université
PARIS-SACLAY

Rapport de projet PCII

Réalisé par
Shiqing HUANG

Sous la direction de
Thi Thuong Huyen Nguyen

Année universitaire 2020-2021
Licence Informatique

SOMMAIRE

Introduction…….…….. 2

Analyse globale…….... 3

Plan de développement……... 4

Conception générale…….…... 6

Conception détaillée……..….. 8

Résultat……..……. 16

Documentation utilisateur……...……..17

Documentation développeur……. 17

Conclusion……..……... 18

Introduction

Nous voulons réaliser un mini-jeu inspiré de <*flappy bird*>. Nous remplacerons l'oiseau par un ovale, il tombera progressivement tout seul. Le but du jeu est d'empêcher l'ovale de sortir la ligne. Pour ce faire, le joueur peut cliquer sur l'écran pour élever l'ovale et la déplacer le long d'une ligne brisée. La vitesse à laquelle la ligne brisée se déplace augmentera avec le temps, donc la difficulté du jeu augmentera également.

Voici à quoi ressemble l'interface graphique du jeu:

Analyse globale

Pour réaliser le jeu, plusieurs fonctionnalités principales sont attendues:

1. l’interface graphique avec l’ovale et la ligne brisée
2. le défilement automatique de la ligne brisée
3. la réaction de l’ovale aux clics de l’utilisateur

Ensuite plusieurs fonctionnalités supplémentaires:

1. Amélioration des dessins
2. Ajouter d’éléments de décors

(Ils seront divisés en 3 parties pour réaliser)

La première partie:

- Créer une fenêtre avec l’ovale;
- Déplacer l’ovale vers le haut, lorsqu’on clique dans l'interface.

La deuxième partie:

- L’ovale perde de l’altitude tout seul;
- Dessiner la ligne brisée dans la fenêtre;
- Défilement automatique de la ligne brisée.

La troisième partie:

- Détecter les collisions;
- Amélioration des dessins;
- Ajouter des éléments de décors.

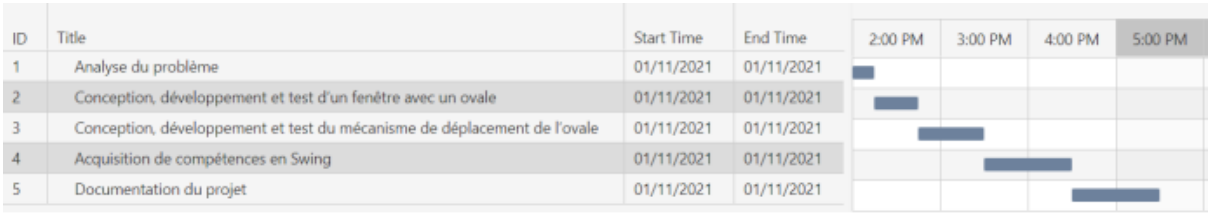
Plan de développement

- La première partie du projet (séance 1)

  Liste des tâches :

Tâches	Début	Durée	Fin
Analyse du problème	14:00	15mn	14:15
Conception, développement et test d'un fenêtre avec un ovale	14:15	30mn	14:45
Conception, développement et test du mécanisme de déplacement de l'ovale	14:45	45mn	15:30
Acquisition de compétences en Swing	15:30	60mn	16:30
Documentation du projet	16:30	60mn	17:30

  Diagramme de Gantt

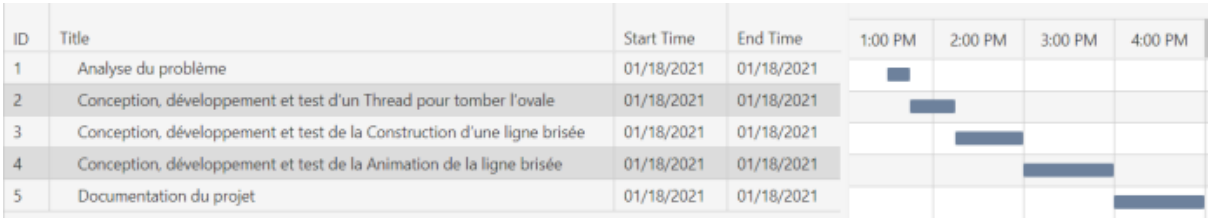


- La deuxième partie du projet (séance 2)

  Liste des tâches :

Tâches	Début	Durée	Fin
Analyse du problème	13:30	15mn	13:45
Conception, développement et test d'un Thread pour tomber l'ovale	13:45	30mn	14:15
Conception, développement et test de la construction d'une ligne brisée	14:15	45mn	15:00
Conception, développement et test de l'animation de la ligne brisée	15:00	60mn	16:00
Documentation du projet	16:00	60mn	17:00

Diagramme de Gantt



ID	Title	Start Time	End Time
1	Analyse du problème	01/18/2021	01/18/2021
2	Conception, développement et test d'un Thread pour tomber l'ovale	01/18/2021	01/18/2021
3	Conception, développement et test de la Construction d'une ligne brisée	01/18/2021	01/18/2021
4	Conception, développement et test de la Animation de la ligne brisée	01/18/2021	01/18/2021
5	Documentation du projet	01/18/2021	01/18/2021

- La troisième partie du projet (séance 3)

  Liste des tâches :

Tâches	Début	Durée	Fin
Analyse du problème	13:30	15mn	13:45
Calcule la formule pour trouver la position ordonnée de la ligne droite correspondant à la position en abscisse du centre de l'ovale	13:45	45mn	14:30
Conception, développement et test de la détection l'ovale est sorti de la ligne brisé	14:30	30mn	15:00
Conception, développement et test de la situation à la fin du jeu	15:00	60mn	16:00
Documentation du projet	16:00	60mn	17:00

Diagramme de Gantt

ID	Title	Start Time	End Time
1	Analyse du problème	01/25/2021	01/25/2021
2	Calcule la formule pour trouver la position ordonnée de la ligne droite correspondant à la position en abscisse du centre de l'ovale	01/25/2021	01/25/2021
3	Conception, développement et test de la détection l'ovale est sorti de la ligne brisée	01/25/2021	01/25/2021
4	Conception, développement et test de la situation à la fin du jeu	01/25/2021	01/25/2021
5	Documentation du projet	01/25/2021	01/25/2021

Conception générale

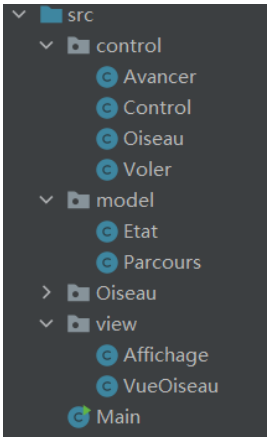
Tout d'abord, l'interface graphique s'est construite autour le Modèle Vue Contrôleur (MVC) qui permet de bien organiser le code source.

Le but est de diviser la logique du code en trois parties:

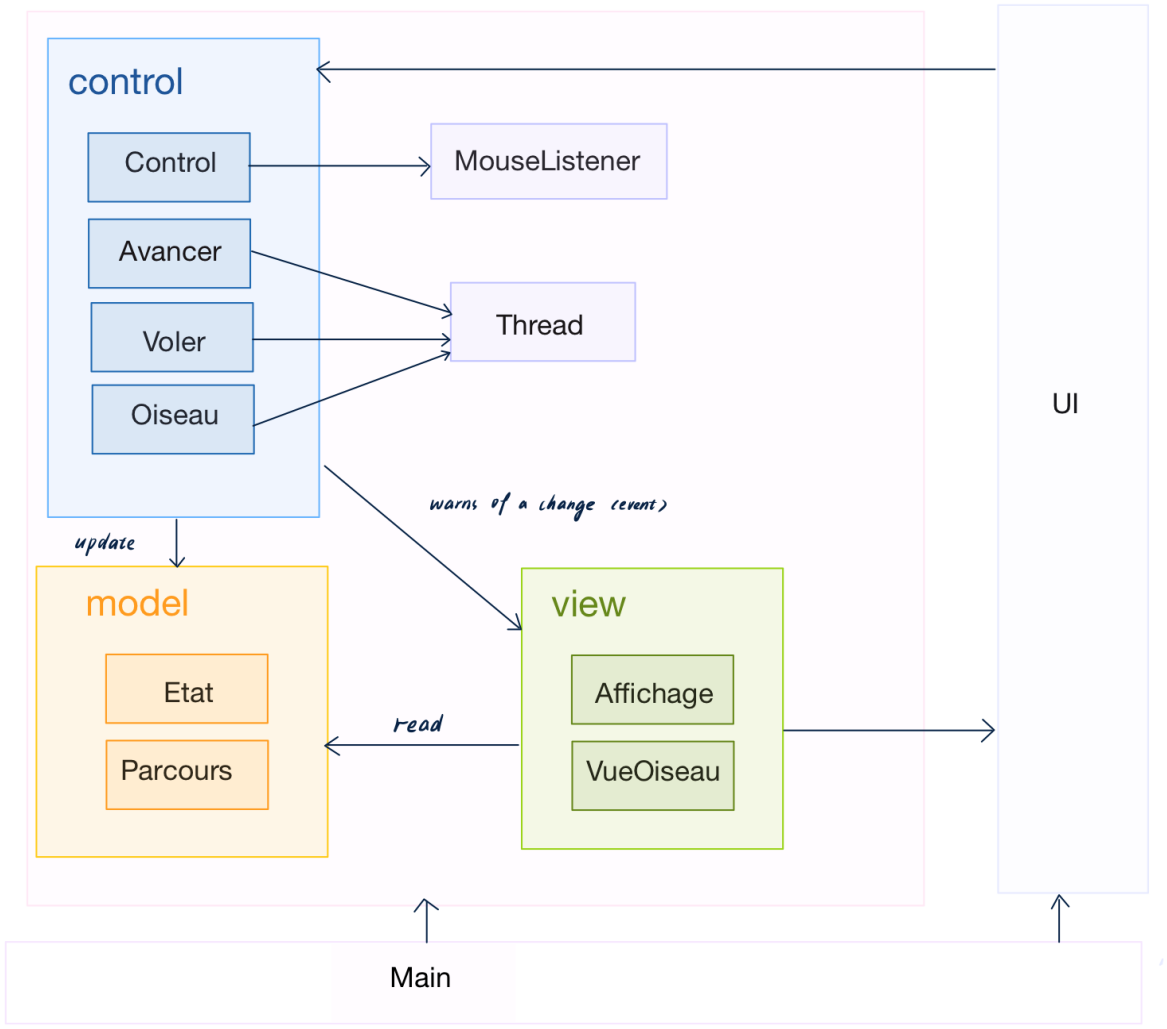
- Le **Modèle** <model>: définit l'ensemble des données qui caractérisent l'état de l'interface. La modification de ces données correspond à un changement de l'affichage dans l'interface graphique.
- La **Vue** <view>: s'occupe de dessiner l'interface, c'est a dire, une fenêtre avec de l'ovale et de la ligne brisée.
- Le **Contrôleur** <Control>: effectue les changements dans l'état et informe la vue d'un changement. Lorsque l'interface est interactive, c'est lui qui gère les événements.

Dans ce projet, nous avons trois packages, qui représentent respectivement le modèle, la vue et le contrôle, comme le montre la figure ci-dessous.

De plus, 8 images sont placées dans le package <oiseau> pour dessiner l'oiseau dynamique



Ensuite, les fonctionnalités sont introduites sous la forme de <blocs fonctionnels>.



Conception détaillée

Au début du projet, on voulait créer une fenêtre avec l'ovale utilisant l'API Swing et la classe ***JPanel***. Les dimensions de l'oval et de la fenêtre sont définies dans des constantes.

On définit l'ordonnée de l'ovale dans la classe ***Etat***(L'initialisation est égale à ***Oval_y***), puis utilise la méthode <*drawOval*> de la classe ***Graphics*** dans la classe ***Affichage*** pour la dessiner.

Les constantes de la classe ***Affichage***:

- **LARG** : Largeur de l'interface
- **HAUT** : Hauteur de l'interface
- **LARG_Oval**: Longueur de l'ovale
- **HAUT_Oval**: Hauteur de l'ovale
- **Oval_x** : Centre du ovale
- **Oval_y** : Centre du ovale

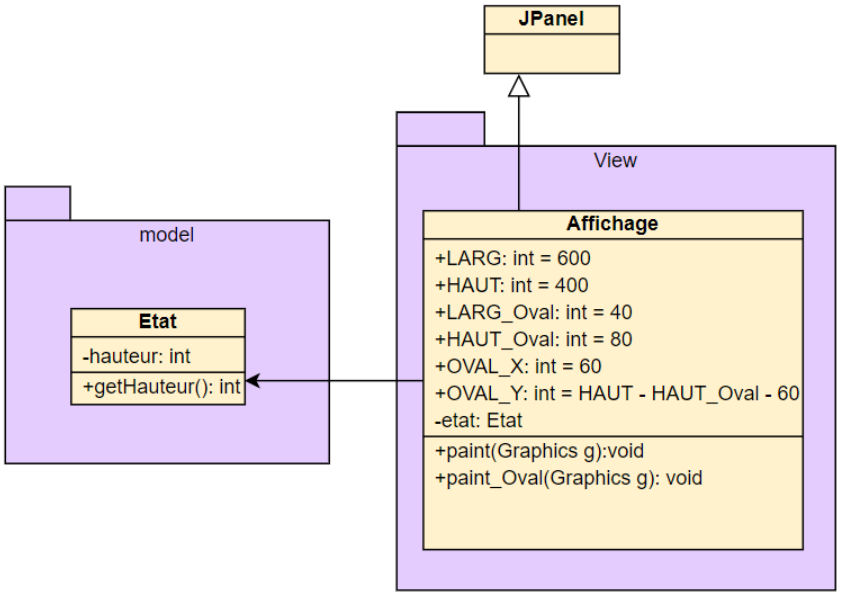


Figure-1 du l'interface graphique avec l'ovale

Afin de faire sauter l'ovale lorsque le joueur clique, on utilise la programmation événementielle avec la classe ***Control*** qui implémente l'interface <*MouseListener*>. La hauteur du saut est définie dans une constante de la classe ***Etat***.

La classe ***Control*** peut réagir aux interactions du joueur. Lorsque l'événement est capturé, le Contrôle met à jour la hauteur de l'ovale en appelant la méthode <*jump*> de l'etat. La méthode <*repaint*> de l'affichage est ensuite appelée par le Control pour redessiner l'interface.

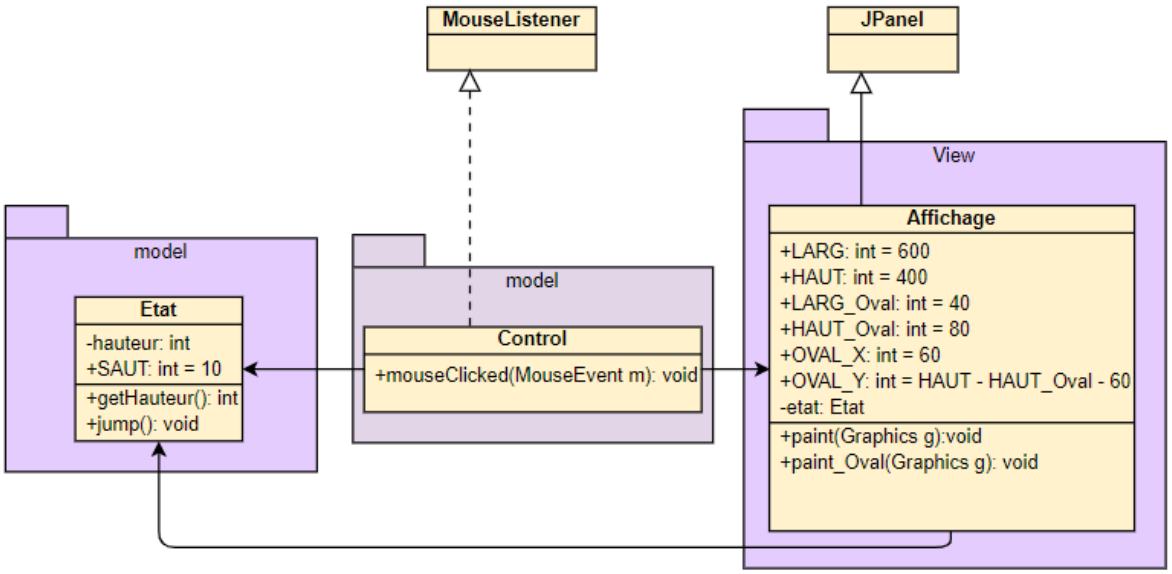


Figure-2 du mécanisme de déplacement de l'ovale

De plus, l'ovale doit perdre de l'altitude tout seul.

On ajoute une méthode <*moveDown*> dans la classe ***Etat*** qui permet de modifier la valeur de la hauteur de quelques pixels vers le bas (la chute est définie dans une constante dans la classe ***Etat***), sans sortir de la zone de dessin.

La classe ***Voler*** hérite du ***Thread*** et elle appelle d'abord la méthode <*moveDown*>. On utilise <*Thread.sleep*> à chaque tour de boucle et puis redessiner l'ovale par la méthode <*revalidate*> et <*repaint*> de la classe ***Affichage***.

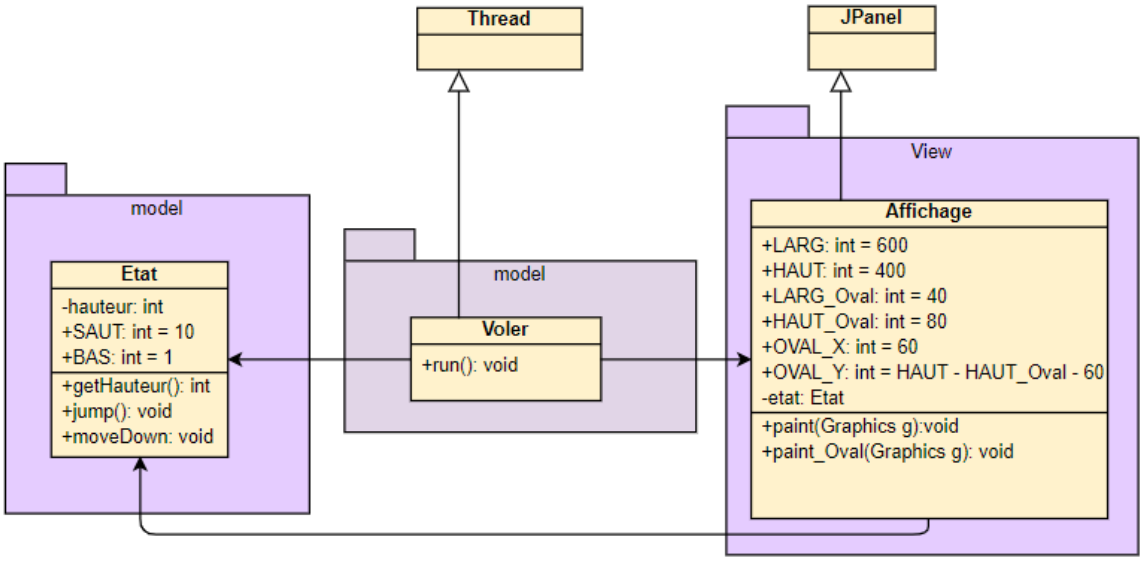


Figure-3 de tomber l'ovale

Il est temps d'ajouter la ligne brisée au jeu. On a créé une classe ***Parcours*** dont l'attribut principal est un objet *ArrayList<Point>* (une liste de points) qui permet de définir des objets caractérisés par des coordonnées (x,y). Et utilise *<drawLine>* dans la classe ***Affichage*** pour afficher la ligne brisée après récupérer la liste de points.

Dans le constructeur, on stocke le centre de l'ovale dans la liste de points comme point de départ de la ligne brisée, la méthode *<initList>* est appelée ensuite qui prend en charge l'initialisation.

Pour initialiser la liste de points:

Constructeur: Parcours()

```
Liste_Points = new ArrayList<Point>()

int x <-  Affichage.Oval_x + Affichage.LARG_Oval / 2;

int y <- Affichage.Oval_y + Affichage.HAUT_Oval / 2;

Liste_Points.add( new Point(x,y) );

initList();
```

initList():

```
int x <- Liste_Points[length].x; int y;
Faire
    x <- x + Random(Affichage.LARG);
    y <- Random(Affichage.HAUT - 2*Affichage.HAUT_Oval)
        + Affichage.HAUT_Oval;
    Liste_Points.add( new Point(x,y) );
Tant que x < Affichage.LARG
```

Cette liste est parcourue dans la classe ***Affichage***, et à l'aide de la méthode *<drawLine>*, les points sont reliés pour former une ligne.

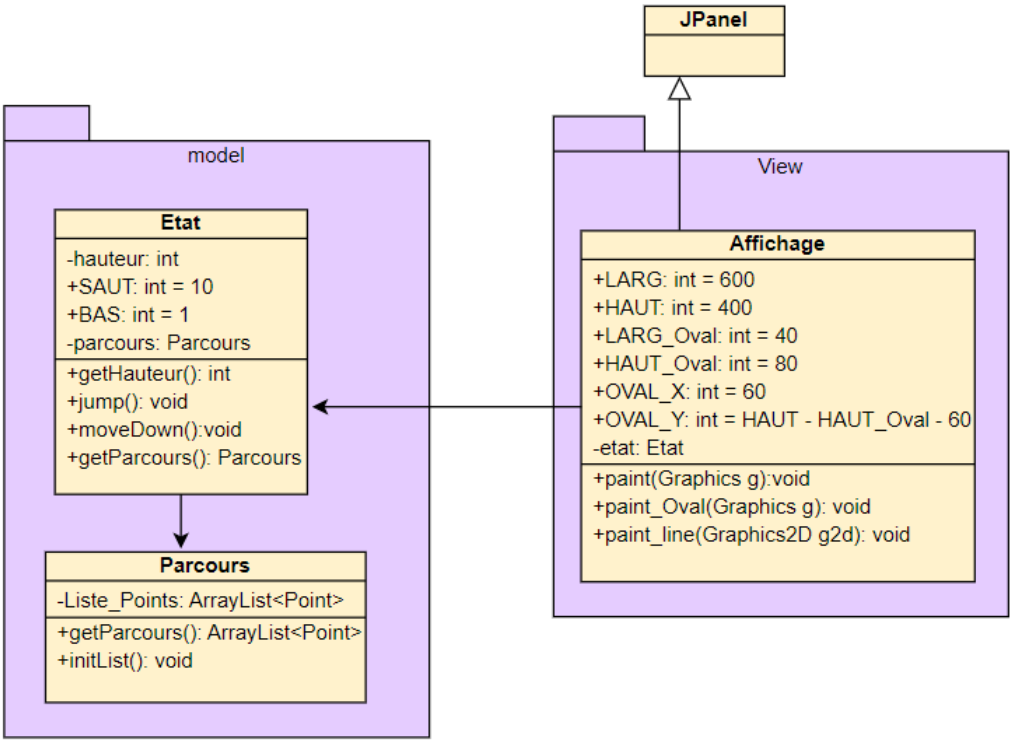


Figure-4 de la construction d'une ligne brisée

Maintenant on veut faire le défilement automatique de la ligne brisée. La classe ***Avancer*** hérite de la classe ***Thread*** qui appelle la méthode *<setPosition>* dans la classe ***Parcours*** pour ajouter une valeur fixe à l'abscisse de tous les points de la liste de points à l'infini. Sa structure de code est similaire à la classe ***Voler***.

La valeur du déplacement est définie dans une constante dans la classe ***Parcours***:

AVANCER_POSITION = 1

Pour continuer le jeu, la ligne est infinie dans l'interface. On doit donc ajouter de nouveaux points à la liste de points le cas échéant:

```
//Lorsque le dernier point rentre dans la fenêtre visible
Si Liste_Points[Liste_Points.length].x < Affichage.LARG alors
    générer aléatoirement un nouveau point
    Liste_Points.add( point )
```

Et, on supprime les points inutiles :

```
//Lorsque le deuxième point sur de la zone visible,
Si Liste_Points[0].x < 0 alors
    Liste_Points.remove(0);
```

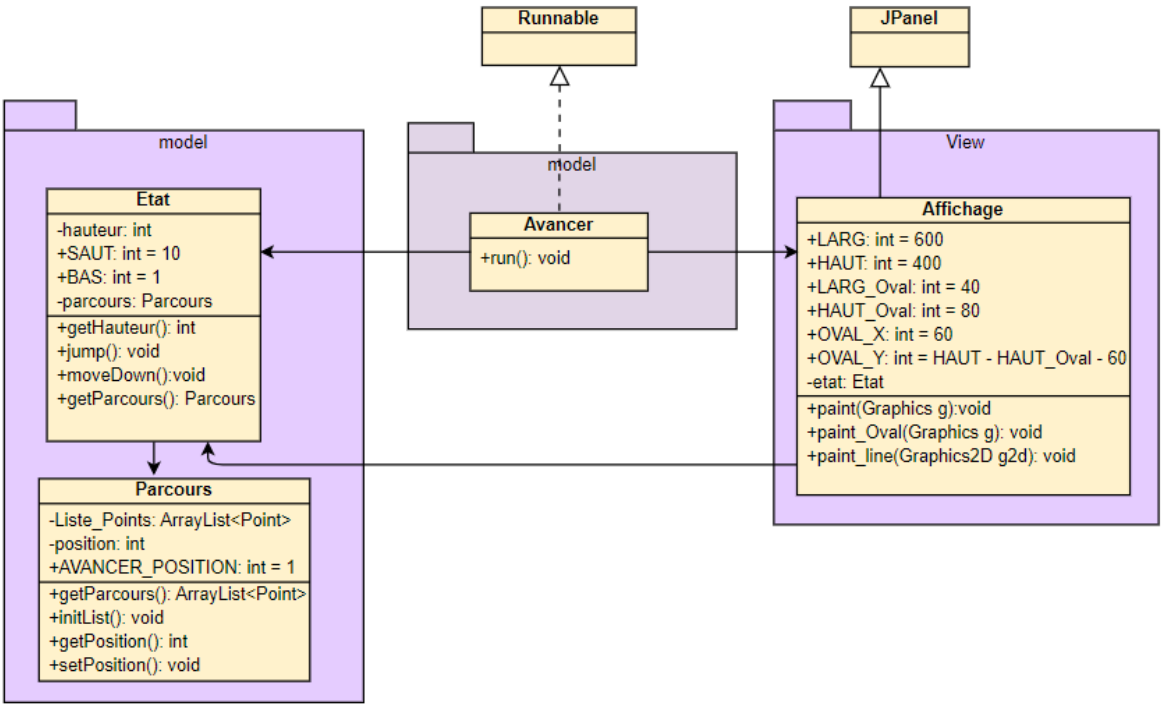


Figure-5 du défilement automatique de la ligne brisée

Une étape importante pour compléter la mécanique du jeu est la détection des collisions. On a une méthode <*testPerdu*> dans la classe ***Etat*** qui est utilisé pour calculer et retourner true si la ligne brisée n'est pas dans l'ovale. Le résultat de retour de cette méthode sera utilisé comme condition de boucle du thread dans le package ***control***.

Tout d'abord, nous parcourons la liste des points et comparons l'abscisse du centre de l'ovale x avec les abscisses des éléments de la liste, afin de trouver les deux points (x1, y1), (x2, y2) sur les côtés gauche et droit de l'ovale. Ensuite, nous avons calculé l'équation de la droite passant par ces deux points. Enfin, ramenez l'abscisse du centre de l'ovale dans cette formule pour obtenir l'ordonnée correspondant à la ligne brisée y_line.

Utilisez deux points pour représenter l'équation d'une ligne droite les traversant:

$$(x - x1) / (x2 - x1) = (y - y1) / (y2 - y1)$$

Donc

$$(x - x1) / (x2 - x1) = (y\_line - y1) / (y2 - y1)$$

$$y\_line = (x - x1) * (y2 - y1) / (x2 - x1) + y1$$

Ou soit l'équation de la droite:

$$y = kx + b$$

$$k = (y2 - y1) / (x2 - x1)$$

$$b = (x2*y1 - x1*y2) / (x2 - x1)$$

$$y\_line = kx + b$$

Les conditions de la détection de l'ovale n'est pas sorti de la ligne brisée sont:

y <= y_line et y_line <= (y + Affichage.HAUT_Oval)

Après avoir détecté l'ovale est sorti de la ligne brisée, les threads de vol et d'avancement du parcours seront arrêtés. Ensuite, le score de joueur sera affiché par un message utilisant la classe JOptionPane.

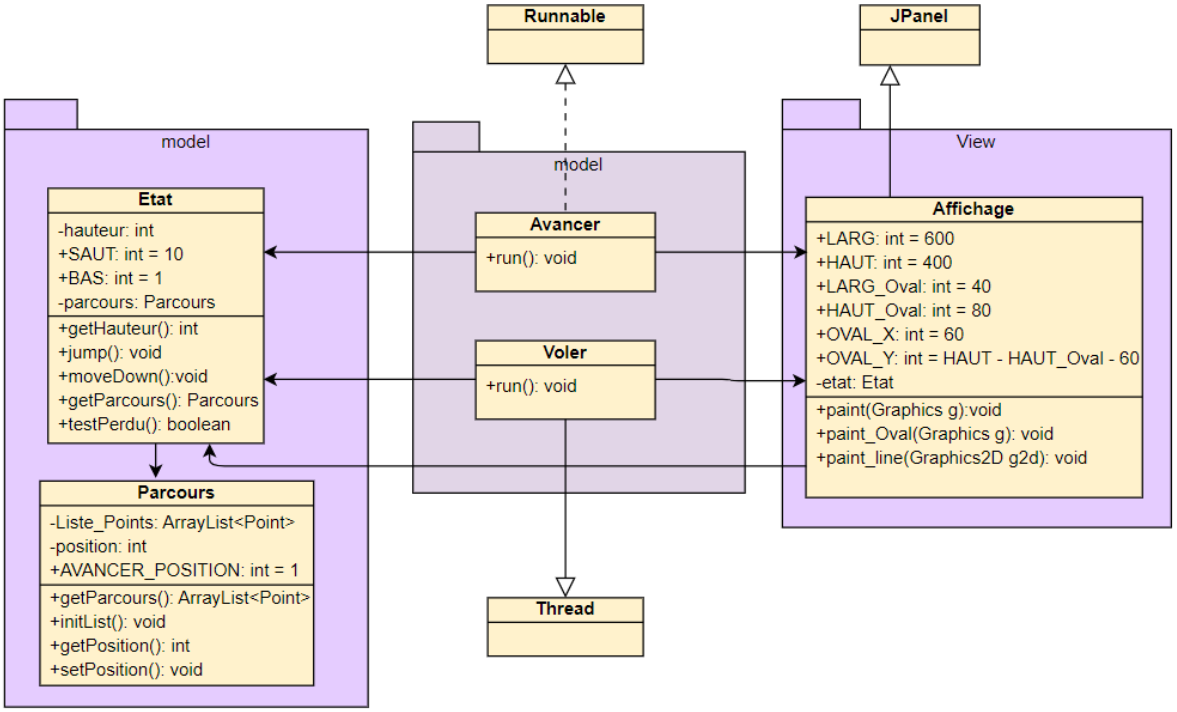


Figure-6 de la détection des collisions

Enfin on veut embellir le jeu.

Pour améliorer la qualité de l'affichage, on a implémenté des méthodes de la classe ***Graphics2D***. On a utilisé la méthode <*setStroke*> et <*setColor*> pour un ovale et une ligne brisée plus gros et plus coloré.

En plus, on a ajouté de l'élément de décor (un oiseau).

Les quatre attributs de l'oiseau sont définis dans la classe ***Oiseau***:

- **delai**: le temps (en millisecondes) entre chaque mise à jour de l'affichage pour l'oiseau
- **etat**: la position est l'oiseau
- **hauteur**: la hauteur de l'oiseau
- **position**: l'abscisse de l'oiseau

Le constructeur choisit une valeur aléatoire pour le délai et la hauteur. La position est fixée de manière à ce que l'oiseau soit complètement à droite, au-delà de la fenêtre visible. La fonction <*run*> met à jour l'état et la position de l'oiseau tous les **delai** millisecondes. Lorsque l'oiseau sort complètement la zone visible, le Thread s'arrête.

La classe ***VueOiseau*** possède un attribut de type *ArrayList<Oiseau>* et elle a une méthode <*dessiner*> pour gérer l'affichage des oiseaux. Cette méthode place dans g (0-7) l'image correspondant à l'état de l'oiseau, placée à la position courante, pour chaque élément de sa liste.

Enfin, l'affichage appelle la méthode <*dessiner*> pour le faire afficher dans l'interface graphique.

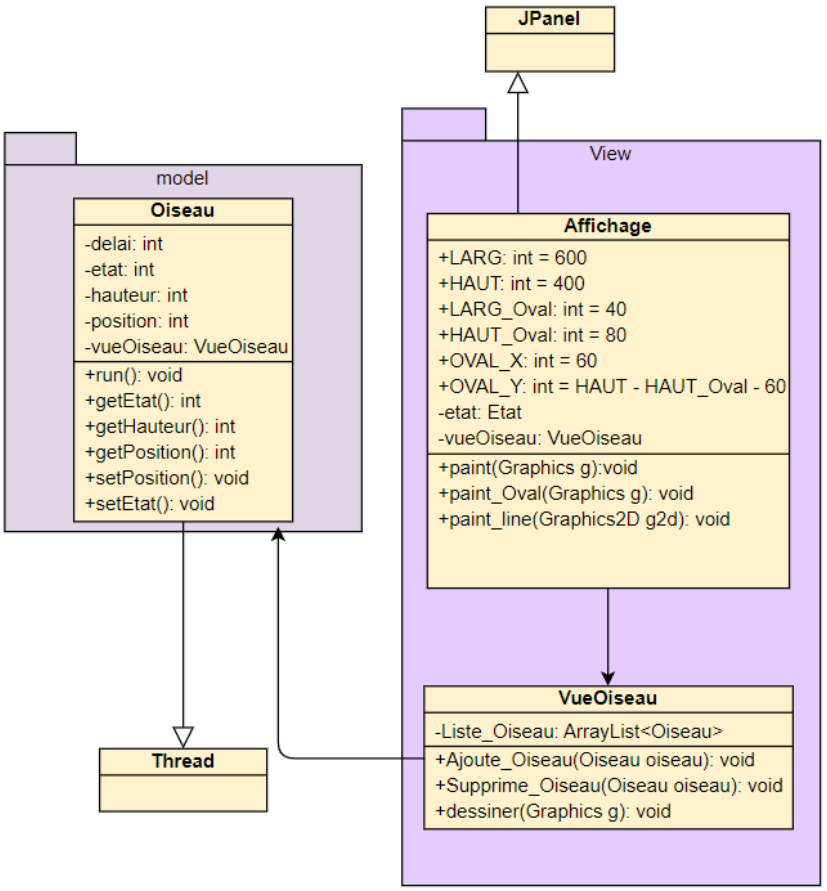
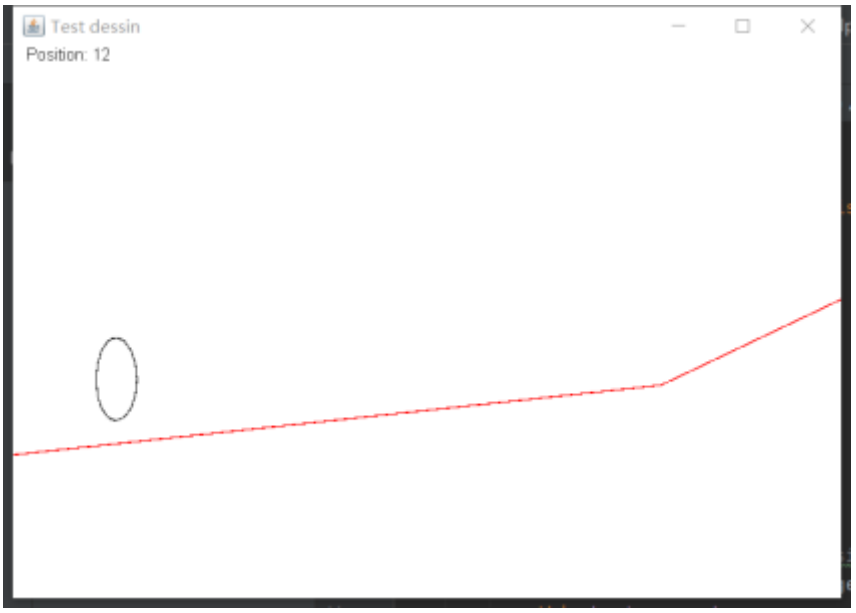


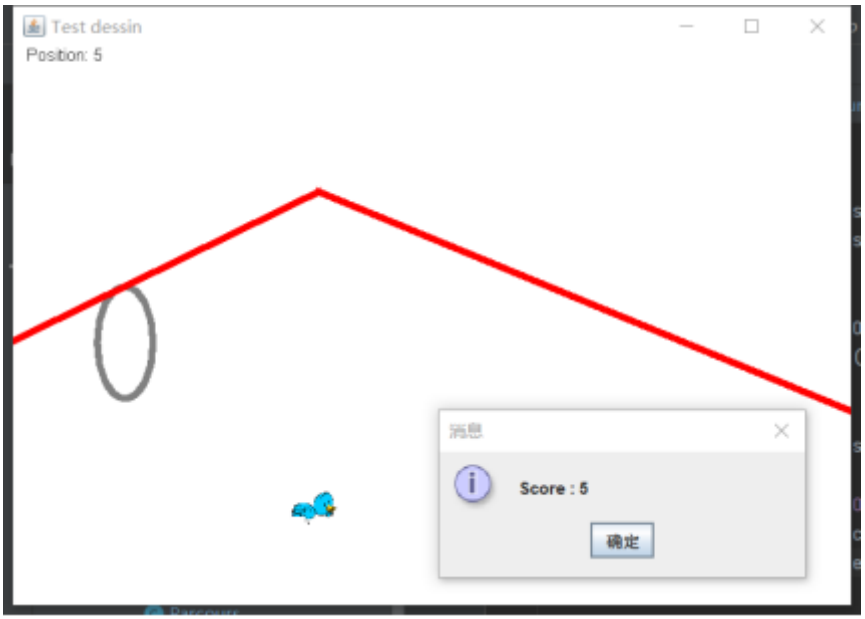
Figure-6 de l'ajoute de l'élément de décor

Résultat

- Dans la deuxième partie du projet (séance 2)



- Dans la troisième partie du projet (séance 3)



Documentation utilisateur

- Prérequis :
    - Java avec un IDE
    - Le logiciel requiert Java jdk >= 1.8
- Mode d'emploi:
    - Importez le projet dans votre IDE, sélectionnez la classe ***Main*** à la racine du projet, ici c'est la classe ***Main*** puis < *Run as Java Application* > (dans le cas Eclipse) ou < *Run 'Main.main()'* > (dans le cas IntelliJ IDEA).
    - Cliquez sur la fenêtre pour faire monter l'ovale.

Documentation développeur

C'est la classe ***Main*** qui contient la méthode <*main*>. Nous utilisons le modèle MVC pour organiser les codes afin de les rendre plus lisibles et propres.

Le package ***model*** définit l'ensemble des données qui caractérisent l'état de l'interface. Vous pouvez contrôler la taille du saut de l'ovale en modifiant la constante ***SAUT*** et sa vitesse de chute en modifiant la constante ***BAS*** dans la classe ***Etat***. Il est possible de changer la taille du déplacement de la ligne avec la constante ***AVANCER_POSITION*** dans la classe ***Parcours***.

Allez dans le package **view** pour voir comment afficher l'interface. Vous pouvez modifier les valeurs des constants définis dans la classe ***Affichage*** afin de changer la taille de la fenêtre avec les constantes ***HAUT*** et ***LARG***. Vous pouvez également changer la valeur de ***LARG_Oval***, ***HAUT_Oval***, ***Oval_x*** et ***Oval_y*** pour modifier la taille de l'ovale ainsi que ses coordonnées de départ.

Le package **control** vous permet de savoir comment gérer les événements et la manière dont l'état du modèle change. L'animation de la ligne et le mouvement de l'ovale sont définis ici. La constante ***TIMEAVANCER*** dans la classe ***Avancer*** vous permet de définir la vitesse de l'animation de la ligne.

Pour l'instant, je n'ajoute actuellement qu'un oiseau à l'arrière-plan, nous pouvons en avoir plus. Et nous pouvons aussi avoir un parcours plus élégant utilisant des courbes de Bézier.

Conclusion et perspectives

J‘ai réalisé une fenêtre dans laquelle est dessiné l’oval et la ligne. J’ai utilisé des Threads pour réaliser le défilement automatique de la ligne brisée vers la gauche afin que l'oiseau semble voler vers l'avant. L’ovale tombe tout seul. Afin de l'empêcher de quitter la ligne brisée, le joueur peut cliquer sur la fenêtre pour la faire sauter. Au fil du temps, la vitesse de la ligne se déplacera de plus en plus vite, donc la difficulté pour le joueur augmentera également. Une fois l’ovale détectée, elle a sorti la ligne. Le jeu s'arrêtera et un message sera émis pour indiquer au joueur le score.

Grâce à l'utilisation de Thread, chaque fois que le Thread est exécuté, l'oiseau charge une image différente, ce qui donne à l'oiseau un mouvement de vol cohérent.

Ce projet m'a permis de maîtriser la méthode d'utilisation des threads et de devenir plus compétent dans l'utilisation de Swing. Dans le processus de rédaction du rapport, cela m'a également aidé à mieux comprendre les principes de fonctionnement de mon projet.

J'ai rencontré quelques difficultés lors de la conception du mécanisme de collision. Au début, je n'ai pas pensé à utiliser la méthode de projection, j'ai passé beaucoup de temps à calculer la formule tangente ou croisée d'une droite et d'une ellipse. Et je veux calculer une formule liée à la vitesse de déplacement de la ligne et à la vitesse de descente de l'ellipse, ce qui peut raisonnablement concevoir la pente de la ligne. Mais je ne l'ai pas trouvé.

Il existe également des optimisations pour ce projet, telles que l'embellissement de l'image, avec plus de décorations, la transformation de la courbe en une courbe de Bézier, la conception de pente de ligne plus raisonnable. La difficulté du jeu peut être sélectionnée.