**DONCECCHI Carlo Elia**

**Rapport projet PCII**

Introduction

Nous voulons réaliser une version très simplifiée un mini-jeu inspiré de flappy bird. Lors de ces séances, afin de familiariser avec la notion d’interface interactive nous avons commencé à implémenter les bases d’un jeu dans lequel le jouer contrôle un ovale à l’aide de sa souris. Le joueur interagit avec l’interface en faisant bouger l’ovale verticalement, grâce à sa souris, afin qu’elle reste sur le chemin représenté par une ligne rouge. Cette implémentation nous a aussi permis de : manipuler les motifs de conception qui nous permettent d’organiser les différentes classes qui constituent un projet, particulièrement en Java ; et manipuler les threads.

Analyse globale

Les fonctionnalités que nous avons traitées lors de ces séances sont : la création de l’interface graphique ; la réaction de l’ovale aux clics du joueur ; la création et animation de la ligne brisée ; et la gestion de l’arrêt de la partie et de la détection des collisions.

Pour ce qui est de l’interface graphique le sujet nous fournissait la plupart des notions nécessaires à son implémentation.

Pour l’implémentation de la réponse aux clics de la souris la difficulté était : de comprendre par où commencer ; et de comprendre l’implémentation des threads.

Pour l’implémentation de la création de l’animation de la ligne brisée, la difficulté était de faire en sorte que le tout soit cohérent avec l’implémentation des autres classes. Sa sous-fonctionnalité principale est l’initialisation de ligne brisée dont on va fournir le pseudo-code.

Pour l’implémentation de la gestion de l’arrêt, la difficulté était de faire en sorte que le résultat de la formule qui permet de déterminer la valeur de l’ordonnée sur la ligne brisée soit exacte. Ses sous-fonctionnalités sont : l’arrêt des threads ; l’arrêt de la réaction aux clics de la souris ; et le message qui affiche le score de l’utilisateur.

Plan de développement

Je vais traiter les 3 séances de la conception du jeu séparément.

* Première séance :
* Liste des tâches
* Analyse du problème (15 min)
* Conception, et développement d’une fenêtre avec un ovale (30 min)
* Conception, développement et test du mécanisme de déplacement de l’ovale (45 min)
* Acquisition des compétences en Swing (60 min)
* Documentation du projet (60 min)
* Diagramme de Gannt
* Deuxième séance :
* Liste des tâches
* Analyse du problème (30 min)
* Conception, développement et test d’une fenêtre avec un ovale (45 mn)
* Conception, développement et test du mécanisme de déplacement de l’ovale avec implémentation du thread pour le vol (1 h)
* Conception de la construction de la ligne brisée, et de son animation (2h)
* Acquisition de compétences en Swing (60 mn)
* Documentation du projet (1h30)
* Diagramme de Gantt
* Troisième séance
* Liste des tâches
* Analyse du problème (30 min)
* Conception du calcul de la valeur de l’ordonnée sur la ligne brisée (60 min)
* Implémentation de la méthode testPerdu (1h15 min)
* Mise en relation avec tout le programme (1h)
* Documentation du projet (30 min)
* Diagramme de Gannt

Conception générale

MODELE

(Classe

Avancer/Etat/

Parcours/Voler

CONTROLEUR

(Classe Control)

La vue lit les modifications dans le modèle

VUE

(Classe

Affichage)

Fonctionnalité d’implémentation de l’interface graphique

La fonctionnalité liée au clic de la souris concerne les deux flèches vertes, c’est-à-dire les différentes modifications que le contrôleur apporte au modèle et à la vue.

Dans le modèle on retrouve : l’implémentation des threads ; de la ligne brisée ; et détection des collisions.

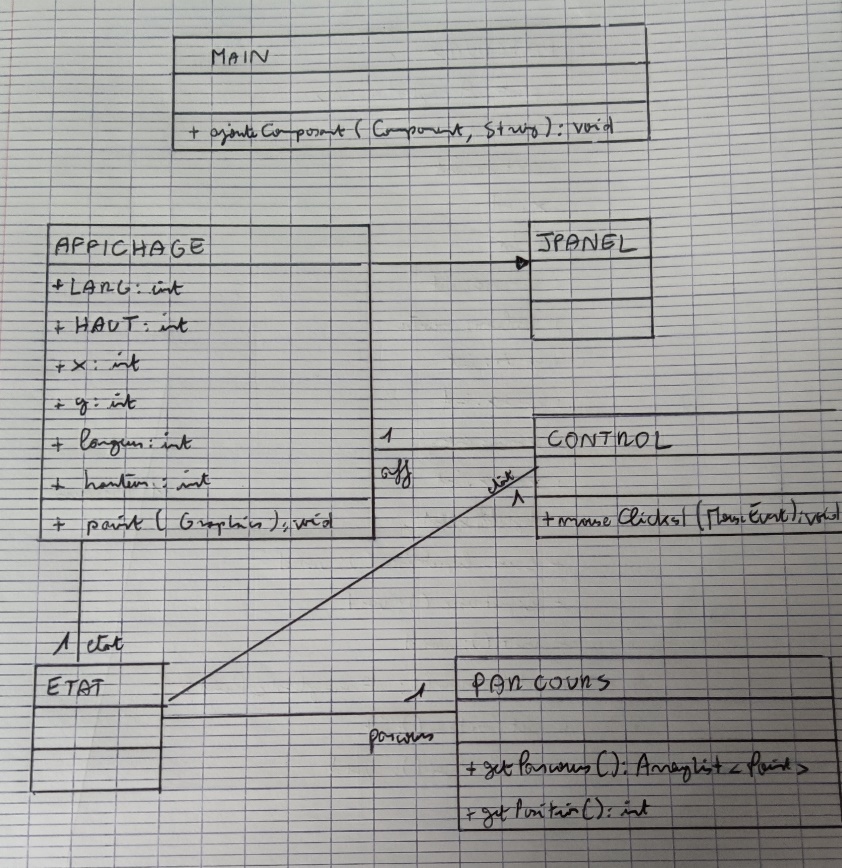
Conception détaillée

* Pour la fonctionnalité d’implémentation de l’interface graphique on utilise la classe **Affichage** qui extends la classe **JPanel**. Les dimensions de la fenêtre sont données par les constantes de classe **LARG**(largeur de la fenêtre) et **HAUT**(hauteur de la fenêtre). Pour l’ovale on la dessine grâce à la méthode **drawOval,** de la classe **Graphics**, qui prend en paramétré les constantes de classes : **x**,**y**,**Largeur**,**hauteur**, qui sont les dimensions de l’ovale. La méthode **drawOval** est appelée dans la méthode **paint** de classe **JPanel** qu’on a réimplémentée dans la classe **Affichage**. La méthode **repaint** est appelée dans la méthode **mouseClicked** de la classe **Control**.

Ensuite on crée un objet de la classe **Affichage** dans la fonction main et on le passe en paramètre à la méthode **ajouteComposant** dans la classe **Main.** Cet objet est le composant qu’on va ajouter à une fenêtre de la classe **JPanel** qu’on a créé dans la méthode.

Pour ce qui est de l’affichage de la ligne brisée et des points, toujours dans la méthode **paint**:

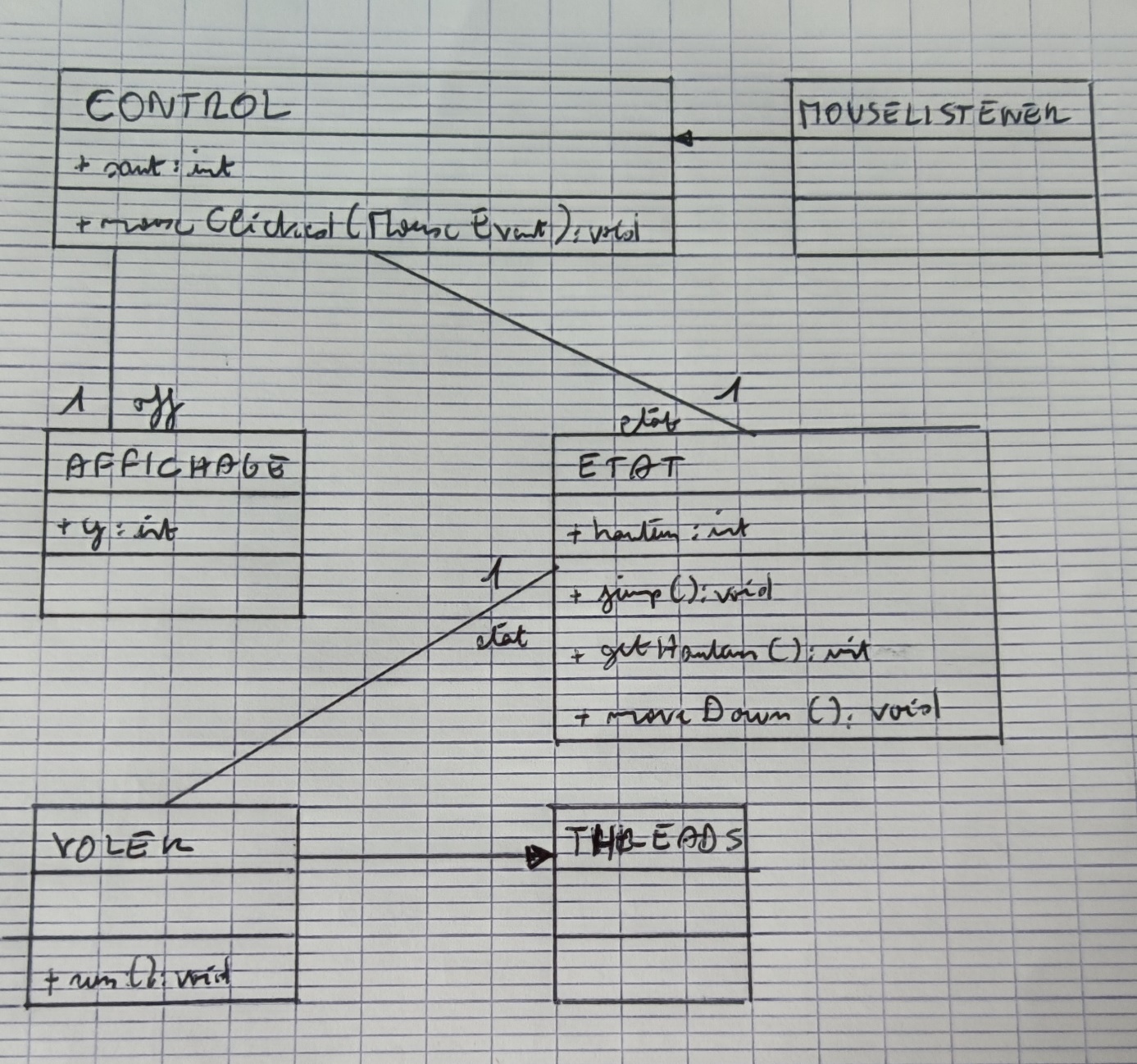
* Pour l’affichage de la ligne brisée on utilise la méthode **getParcours** de la classe **Parcours** qui est un attribut de l’attribut **etat** ( de type **Etat**) de la classe **Affichage**.
* Pour l’affichage des points on récupère l’entier renvoyé par la méthode **getPosition** de la classe **Parcours** qui est un attribut de l’attribut **etat**(de type **Etat**) de la classe **Affichage**; on affiche les points dans un carré blanc qu’on crée à l’aide de la méthode **clearRect** de la classe **Graphics**.



* Pour la fonctionnalité d’implémentation de la réponse aux clics de la souris on utilise **Control** qui implémente la classe **MouseListener**. Pour ce qui est de ce programme on a besoin d’implémenter la méthode **mouseClicked** de la classe **MouseListener**; pour changer la position de l’ovale la méthode **mouseClicked** appelle la méthode **jump** de la classe **Etat.** La classe **Control** et la classe **Affichage** ont dans leurs attributs une instance de la classe **Etat**.

La méthode **jump** change la position de l’ovale représentée dans la classe **Etat** par l’attribut **hauteur** initialisé grâce à l’attribut **y** de la classe **Affichage**. La modification n’est effectuée que si la nouvelle position est strictement supérieure à 0.

Pour l’implémentation de la perte d’altitude de l’ovale on part de la méthode **moveDown** de la classe **Etat** qui change la hauteur de l’ovale de quelques pixel vers le bas grâce à la constante **saut** de la classe **Control**. On vérifie, comme pour la méthode **jump**, que l’ovale ne sorte pas du cadre. Ensuite on implémente la descente automatique de l’ovale lorsque le joueur ne clique pas grâce à la classe **Voler** qui dérive de **Thread.** Dans la méthode **run** on appelle la méthode **movedown** de l’attribut **etat**(de type **Etat**). Entre chaque chute nous mettons une pause de 500 millisecondes. Dans la méthode **run** on appelle la méthode **repaint**.



* Pour la fonctionnalité de la ligne brisée le tout est implémenté grâce à la classe **Parcours**
* Pour sa construction nous allons fournir le pseudo code de la fonction qui initialise la ligne brisée :

Fonction initialise : type void

Objectif : valeur tirée au hasard

Vitesse : 1 (raideur de la pente)

On ajoute le premier point à la liste

Montée : faux (direction de la pente)

Tant que Objectif < Largeur de l’affichage graphique

Limite = faux (pour savoir si la limite est atteinte)

Tant que x =/= Objectif

x : x+1

Si Montée

y : y – Vitesse

Si y<ymin

Limite : vrai

y : ymin

x : Objectif

Montée : false

Fin Si

Sinon

y : y + Vitesse

Si y>ymax

Limite : vrai

y : ymax

x : Objectif

Fin Si

Fin Si

On ajoute le point

Objectif : valeur tirée au hasard

Si Limite = faux

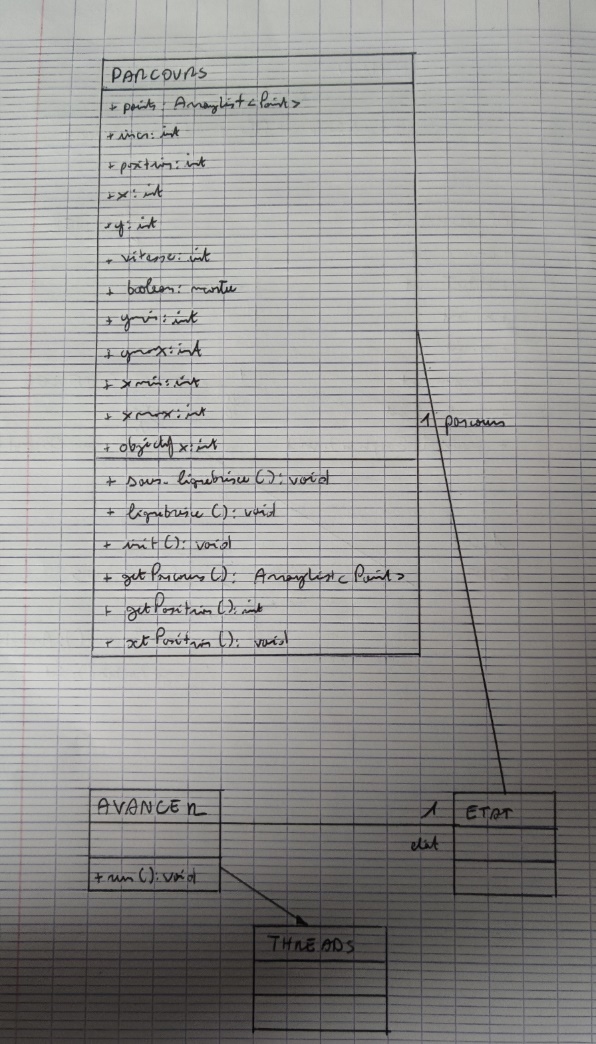
On donne une valeur à Montée

Fin Si

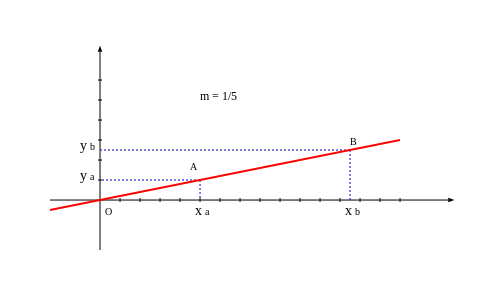
Fin du Tant que

Ajout du point en dehors de l’affichage grâce à la méthode décrite dans la boucle

* Pour la mise à jour de la ligne brisée, qui ne renvoie que les points visibles on utilise la méthode **getParcours** qui ajoute un nombre de points égal à la valeur de l’incrémentation de l’attribut **position.** Cette incrémentation est donnée par la constante de classe **incr**. On renvoie une liste qui contient les points à afficher.
* Pour le défilement de la ligne on utilise la classe **Avancer** qui dérive de **Thread** et appelle la méthode **setPosition** de la classe **Parcours** en passant par l’attribut **etat** (de type **Etat**). La méthode **setPosition** incrémente l’attribut **position** de la classe **Parcours** grâce à la constante de classe **incr**.



* Pour la fonctionnalité de la gestion de l’arrêt de la partie et la détection des collisions



* La formule qu’on utilise pour la détection des collisions est :

O(xO,yO)

A(xA,yA)

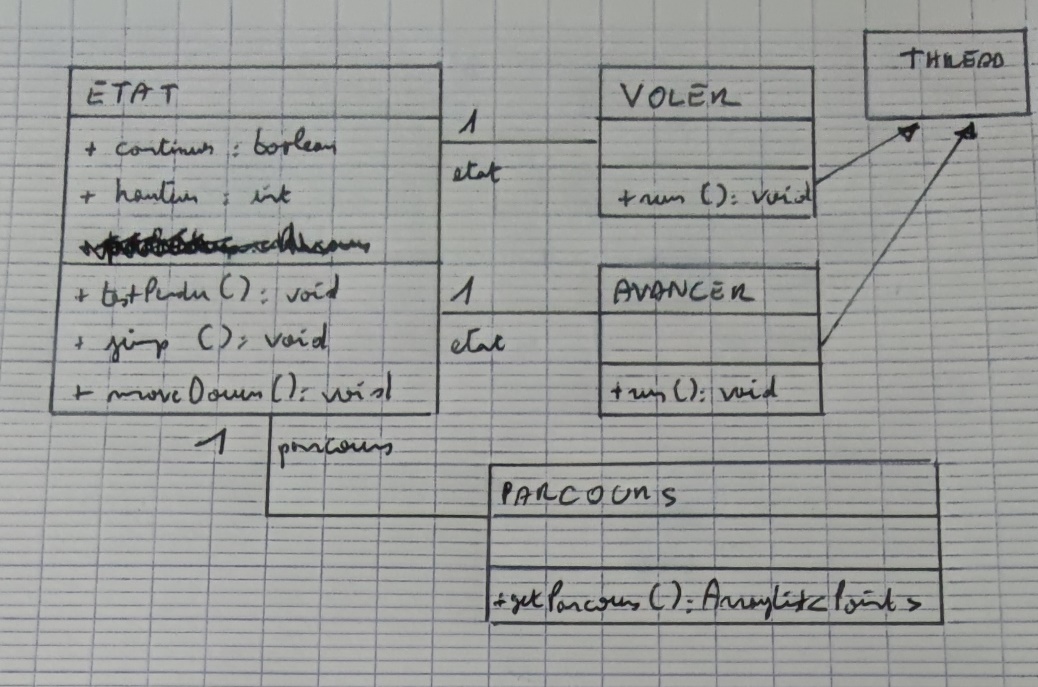
B(xB,yB)

Pente = (yB-y0)/(xB-x0)

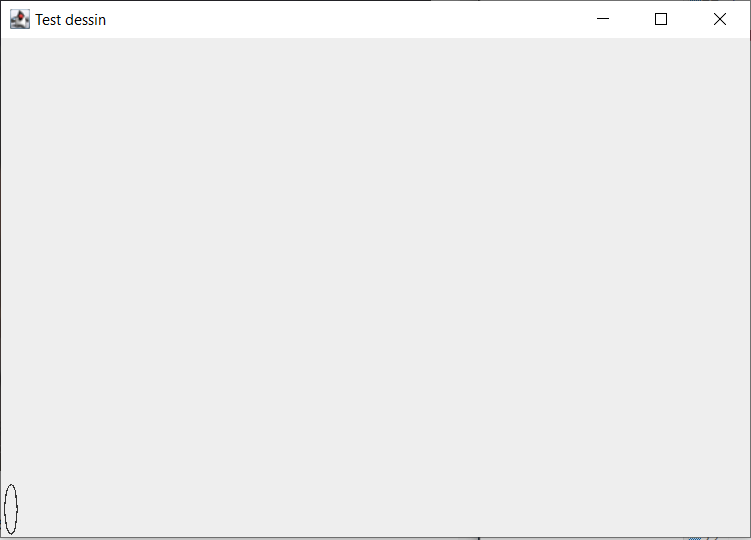
Pente = (yB-yA)/(xB-xA)

yA = yO – Pente\*(xO-(Distance entre x=0 et x=Milieu de l’ovale))

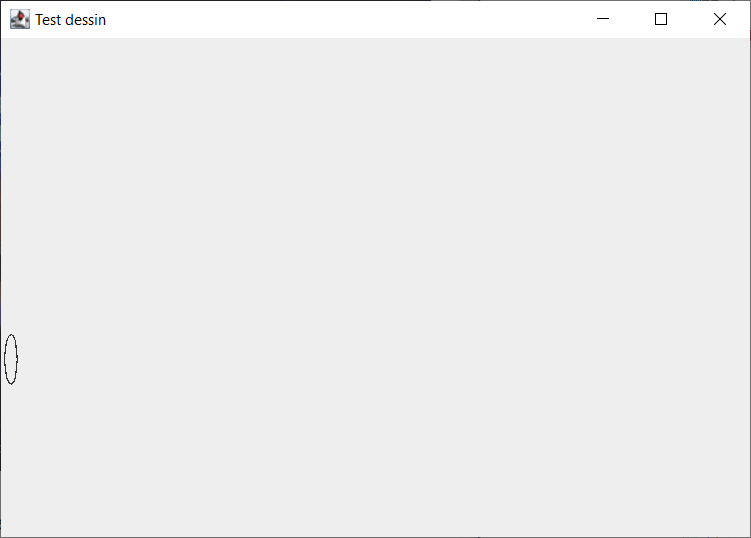
* Ce calcul est effectué dans la méthode **testPerdu**  de la classe **Etat**, les points A et B sont le premier point et le deuxième de la liste renvoyée par la méthode **getParcours** de la classe **Parcours**; toujours dans la méthode **testPerdu**. Ensuite à l’aide de la valeur max et min de l’ordonnée de l’ovale on vérifie qu’il n’y a pas eu de collision.
* La méthode **testPerdu** est appelée dans les méthodes **moveDown** et **jump** de la classe **Etat**. S’il y eu une collision on met l’attribut **continuer** de la classe **Etat** à faux ce qui va arrêter les méthodes **run** des Threads **Voler** et **Avancer**. De plus la méthode **jump** ne pourra plus être exécutée ce qui va entrainer que l’interface ne réagira plus aux clics.
* Si une collision est détectée la méthode **testPerdu** affiche un message qui affiche le score grâce à la méthode **showMessage** de la classe **JOptionPane**.



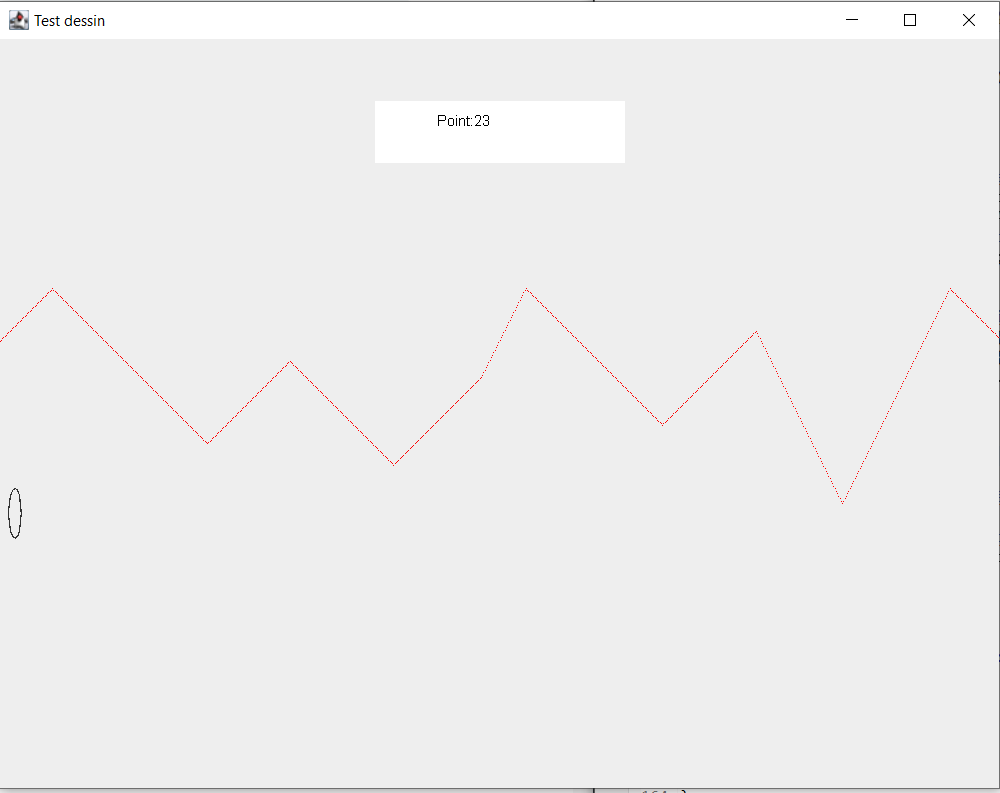
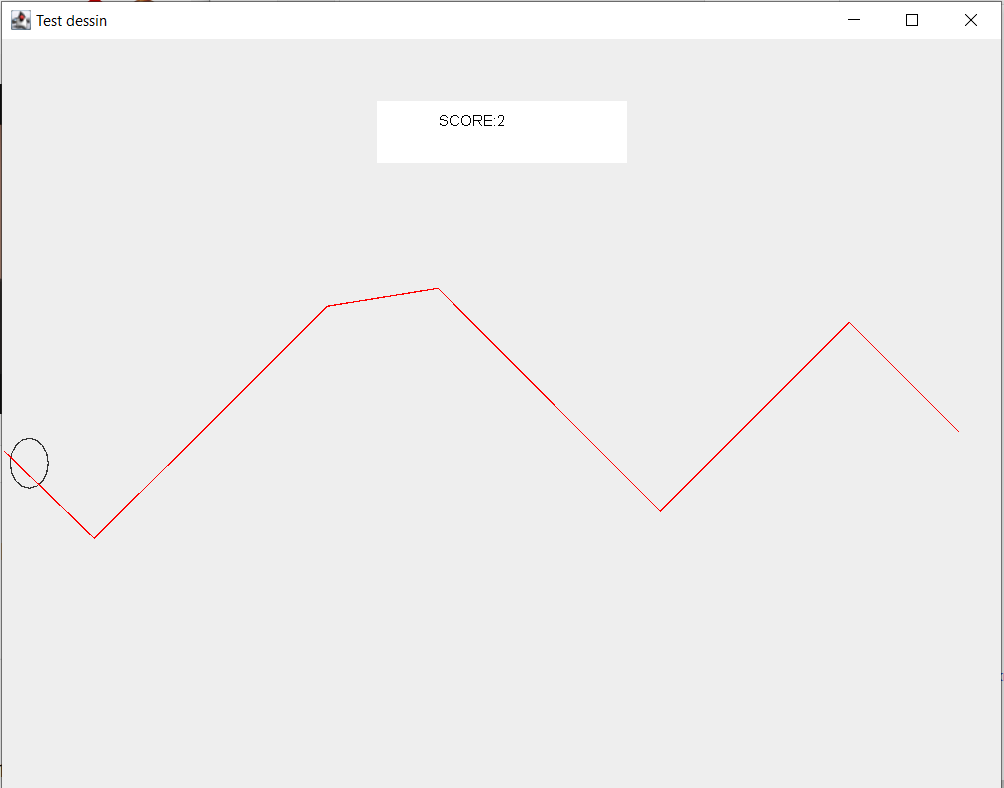
Résultat



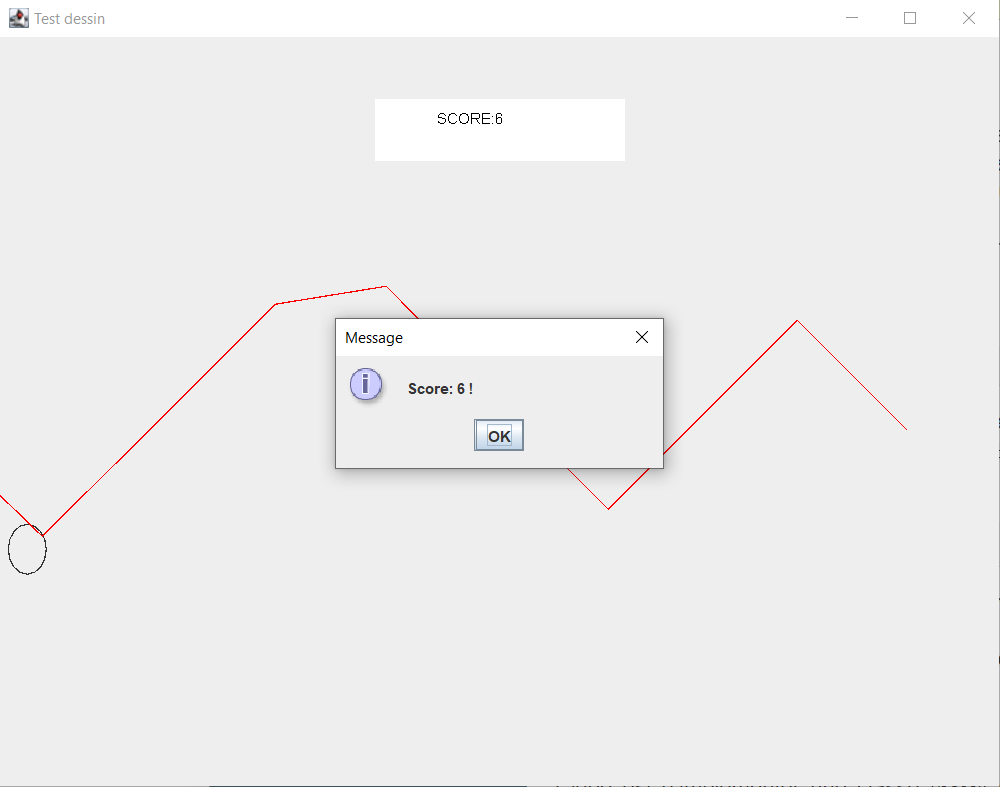
*Fenêtre qui apparait lorsqu’on change lance le programme*



*Position de l’ovale après plusieurs clics*

*Affichage avec l’implémentation de la ligne brisée*



*Perte de la partie*

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.

Documentation développeur

Les prochaines fonctionnalités pourraient être :

* Amélioration des dessins
* Ajout d’éléments de décor
* Arrondir la ligne brisée

Conclusion et perspectives

Lors de cette première implémentation nous avons obtenu une ovale qui ne peut que se déplacer à la verticale en réponse à l’action de l’utilisateur sur la souris.

Les difficultés ont été :

* de comprendre par où commencer pour l’implémentation du **MouseListener** car auparavant nous n’avions jamais eu à faire avec cette fonctionnalité, pour résoudre ce problème nous avons regardé des exemples d’implémentation de contrôleurs
* de comprendre la question 5 de l’énoncé de l’exercice 6 qui demande de créer un objet de la classe contrôleur alors que pour faire fonctionner le programme on n’a pas besoin de l’utiliser, le problème n’a pas vraiment été résolu, on a passé en paramètre à la méthode **ajouteComposant** le composant affichage en passant par l’objet contrôleur crée.
* de faire en sorte que la ligne brisée ait une structure variée et soit compatible avec la déplacement de l’ovale. Ce problème a été résolu grâce aux différents attributs ajoutés à la classe **Parcours comme** : **vitesse** ou **montee**.
* de faire en sorte que les points choisis, par la méthode **testPerdu,** pour encadrer l’ovale soient corrects. Pour résoudre ce problème on a travaillé sur la condition qui permet de sélectionner ou non les points visibles au sein de la méthode **getParcours.**