage graphique afin de rajouter des informations pour le joueur (par exemple : le score et le temps) ou des éléments graphiques afin de faciliter l'immersion du joueur dans le jeu.