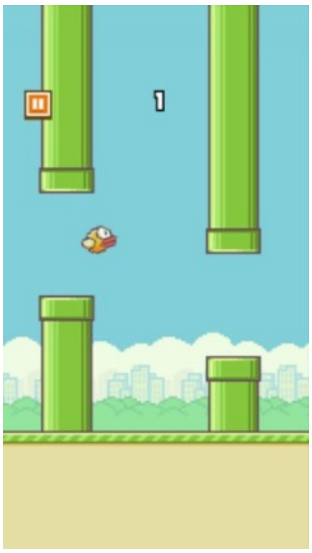
1.INTRODUCTION

L'objectif est de réaliser un mini-jeu inspiré de FlappyBird dans lequel se déplace le long d'une ligne brisée. Le but est d'éviter que l'ovale sort de la ligne un minimum de fois tout en essayant d'aller le plus loin possible dans la ligne brisée. Pour avancer l'utilisateur peut clicher sur l'écran pour faire monter l'ovale qui redescend ensuite tout seul.



2.ANALYSE GLOBALE

Les principales fonctionnalités à developper ont été:

- Création de la fenêtre où se dessine l'oval: fonctionnalité a été assez facile a implémenter grâce a la documentation de java swing.

- Déplacement de l'ovale vers le haut quand on clique la fenêtre: fonctionnalité un peu plus difficile a implémenter et essentielle pour l'interaction avec l'utilisateur.

- Organisation du projet sous la forme du modèle MVC: essentiel pour maintenir le projet propre et organisé.

- Utilisation des threads pour rafraîchir le modele (les données) de l'oval: il se déplace ver le bas quand l'utilisateur touche pas la fenêtre.

- La création et affichage d'une ligne brisée et son déplacement automatique pour donner l'impression que l'ovale avance le long de cette ligne:
    - Création de la ligne brisée.
    - Affichage de la ligne brisée.
    - Animation de la ligne brisée.

- Implémentation d'un mécanisme de détection de collisions.

- Définition de la condition de fin de jeu et arrêt de threads pour afficher l'écran de fin de jeu.

- Ajout des oiseaux qui apparaissent aléatoirement comme décors: ils augmentent l'impression de defilement:
    -Définition de l'objet oiseau.
    -Affichage de l'oiseaux.
    -Génération de plusieurs oiseaux.

- Synchronisation des threads: une seule thread mets à jour l'affichage, toutes les autres mettent a jour un modèle.

3. PLAN DE DÉVELOPPEMENT

Liste de taches:

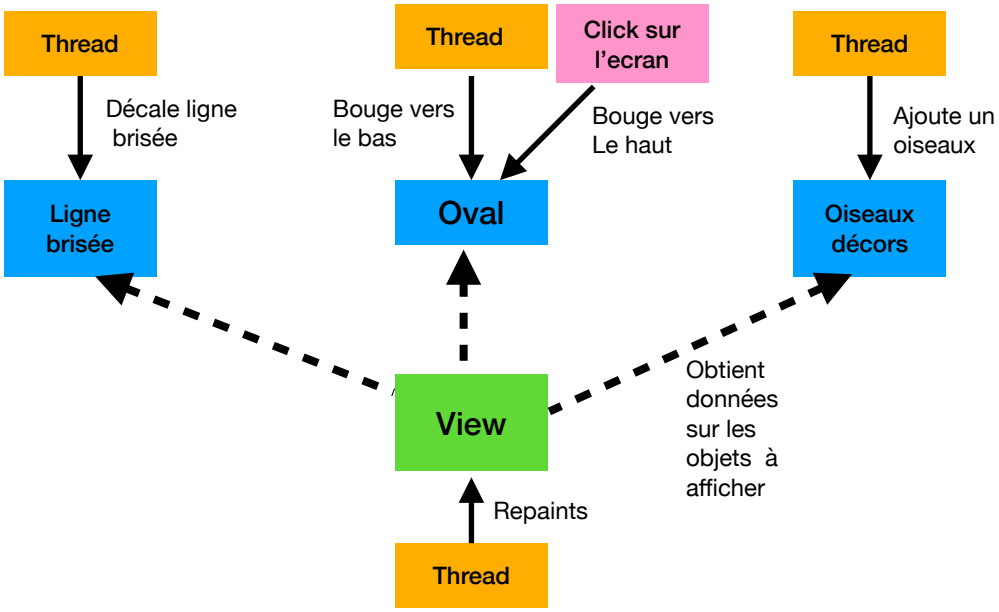
a.Analyse du problème (15 mn)
b.Conception, développement et test d'un fenêtre avec un ovale (30 mn)
c.Conception, développement et test du mécanisme de déplacement de l'ovale (45 mn)
d.Acquisition de compétences en Swing (60 mn)
e.Documentation du projet (60 mn)

f.Conception, développement et test du thread pour déplacer l'oval vers le bas (30 mn)
g.Conception, développement et test de la construction de la ligne brisée (30 mn)
h.Conception, développement et test de l'animation de la ligne brisée (75 mn)
i.Documentation du projet (60 mn)
j.Analyse du problème algorithmique pour détecter les collisions (30 mn)
k.Conception, développement et test du mécanisme de collisions (75mn)
l.Conception, développement et test des différentes nouvelles classes qui permettent la génération des oiseaux du décor(60 min)
m. Conception et développement de la thread qui met a jour l'affichage pour permettre la synchronisation (30 min)
n. Déboggage de la thread qui permet la synchronisation (120 mn)
o. Documentation du projet (120mn)

NB: Pour que le diagramme de Gantt suivant soit plus lisible j'ai décidé de le diviser par heures, ainsi un bloc avec la couleur jaune est une tache qui a durée 15 min dans un bloc d'une heure, Orange 30 min, Rouge claire 45 min et Rouge foncé 1 Heure.

H/ Tache	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	15													
B	30													
C	15	30												
D		30	30											
E			30	30										
F				30										
G					30									
H					30	15								
I						45	15							
J							30							
K							15	60						
L									60					
M										30				
N										30	60	30		
O													60	60

4. CONCEPTION GENERALE



MAIN:
On Construit une fenêtre du type JFrame
On construit des objets pour l'oval, la ligne brisée et les oiseaux de decors.
On construit un objet view qui prends tous les objets modèles comme paramètre.
On affiche la view dans la fenêtre.

5. CONCEPTION DÉTAILLÉE

CRÉATION DE FENÊTRE OU SE DESSINE L'OVAL

Pour la fenêtre avec un ovale, nous utilisons l'API Swing et la classe `JPanel`. Nous définissons les dimensions de l'oval et de la fenêtre dans des constantes visibles dans le diagramme de classes ci-dessous.
Ensuite on écrit une méthode dans la classe view qui prends en paramètre un objet Graphic g et qui se charge de dessiner l'objet du modele: dans ce cas l'ovale. Cette méthode est ensuite appelée dans la méthode paint de view. Ce meme traitement sera appliqué a tout nouveau objet modèle introduit pendant le projet.

DÉPLACEMENT DE L'OVAL QUAND LA FENETRE EST CLIQUÉE

Pour le déplacement de l'ovale, nous utilisons la programmation événementielle avec la classe `MouseListener` : quand l'utilisateur clique sur l'écran l'oval est bougé vers le haut: on modifie l'ordonnée de l'ovale avec la méthode de sa classe (la classe Model)

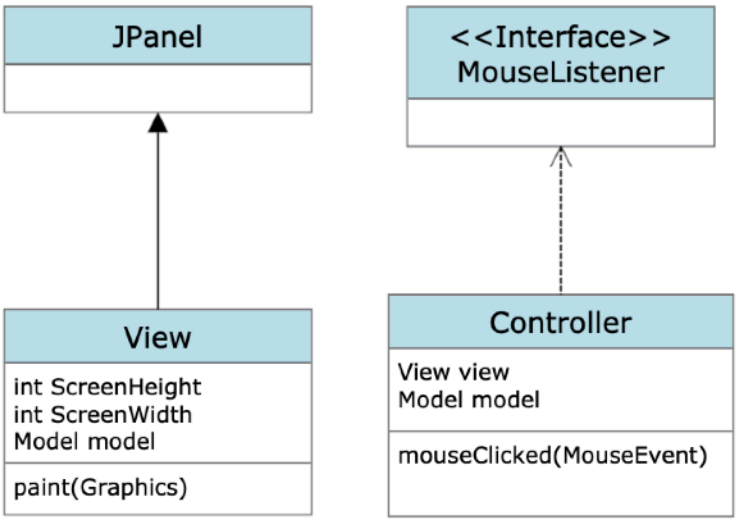
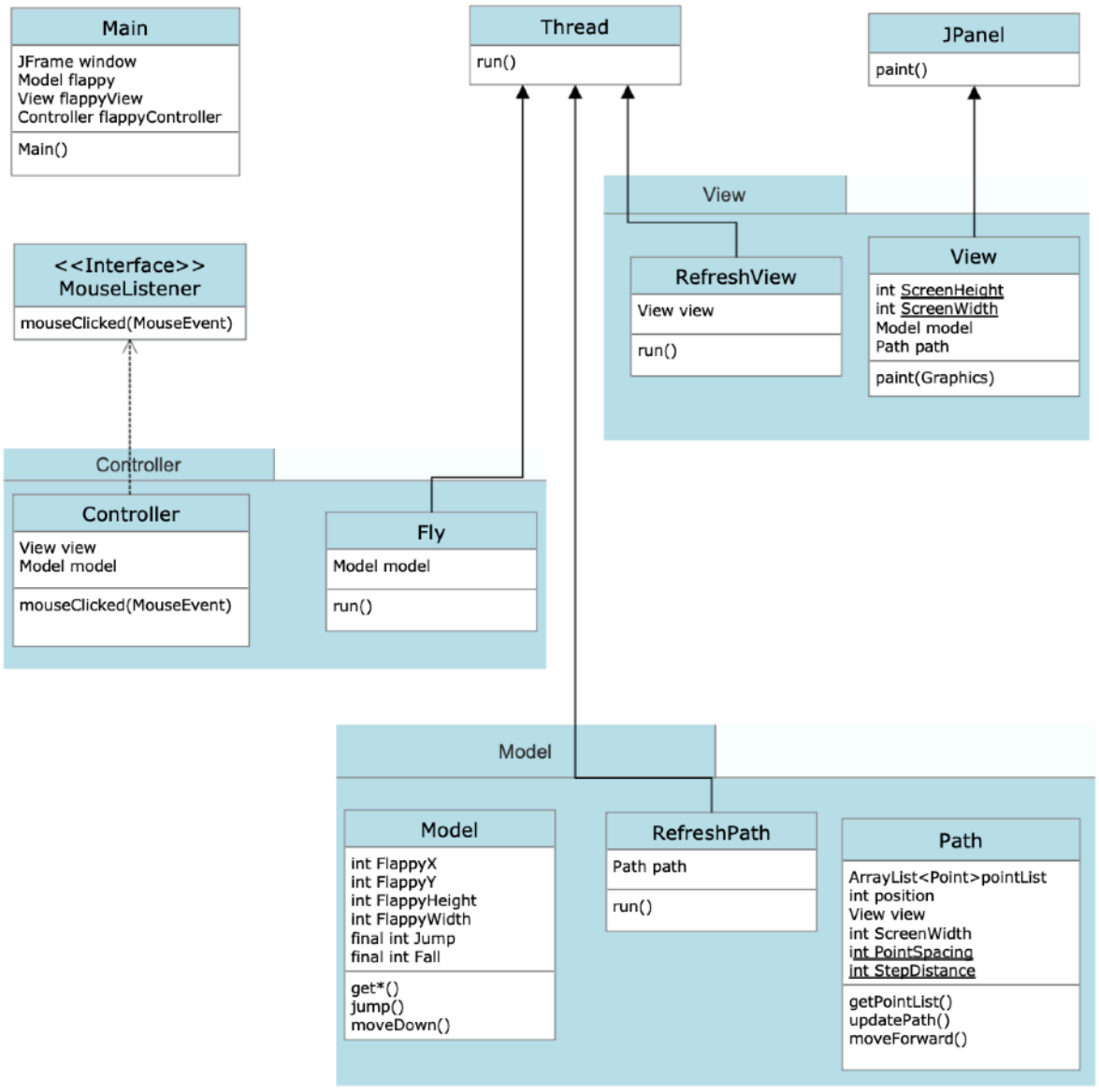


Diagramme des classes des fonctionnalités décrites en dessus.

ORGANISATION DU PROJET SOUS FORME MVC
Création de 3 packages: Model, View et Controller. Les classes de modele sont les modèles du projet: l'ovale, le path (ligne brisée) et les oiseaux du décors. Les classes de la view sont liés à l'affichage ou a son rafraîchissement. Les classes dans controller modifient les objets a afficher

IMPLEMENTATION DES THREADS QUI RAFRAÎCHISSENT INDÉPENDAMMENT LES ELEMENTS DU MODEL
Les classes héritent de la classe `Thread` puis on @Override la méthode run pour rafraîchir les données de nos objets du modele chaque x millisecondes sous une while(true) loop. On start les Threads à l'intérieur du constructeur de la classe qui décrit le modele que cette thread va modifier.

Voici le diagramme de classes qui montre clairement les deux dernières fonctionnalités décrites dessus:



CRÉATION, AFFICHAGE ET ANIMATION DE LA LIGNE BRISÉE

Description des algorithmes utilisés:

L'algorithme qui crée la ligne brisée est codé dans le constructeur de la classe Path:

L'algorithme se sert de la constante:
PointSpacing: c'est l'espace entre chaque point de la ligne brisée, on le définit comme la width de l'écran divisé par 10.

Initialisation:
x=PointSpacing.
y=nombre aléatoire dans l'écran (entre 0 et ViewHeight).

Tant que x reste dans l'écran faire:
Ajouter le point (x,y) à l'attribut pointList.
x+=PointSpacing
y= un autre nombre aléatoire dans l'écran.

Algorithme qui anime la ligne brisée (classe path) en modifiant sa liste de points pointList dans sa la méthode de path, moveForward():
Initialisation:
StepDistance: c'est la distance qu'on va décaler les points de la ligne brisée pour simuler que l'ovale se déplace vers la droite.

Pour tout point p de la ligne brisée (de l'ArrayList pointList):
p.x-=StepDistance

Cet algorithme simple mets a jour la liste de points quand il est appelé par le Thread qui mets a jour path (la ligne brisée)

Algorithme qui mets à jour la pointListe pour qu'elle ne contienne que les points visibles ou nécessaires (implémenté dans la méthode updatePath) et ajoute un nouveau point:

Initialisation:
On crée une copie de pointList

Si Flappy avance jusqu'à un point de la ligne brisée faire:
(On peut identifier cette situation si la position est un multiple de PointSpacing)

-Eliminer premier element de la copie
-Ajouter un point a la fin de la ligne brisée qui a pour coordonnées:
x : coordonnée x du dernier point de la copie plus PointSpacing.
y: un nombre aléatoire dans l'écran.

Mettre a jour pointList en la remplaçant par cette copie modifié.

Ces trois mécanismes et la thread qui rafraîchit le path (ligne brisée): RefreshPath permettent de faire marcher cette fonctionnalité.

Mécanisme qui assure un déplacement effectif de l'oval:
Pour éviter de créer une ligne brisée trop agressive (qui génère des points trop loin de l'ovale)

j'ai décidé de fixer les nombres aléatoires des ordonnées des points à des nombres entre 0.4 de l'hauteur et 0.6 de l'hauteur de l'écran. Ceci assure que à la vitesse de rafraîchissement de l'oval (Thread fly) celui ci arrive a suivre la ligne brisée sans problèmes.

MÉCANISME DE DETECTION DE COLLISIONS

Description de l'algorithme qui détecte les collisions:
À chaque appel:

Initialisation:

pointAhead (P2): premier point dans la liste des points de la ligne brisée (liste ordonnée en abscisses croissantes) à avoir une abscisse supérieure a l'abscisse de l'ovale.
pointBehind(P1): point juste avant de pointAhead dans la même liste.
Ovale= point au centre de l'ovale

Verification que la droite entre pointAhead et pointBehind se trouve entre les limites dimensionnels de l'ovale:
Calcul de la pente:
Pente= P2.y -P1.y / P2.x-P1.x

Calcul de l'ordonnée du point d'intersection entre la droite et le centre de l'ovale:
y= P1.y+ pente* (ovale.x- P1.x)

Condition de non collision:
y>(oval.y-hauteurOval/2) et y<(oval.y+hauteurOval/2)

Cet algorithme va être appelé par la thread qui mets a jour le path (ligne brisée) et donc chaque fois que on la décale on va tester pour voir s'il y a eu des collisions. Si c'est le cas on va augmenter le compteur des collisions dans la classe path.

CONDITION DE FIN DE JEU ET AFFICHAGE DU DEBUT DU JEU

Après la fin du codage du mécanisme de détection j'ai pensé a changer le but du jeu: au lieu de arrêter toutes les thread et finir le jeu au but d'une collision, j'ai décidé de fixer un nombre de collisions pour terminer le jeu (25). Ainsi au but de 25 collisions le jeu va s'arrêter (la ligne va arrêter de défiler mais flappy peut encore répondre aux clicks pour avoir une sensation que le jeu n'est pas frozen) et un message de fin du jeu va s'afficher.

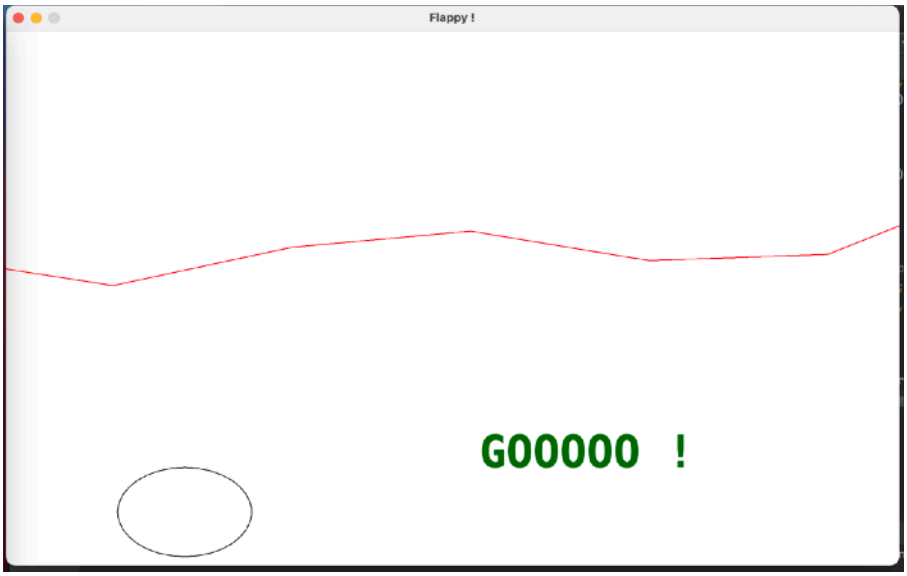
L'affichage du texte du début de game et les marqueurs(score et collisions) ont été implémentés dans les méthodes de la classe View: drawMarkers et drawGameStart respectivement.

Pour coder ceci j'ai utiliser un attribut de la classe path (ligne brisée) que j'appel *score*: il compte le nombre de points de la ligne brisée dépassés par flappy.

Le jeu ne commence que si score =0.
Score est initialisé à -5: flappy dépasse 5 points de la ligne brisée et en suite la game commence (on commence a tester s'il y a eu une collision) et les marqueurs s'affichent. Le but est de donner du temps à l'utilisateur pour s'habituer au jeu.

L'affichage du texte au debut du jeu est lié au score:
En effet dans la classe view j'ai crée une méthode pour dessiner le texte qui prends en paramètre le score et dessine un certain string a une certaine couleur selon le score:

Si le score vaut -1 ou 0 (juste avant et au moment que le jeu commence)il affiche "GOOO" en vert:
Par contre si c'est pas le "score correspondant " le string s'affiche quand meme mais en couleur blanche pour qu'il soit invisible à l'utilisateur.
Ceci a était fait pour éviter un bug qui cause que mon affichage freeze soudainement quand je affiche le texte après la creation de View.



OISEAUX DU DÉCORS

Objet oiseaux
Chaque oiseaux est un objet de type Bird et cette classe hérite de `Thread` . Chaque Oiseaux a son abscisse initialisé à ScreenWidth pour qu'il semble apparaitre au delà de l'écran. Son ordonnée est décrite par le suivant algorithme:

Initialisation:
intervalle1 =un nombre aléatoire entre 0.6 de l'hauteur de l'écran et l'hauteur de l'écran moins l'hauteur de l'oiseaux.
intervalle2= Un nombre aléatoire entre 0 et 0.4 de l'hauteur de l'écran moins l'hauteur de l'oiseau

Execution:
Rentrer intervalle1 et intervalle2 dans un tableau
Choisir aléatoirement entre un des deux
//fin de l'algorithme

Le résultat correspondra a la nouvelle ordonnée générée aléatoirement entre 2 intervalles qui se situent en dessous et en dessus de la ligne brisée et de l'ovale (on s'assure que les oiseaux de décors ne se dessinent sur la ligne brisée ou sur l'ovale bloquant la vision de l'utilisateur)

Un attribut délai va définir le temps de sleep du thread (initialisée entre 600 et 400 dans le constructeur car personnellement cette vitesse me semble satisfaisante).

Le thread est initialisé dans son propre constructeur.

Finalement dans son propre constructeur on a aussi l'état de l'oiseaux qui est un entier entre 0 et 7: ceci correspond a ces 8 états possibles parmi lesquels itère l'oiseau (à chaque état correspond une image dans une série d'images qui simulent l'animation de l'oiseaux quand les images sont affichées consécutivement).

Affichage des oiseaux

Pour générer et afficher des oiseaux on a besoin de les stocker quelque part: une classe BirdView avec un attribut ArrayList<Bird> birds nous permet de faire ça.

Mini-algorithme de mise à jour de la liste d'oiseaux:

On utilise un `ListIterator` pour itérer sur la liste sur laquelle on enlève des elements.
Si un des oiseaux de la liste birds n'est pas visible (est en dehors de l'écran) :
Remove l'element de birds.

Algorithme d'affichage des oiseaux:

Si la liste birds est non vide faire:

Initialisation: Mettre a jour la liste birds.

Pour chaque bird dans la liste birds faire:
    obtenir l'image lié a son état actuel.
    scale son image a une taille approprié (1/3 de l'image originale)
    afficher l'image à l'hauteur et abscisse de l'oiseau.

Cet algorithme est implémenté dans la méthode drawBirds de BirdView et va être appelé par la méthode paint de la classe View.

Generation de nouveaux oiseaux

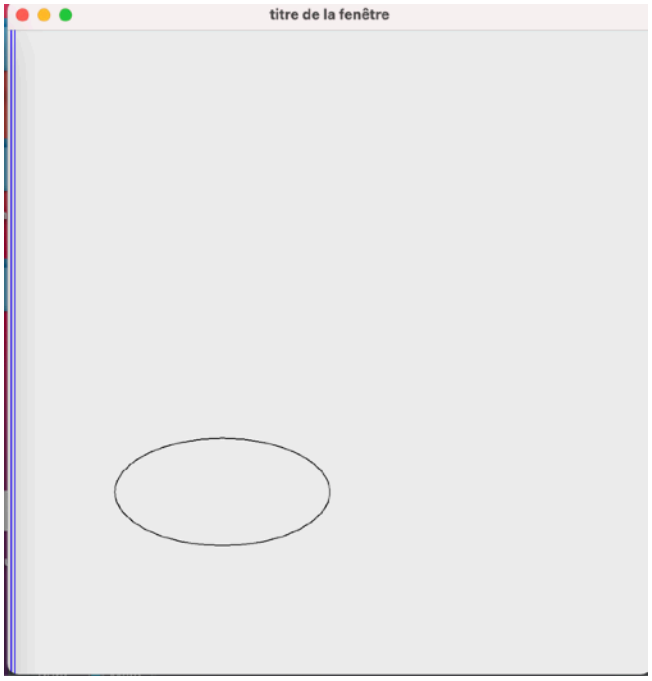
Pour générer des nouveaux oiseaux on a besoin d'une thread qu'on va appeler BirdGenerator elle hérite de `Thread` et va appeler la méthode generateBird de BirdView chaque 5 secondes. La méthode generateBird va ajouter un nouveau oiseau à la liste birds avec une probabilité de 0.7. Ceci est fait avec `Java.util.random`.

SYNCHRONISATION DES THREADS

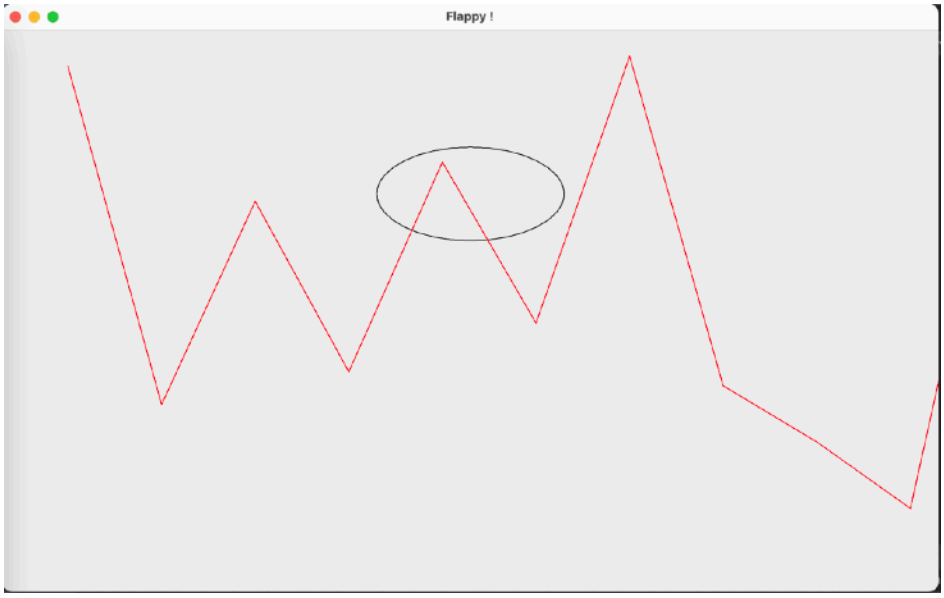
Cette fonctionnalité consiste a créer une thread qui mets a jour l'affichage, ceci est fait grâce a `Java.swing.Timer`. Si je fait cette thread avec une classe qui hérite de `Thread` lors de l'ajout un oiseau et de son affichage la View se congèle pendant quelques secondes. Ceci est apparemment lié au sleep du thread qui ajoute les oiseaux.

6.RÉSULTAT

VOICI LE PROGRÈS TOUT AU LONG DU PROJET:
SÉANCE 1:



SÉANCE 2:



SÉANCE 3 ET FIN DU PROJET :

7. DOCUMENTATION UTILISATEUR
-Prérequis : Java avec un IntelliJ IDEA.
-Mode d'emploi: Importez le projet dans votre IDE, sélectionnez la classe Main à la racine du projet puis « Run as Java Application ».
-Instructions du jeu: Cliquez sur la fenêtre pour faire monter l'ovale. Vous perdez quand vous dépassez les 25 collisions. Essayez d'arriver le plus loin possible !

8.DOCUMENTATION DÉVELOPPEUR

Méthode Main:
Elle se trouve dans la classe main en dehors des packages.

Classes importantes
Model: C'est ou il y a la description de l'ovale qui a pour principales constantes ScreenWidth et ScreenHeight qui définissent la taille de l'écran et FlappyHeight, FlappyWidth qui définissent les dimensions de Flappy.

View: Elle va être ajouté a un JFrame pour être affiché à l'écran. Elle contienne 3 attributs principaux: model, path et BirdView (un attribut par objet à modéliser).

Path: Contient une liste de points (ceux de la ligne brisée) ArrayList<Point> pointList. Ces autres attributs sont la position(initialisée à 0), le score (nombre de points dépassés par l'Oval, initialisé à -5), le compteur de collisions et un objet de type Model.
Les constantes utiles sont: PointSpacing(l'espace entre les points de la ligne brisée), StepDistance(le nombre de pixels que tous les points visibles sont décalés) et maxNumberOfCollisions (nombre de collisions maximal avant que l'utilisateur perds le jeu).

BirdView: Cette classe contienne l'ArrayList<Bird>birds ou on sauvegarde les oiseaux qui s'affichent à l'écran. Les constantes/Variables importantes sont: scaleWidth et scaleHeight qui représentent la taille des images de chaque oiseau.
NB: La classe Bird est aussi importante mais elle est expliquée dans la section 5. Conception détaillé.

Fonctionnalités a implémenter

Quelques fonctionnalités sont encore améliorables: On peut gérer les collisions de façon que on ne compte pas plus d'un collision par intervalle de temps inférieur au temps de réaction de l'utilisateur: cela va éviter que une seule erreur de l'utilisateur cause plus d'une collision.
Additionnellement il faut gérer le lag qui se produit par fois et peut affecter négativement le gameplay.

9.CONCLUSION ET PERSPECTIVES

Au début du développement du projet, on a construit une interface interactive très élémentaire qui affiche un ovale et le déplace quand l'utilisateur click.
Le plus difficile était de inclure l'objet qui hérite de MouseListener pour qu'il attende des cliques lors de l'affichage de la fenêtre: ma solution temporelle était de créer un Objet contrôleur dans le constructeur de View. Le problème de cette solution est qu'elle ne permette pas l'indépendance entre les classes. J'ai ainsi créer un objet contrôleur dans le main et j'ai donné accès à la JFrame pour qu'il ajoute directement dans son constructeur le MouseListener. J'ai donc appris a séparer les éléments du modele MVC en jouant avec les paramètres des constructeurs des classes.

Pendant la deuxième séance j'ai appris a utiliser les threads en java et j'ai compris que leur importance est dans le fait qu'elle permettent le rafraîchissement en parallèle des différents éléments du modele(ici l'ovale de Model et la ligne brisée de la classe Path).

On a rencontré des difficultés quand j'avais plusieurs classes qui appelaient la méthode paint de View et n'étaient pas indépendantes de View, J'ai donc appris a séparer mes mises à jour dans des différentes threads qui à leur tour mettent à jour leur modèles.

Le projet à été progressivement complété avec l'implémentation de la ligne brisée, la détection de collisions et finalement l'ajout des oiseaux du décors.
J'ai commencer à apprendre sur les problèmes de concurrence notamment avec le java.swing.Timer vers la fin du projet, cet outil m'a permis de résoudre le problème le plus difficile que j'ai rencontré tout au long du projet. Le Timer a remplacé une Thread avec un sleep et m'a donc permis d'éviter que toute ma View se congèle quand j'ajoute un oiseau.