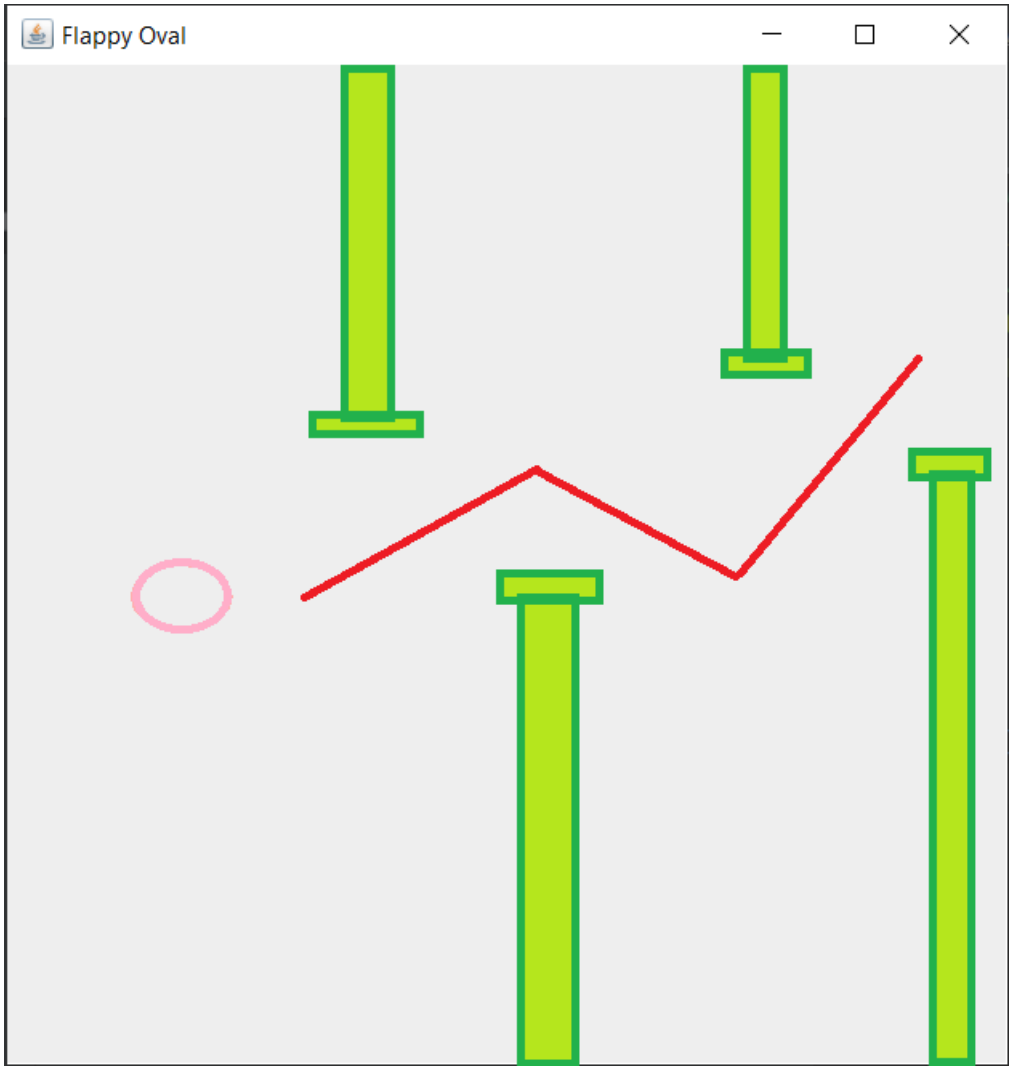
Rapport Flappy Bird

Thomas Delépine

January 2022

1 Introduction

Nous voulons créer une version du jeu mobile Flappy Bird. Pour cela nous modélisons l'oiseau par un oval qui se déplacera de bas en haut aux cliques du joueur puis redescendra de lui-même. Le but du joueur sera de se déplacer entre les tuyaux suivant une ligne imaginaire, l'interface graphique aura donc cette allure :



2 Analyse Globale

Pour implémenter une version simplifiée de Flappy Bird, il y a plusieurs fonctionnalitées à coder :

- affichage graphique de l'oval
  - l'oval a certaines dimensions
  - l'oval est situé à certaines coordonnées
- saut de l'oval aux cliques du joueur
  - un saut à une certaine amplitude
  - réaction instantannée du programme aux cliques du joueur
- chute constante de l'oval simulant un vole
  - une chute a une certaine amplitude
  - l'oiseau chute régulièrement
  - la fréquende de chute est constante
- génération d'un parcours
  - il est imoortant que le parcours doit être suivable par l'oiseau avec les amplitudes de saut et de chute
  - le parcours ne doit pas non plus être simplement linéaire
  - génération aléatoire
- affichage graphique de la ligne brisée
  - l'affichage doit être intuitif
- défilement de la ligne brisée
  - le défilement doit être fluide
  - sensation que l'oval avance
- génération continue d'un parcours
  - il est important que le parcours soit toujours jouable
  - le parcours ne doit pas non plus être simplement linéaire
  - génération aléatoire
- affichage d'un score
  - l'affichage du score doit être fluide
  - lisible
  - donner une bonne idée de l'avancée du jeu au joueur
- les affichages suivent les changements de taille de la fenêtre
  - sans créer de bug d'affichages
  - en gardant les bonnes proportions
  - sans ralentir le jeu
- implémentation des défaites
  - il est important que le jeu ne devienne pas trop dur au début
  - il est important que la partie se lance sans que le joeur perde direct
  - il est important que le jeu ne soit pas non-plus trop permissif
- implémentation d'une accélération de l'oval
  - il est important que l'accélération n'explose pas et que le jeu ne devienne plus jouable
  - l'accélération pourra suivre une courbe exponentielle
  - l'accélération au départ vaut 1
- construction d'un décors style Flappy Bird
  - affichage de tuyaux comme dans l'objectif de départ
- ajout d'oiseaux en mouvements
  - il ne faut pas que les oiseaux surchargent l'affichage
  - les oiseaux sont générés aléatoirement avec des positionnement et vi-

tesses aléatoires
- il est important que le mouvement des oiseaux soit fluide
- il est important que les oiseaux soient derrière le décor pour donner une sensation de profondeur
- les oiseaux pourront avoir une taille proportionnelle à leur vitesse pour un aspect plus réaliste encore

3 Plan de Développement

L'affichage graphique de l'oval ainsi que l'implémentation du saut nécessitent 2h. Pendant ces 2h, on va étudier les librairies nécessaires pour implémenter l'affichage graphique et l'intéraction avec la fenêtre. On va en même temps commencer à rédiger un rapport, ainsi que la documentation.

L'implémentation de la chute de l'oval va prendre 1h, avec l'étude du fonctionnement des threads, et la mise en place du code de la chute. Toujours accompagné de la documentation.

Pour généré la ligne brisée et l'affichée, il faut réflechir avant aux algorithme nécessaire, il y a donc 45 minutes de reflexions, et 45 minutes de codage.

Pour le défilement de la ligne brisée, il faut réfléchir un peu pour comprendre comment créer ce défilement. L'implémentation est ensuite assez rapide, donc 1h en tout.

La génération continue du parcours est assez facile car la première étape de génération de parcours à permis d'avoir les idées claires sur les algorithmes à suivre, donc 45 minutes. La documentation, elle, est assez riche, donc 45 minutes en plus.

L'affichage du score nécessite une petite reflexion pour savoir comment s'y prendre, puis l'implémentation est rapide, donc 30 minutes.

Faire en sorte que l'affichage suive les dimensions de la fenêtre n'est pas évident à bien faire. Il faut 30 minutes de reflexions, puis 15 minutes pour coder et 15 minutes de documentation donc 1h.

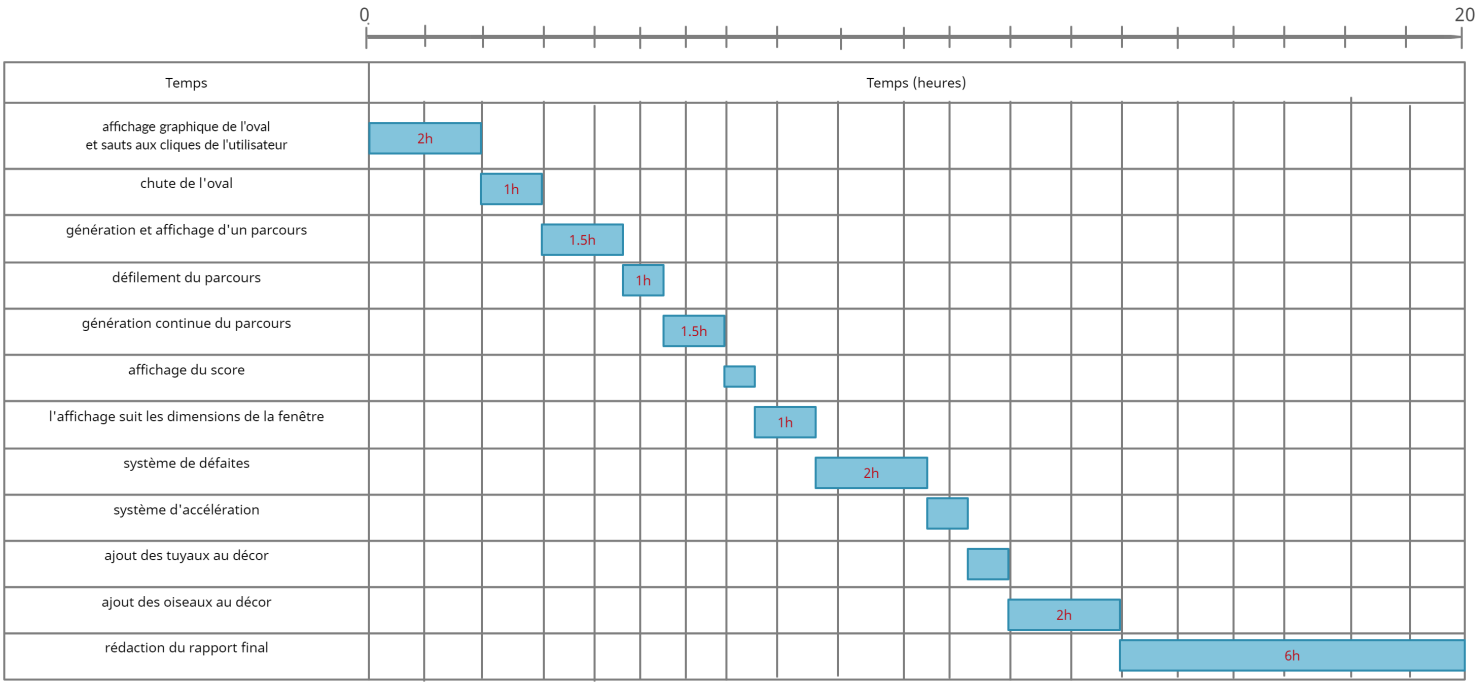
L'implémentation des défaites est l'une des plus grosses parties du projet. Il faut 45 minutes de reflexions, 45 minutes de codage et 30 minutes de documentation, soit 2h.

L'implémentation de l'accélération est très rapide, la reflexion est cependant plus importante, donc 45 minutes en tout.

L'ajout des tuyaux au décor est assez rapide, 45 minutes.

L'ajout des oiseaux au décor n'est pas facile, et assez fastidieux à coder, donc 2h.

La rédaction du rapport en Latex est assez longue, donc 6h.



4 Conception Générale

Nous avons adopté le motif MVC pour le développement de notre interface graphique. En Effet, la module Modèle de notre projet fait office de modèle en stockant les variables et constantes importantes et les méthodes les modifiants. Le module Vue correspond à la gestion de la vue en s'occuaent du contenue de la fenêtre ouverte par le programme. Enfin, le module Controleur correspond à la gestion des intéractions de l'utilisateur avec le programme, en faisant évoluer le modèle et en demandant au programme de mettre à jour l'affichage graphique.

Les fonctions d'affichage de l'oval, de la ligne brisée et du décor se trouvent dans le package Vue. La classe PanelFlappy centralise toutes les méthodes de création d'éléments à afficher. Dans ce package se trouvent aussi les classes représantant les oiseaux du décors, et la classe les animants.

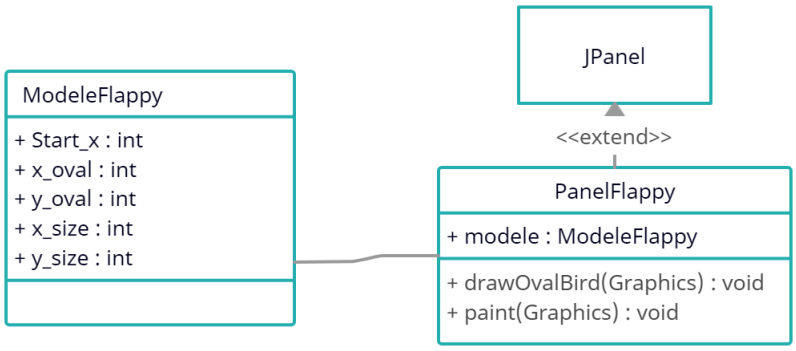
Les fonctions qui permettent à l'utilisateur d'intéragir avec le jeu sont dans la classe Controleur du package Controleur. Dans cette classe se trouve la méthode de réaction aux cliques du joueur qui se charge ensuite de faire sauter l'oval.

Les méthodes de modification du modèle se trouvent dans la classe Modele-Flappy du package Modele. On y trouve les méthodes modifiant les valeurs de positionnement de l'oval dans l'espace, la méthode de calcul de la vitesse de l'oval, la méthode de calcul de la position de l'oval par rapport au parcours pour tester si le joueur à perdu ou non. Dans ce même package se trouve la

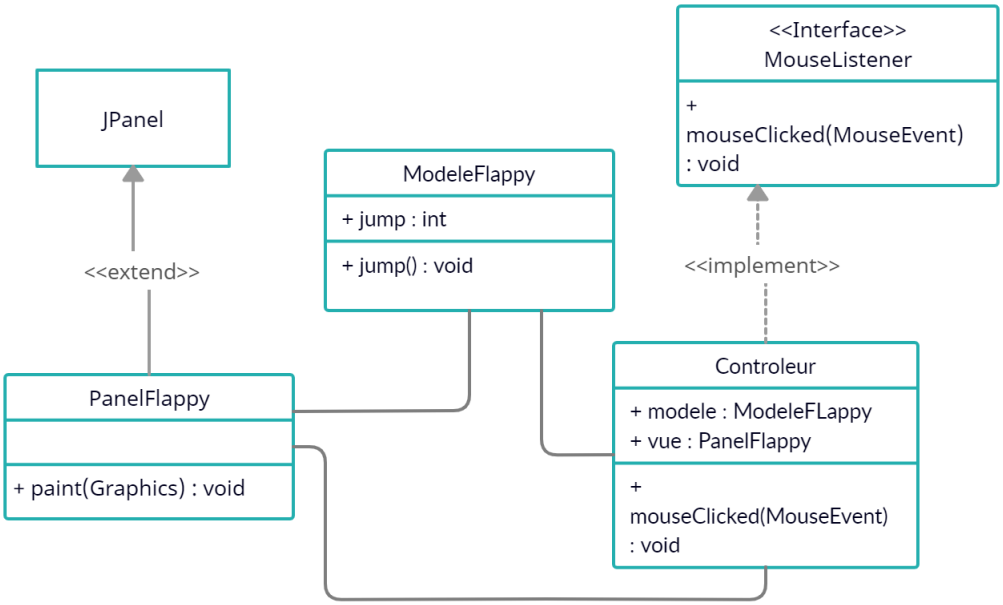
classe parcours qui génère un premier petit parcours, et qui possède une méthode permettant de générer une nouvelle étape du parcours à chaque appels. On y trouve aussi la classe Avancer gérant le décalage de tous les éléments à afficher au cours du temps. Enfin on y trouve la classe GameOver stoppant le jeu quand le joueur a perdu.

5 Conception Détaillée

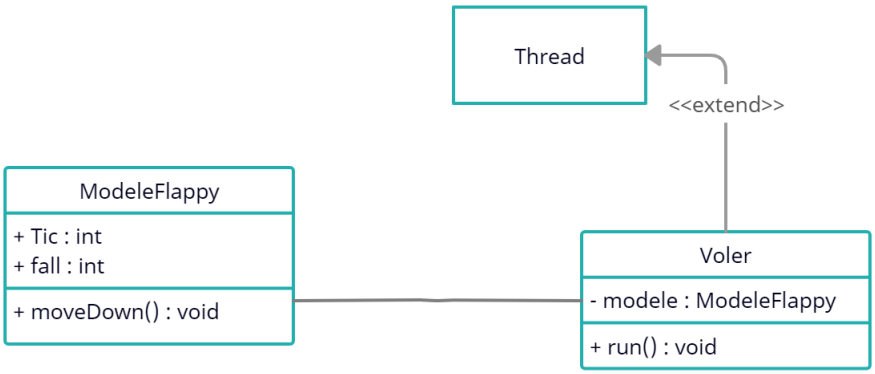
Premièrement, nous avons implémenter l'affichage de l'oval. Dans la classe ModeleFlappy se trouve une variable Start_x désignant la position de départ de l'oval. celle-ci est initialisée à 20% de la largeur de la fenêtre. Nous avons de même les variables x_oval et y_oval désignant la position de l'oval à tout instant. Ces variables sont initialisée respectivement à 20% de la largeur de la fenêtre (position de départ), et à la moitiée de la hauteur de la fenêtre. Enfin on trouve x_size et y_size désignant les dimensions de rectangle encadrant l'oval, et initialisée chacunes à 1/10 des dimensions de la fenêtre.
Ainsi, dans la classe PanelFlappy se trouve la méthode drawOvalBird qui met la couleur de l'environnement à rose, puis dessine un oval aux points x_oval et y_oval, et aux dimensions x_size et y_size. Cette méthode est enfin appelée dans la méthode paint de PanelFlappy.



On va ensuite vouloir que, aux cliques du joueur sur la fenêtre, l'oval monte d'un cran, simulant un saut. Pour se faire, on place dans la classe ModeleFlappy une variable jump, arbitrairement initialisée à 3/50 de la hauteur de la fenêtre. Dans la classe Controleur se trouve alors la méthode mouseClicked qui réagit lors d'un clique et qui fait appel à la méthode jump() de ModeleFLappy puis demande un repaint à la vue. Cette méthode fait monter de jump pixels l'oval si celui-ci se trouve dans la fenêtre après son saut. De cette manière, la hauteur de l'oval est mise à jour, et son affichage aussi.



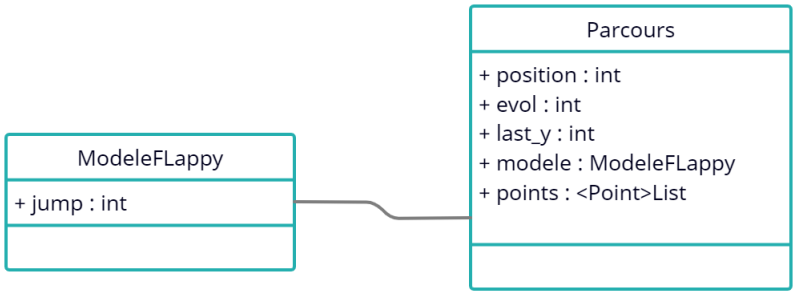
Afin de simuler le vol de l'oval, on va constamment le faire chuter. Pour cela, on va créer un thread dans la classe Voler qui va tourner tant que le jeu continue, et qui va constamment appeler la fonction moveDown de ModeleFLappy, sur le modele. Cette fonction diminue la hauteur de l'oval de la valeur de la variable fall, initialisée à 8/1000 de la fenêtre. Par ailleur, ce thread ne peux pas constamment appeler moveDown car le joueur ne pourrait humainement pas suivre pour garder l'oiseau dans la fenêtre. On fait alors des pauses de Tic millisecondes, ou Tic est une constantes définie dans le modèle, et de valeur arbitrairement 75.
A noter que Tic sera la mesure de base de tous les sleep dans tous les threads du projet, permettant la synchronisation.



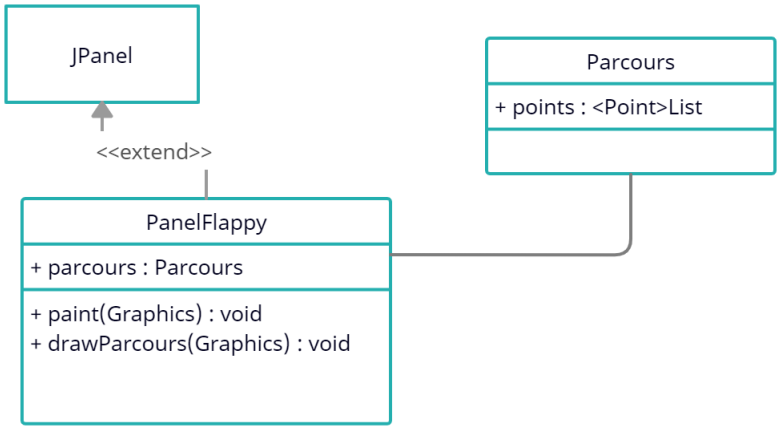
On va ensuite vouloir créer un parcours pour donner un but au jouer : suivre le parcours. Le parcours est alors créé dans le classe Parcours du package Modele. Cette classe possède un attribut evol qui contient le nombre de pixels horizontal entre chaque étape du parcours, un attribut position qui contient la coordonnée en x du dernier point généré, et l'attribut last_y qui contient la coordonnée en y du dernier point généré. Il possède aussi un attribut points qui est une liste de point : les points du parcours. Lors de la construction d'un objet Parcours, l'attribut points est alors compléter afin de créer ce qui sera un parcours suivant l'algorithme suivant :

```
1  points ← [] ;
2  evol ← un tier de la largeur de la fenêtre  ;
3  p_1 ← (x_oval, y_oval);
4  points ← [p_1];
5  position ← x_oval + evol;
6  p_2 ← (position, y_oval + y_size);
7  points ← [p_1, p_2];
8  last_y ← p_2.y;
9  tant que position < largeur de la fenêtre + 3 * evol faire
10 |  position ← position + evol ;
11 |  randy ← last_y ± 3 * jump;
12 |  if randy < (hauteur de la fenêtre)/8 then
13 |  |  randy ← (hauteur de la fenêtre)/4 + randy
14 |  end
15 |  if randy > 7*(hauteur de la fenêtre)/8 then
16 |  |  randy ← 7*(hauteur de la fenêtre)/4 - randy
17 |  end
18 |  points ← points ∪ {(position, randy)};
19 |  last_y = randy;
20 fin
```

De cette manière, on créé deux premiers points de telle sorte que le début du jeu ne mène pas à une défaite instantanée du joueur. Ensuite, on rajoute un nombre fini de points dont la hauteur de chacun est tirée aléatoirement dans une fenêtre réduite autour de la hauteur du point précédent. Cela forme un parcours simple, et qui permet au joueur de le suivre malgrès les contraintes de la valeur de fall, et de la valeur de jump.



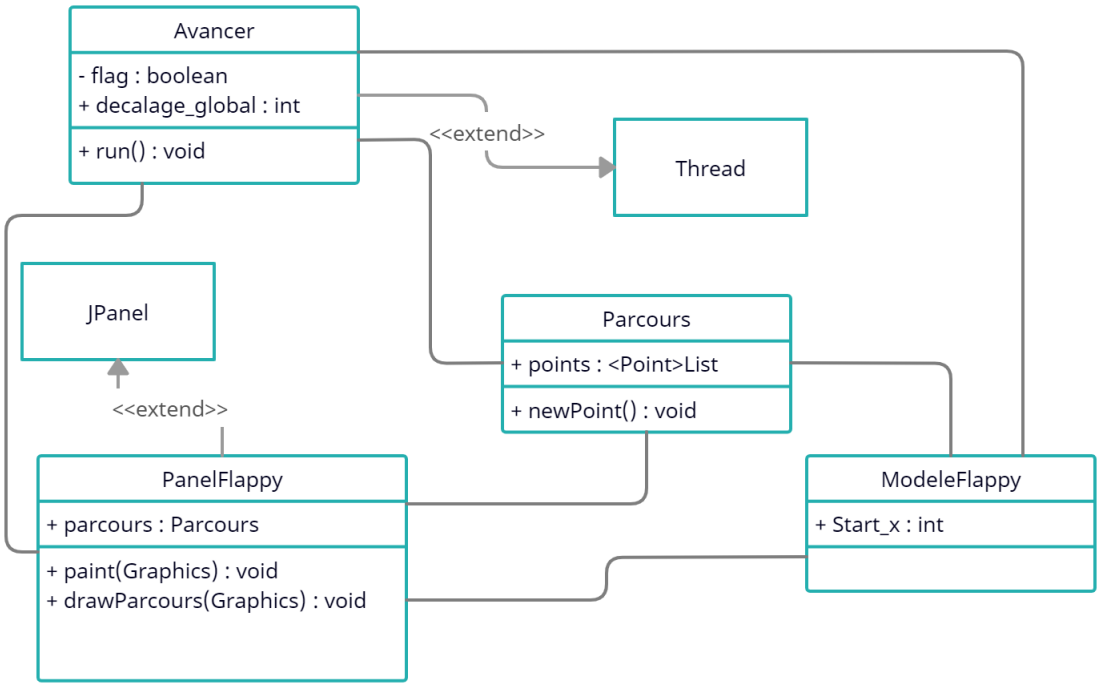
On veut alors afficher le parcours qui est donc une ligne brisée. Pour ce faire, on utilise la méthode drawParcours dans laquelle on parcourt simplement la liste des points générée juste avant, on trace une ligne entre un point et son prédécesseur. Cette fonction sera enfin appelée dans la méthode paint de PanelFlappy.



On souhaite ensuite créer un effet d'avancement de l'oval, tout en laissant l'oval immobile sur les x. Pour cela, on fait défiler le fond, et donc la ligne brisée. Pour ce faire, on crée la classe Avancer, dans laquelle un thread va tous les Tic millisecondes incrémenter une variable : decalage_global. Il suffira alors, lors de l'affichage de la ligne brisée, de décrémenter toutes les coordonnées x des points de la valeur de décalage_global. Cela décale au fur et a mesure le parcours vers la gauche, jusquà ce qu'il sorte de l'écran. On souhaite alors deux choses : toujours voir un parcours à l'écran, et supprimer de la mémoire les points du parcours déjà sortis de l'écran. Pour cela, on ajoute la méthode newPoint dans la classe parcours dont le code est équivalent à une itération de la boucle de l'algorithme présenté précédemment. L'algorithme de suppression/création de points est le suivant :

```
1  flag ← true ;
2  decalage_global ← 0;
3  tant que le jeu continue faire
4  |  decalage_global = decalage_global + 1;
5  |  Sleep(Tic/3);
6  |  if points[1] − decalage_global ≤ Start_x ∧ flag then
7  |  |  newPoint();
8  |  |  flag ← false;
9  |  end
10 |  if points[1] − decalage_global ≤ 0 ∧ ¬flag then
11 |  |  points.remove(0);
12 |  |  flag ← true;
13 |  end
14 fin
```

De cette manière, à chaque tour de boucle, le décalage est augmenté d'une unité. ensuite, soit a lieu l'ajout d'un point, soit a lieu la suppression d'un point. On garde alors toujours un nombre suffisant de points pour afficher un parcours complet à l'image, et la taille du parcours est à la fois minorée et majorée. Ainsi, on décale constamment l'affichage du parcours, et on génère constamment de nouvelles étapes du parcours, permettant un bon affichage de ce dernier.



Pour afficher un score, on modifie simplement le titre de la fenêtre avec la

valeur de decalage_global, de la forme "FlappyOval : n", où n est la valeur de decalage_global. Le changement du titre de la fenêtre se fait avec la méthode setTitle de la classe JFrame. Ce choix à été fait car il est plus original que l'affichage d'un bloc de texte dans la fenêtre, et aussi plus simple et jolie. Le tout étant étonnament fluide et visible, cette option a été gardée.

L'affichage est désormais assez riche, et le joueur peut vouloir modifier les dimensions de la fenêtre. Jusqu'alors, quand il y avait une référence faite à la hauteur ou à la largeur de la fenêtre, il s'agissant en réalité des dimensions de la fenêtre dans le modèle. Ces dimensions étant juste logique, on crée deux méthodes dans la classe PanelFlappy : scaleToX et scaleToY permettant de calculer l'équivalent des coordonnées x et y logiques par rapport à la fenêtre. Le code de ces fonction est le suivant :

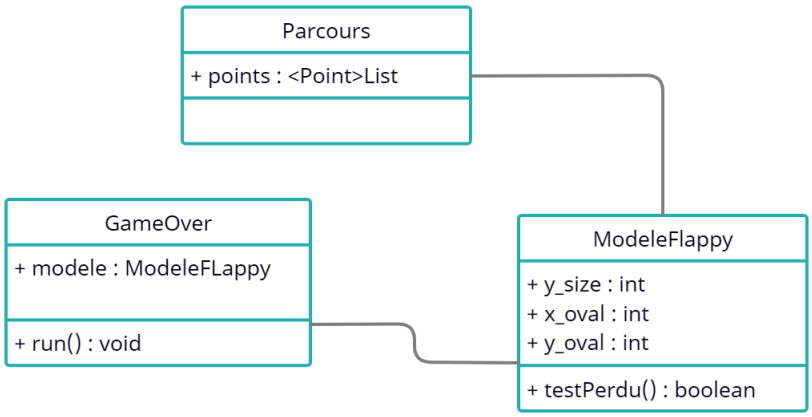
```
public int scaleToX(int x){
   return (int) (x/((double)
       this.modele.width)*this.modele.fenetre.getWidth());
   }
public int scaleToY(int y){
   return (int) (y/((double)
       this.modele.height)*this.modele.fenetre.getHeight());
   }
```

où width et height sont des variables de la classe ModeleFlappy. Il s'agit alors d'une règle de trois. Ces deux fonctions sont ensuite appelées à chaque calcul de coordonnées/distances qui seront ensuite dessinées dans la fenêtre, à la bonne echelle. Cela vaut pour tous les éléments de la fenêtre.

Le jeu est alors presque jouable : il ne lui manque qu'un but, ne pas perdre, il faut donc implémenter un système de game over. Pour cela, on ajoute à la classe ModeleFlappy un attribut theGameIsAlive qui vaut true. On créé ensuite une classe GameOver dans laquelle un thread observe constamment la position de l'oval par rapport au parcours. Cela se fait par un appel à la méthode testPerdu de la classe ModeleFlappy. TestPerdu repère une situation de défaite du joueur et modifie alors la valeur de theGameIsAlive. L'algorithme suivie par la méthode testPerdu est le suivant :

1 $x \leftarrow$ coordonnée en x du milieu du rectangle entourant l'oval;
2 $y \leftarrow$ coordonnée en y du milieu du rectangle entourant l'oval;
3 $cpt \leftarrow$ le rang du dernier point avant l'oval en abscisse;
4 $Y \leftarrow points[cpt].y + (x - points[cpt].x) \times \frac{points[cpt+1].y - points[cpt].y}{points[cpt+1].x - points[cpt].x}$;
5 $\epsilon =$ (hauteur de la fenêtre)/20;
6 $\rightarrow \neg((Y > (y - y\_size/2 - \epsilon)) \wedge ((Y < (y + y\_size/2 + \epsilon)))$;

On calcul la distance entre le centre de l'oval et le parcours, et on s'accorde une marge d'erreur de 5% de la hauteur de la fenêtre (de la vraie fenêtre ici).



Pour rendre le jeu un peu plus difficile, on implémente un système d'accélération au cours du temps. Il s'agira d'un coefficient qui aggrandira les mesures du jeu. Ce coefficient va grandir à mesure que decalage_global grandit. On veut que cette accélération ait l'allure du fonction exponentielle. Cependant, pour que le jeu reste jouable, il faut que cette fonction croit très lentement. On pourra par exemple dire que quand decalage_global atteint 1000, l'accélération double. On choisit donc la fonction suivante :

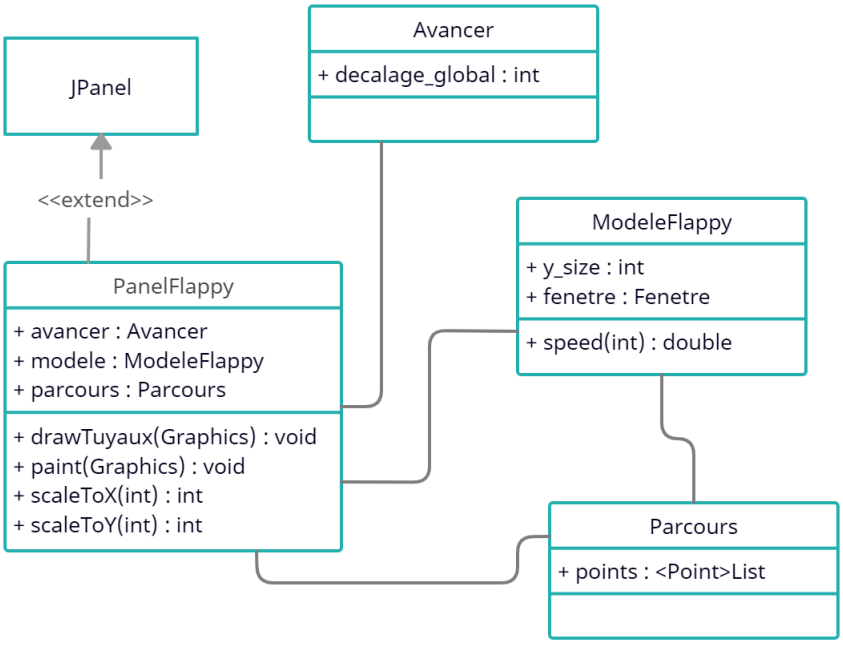
$$f(n) = e^{n*ln(2)/1000}$$

On ajoute alors à ModeleFlappy la méthode speed, prenant en entrée un entier, et renvoyant le résultat de f appliquée à cet entier. Cette vitesse multiplie a chaque fois decalage_global, ce qui fait un effet d'accélération du défilement du parcours. le score est d'ailleur multiplié par la vitesse. On multiplie aussi la valeur de fall par la vitesse afin de rajouter de la difficulté au fur et à mesure du jeu. Au final, le code de drawParcours de la classe PanelFlappy devient par exemple :

```
private void drawParcours(Graphics g){
     g.setColor(new Color(244, 0, 44));
     Graphics2D g2d = (Graphics2D) g;
     Point p = this.parcours.points.get(0);
     double deca =
         this.avancer.decalage_global*this.modele.speed(this.avancer.decalage_global);
     for(Point e : this.parcours.points){
        int x1 = scaleToX((int) (p.x - deca));
        int x2 = scaleToX((int) (e.x - deca));
        int y1 = scaleToY(p.y);
        int y2 = scaleToY(e.y );
        g2d.setStroke(new BasicStroke(3L));
        g2d.drawLine(x1, y1, x2, y2);
        p = e;
     }}
```

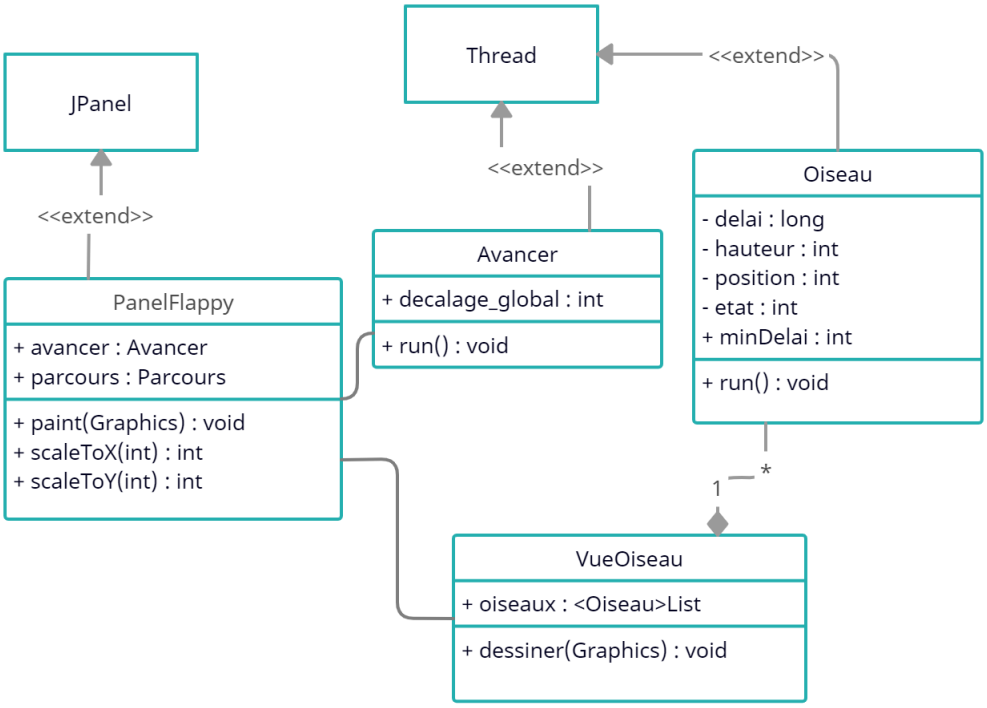
On voit alors l'utilisation systématique de scaleToX et scaleToY, ainsi que l'utilisation de la méthode speed qui multipliée par decalage_globale permet de calculer le nouveau décalage réel : deca..

Le but étant de recréer une version simplifiée de Flappy Bird, on rajoute les fameux tuyaux. Pour cela, on créé la fonction drawTuyaux de PanelFlappy dans laquelle on dessine un tuyau en haut et en bas de chaque points du parcours. On dessine une première forme avec des rectangles vert clair, puis par dessus on dessine des contours de rectangles vert foncé. Chaque rectangle est centré par rapport aux points du parcours, à une distance constante. Lors de l'affichage, on applique les fonctions scaleToX et scaleToY pour que les tuyaux suivent la taille de la fenêtre, et on les décale de la même façon qu'on décalait le parcours. Cette fonction est alors appelée dans la fonction paint de PanelFlappy.



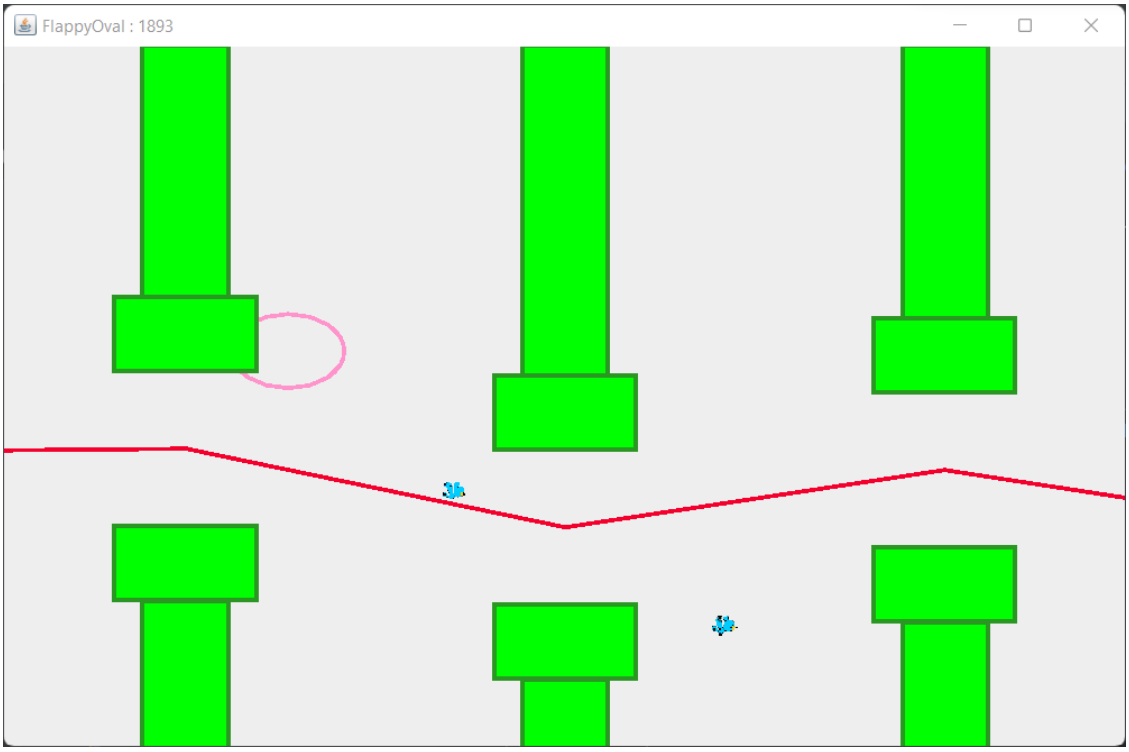
Enfin, on veut ajouter des oiseaux en mouvement dans le décor. Pour cela, on créé la classe Oiseau qui stocke les informations d'un oiseau. On créé la classe VueOiseau qui stocke tous les oiseaux créés dans une liste et qui possède une méthode dessiner, méthode appelée dans paint de PanelFlappy.

- Un oiseau est défini par sa position dans l'espace, un délai de mise à jour, et un etat représantant sa position de vol
- VueOiseau possède un attribut oiseaux de type <Oiseau>List
- lors de la création d'un oiseau, sa position est générée aléatoirement en y, et positionné en x à droite de la fenêtre
- son délai de mise à jour est généré aléatoirement entre minDelai et 3× minDelai
- la hauteur de l'oiseau est un nombre aléatoire généré en fonction des dimensions de la fenêtre, et dans la fenêtre
- lors de l'affichage de l'oiseau, on redéfinie ses dimensions en fonction de son délai de telle sorte qu'un oiseau rapide sera grand et paraitra proche, et un oiseau lent sera petit, et paraîtra éloigné.
- on charge alors une image correspondant à l'etat de l'oiseau, il y a 8 images et cela donne à l'oiseau une animation de vole réaliste
- l'état de l'oiseau est constamment mis à jour dans un thread dans la classe oiseau. A chaque oiseau il correspond donc un thread.
- chaque oiseau met à jour lui même sa position dans ce même thread
- l'affichage de l'oiseau dans drawOiseau les affiche en modifiant leur dimensions et positions en fonction des dimensions de la fenêtre
- La méthode d'affichage des oiseaux est enfin appelée dans paint de PanelFlappy les affichants à l'écran
- un oiseau est généré aléatoirement dans le thread avancer, suivant une probabilité assez basse de telle sorte qu'il y ait en moyenne 4 oiseaux par fenêtre
- à noter que l'appel de la méthode dessiner de VueOiseau se fait en premier dans la méthode paint, afin que ceux-si soient dessinés derrière le parcours, l'oval et les tuyaux, accentuant l'effet de profondeur déjà créé par les différence de vitesse et de dimensions.


Thread
<<extend>>
JPanel
<<extend>>
<<extend>>
Oiseau
- delai : long
- hauteur : int
- position : int
- etat : int
+ minDelai : int
+ run() : void
Avancer
+ decalage_global : int
+ run() : void
PanelFlappy
+ avancer : Avancer
+ parcours : Parcours
+ paint(Graphics) : void
+ scaleToX(int) : int
+ scaleToY(int) : int
1
*
VueOiseau
+ oiseaux : <Oiseau>List
+ dessiner(Graphics) : void

6 Résultat

Le résultat actuel est le suivant :



7 Documentation utilisateur

— Prérequis : Java avec un IDE
— Mode d'emploi : 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.

8 Documentation développeur

On utilise le méthode MVC pour ce projet, chaque partie étant dans un package à son nom. Le programme se lance sur la classe Main du package Main. C'est dans la méthode main de cette classe que tous les objets sont créés. Les deux classes les plus importantes sont ModeleFlappy du package Modele, et PanelFlappy du package Vue. Dans la première classe, les variables importantes sont définie et commentées. C'est le coeur du jeu, on y trouve les méthodes de modification du coeur de jeu. Il y a par example les dimensions de la fenêtre au lancer du jeu, la valeur du temps d'attente de base pour chaque thread, les dimensions de base de l'oval. Toutes ces valeurs pourraient être modifié afin d'accélérer le jeu, le rendre plus agréable pour l'utilisateur.
On y trouve aussi la méthode speed, qui est une fonction mathématique. Cette

fonction peut tout à fait être modifiée pour changer l'allure de l'accélération, toujours dans une optique d'améliorer la jouabilité.
Dans PanelFlappy du package Vue se trouve toutes les méthodes d'affichage des éléments de la fenêtre. Toutes ces méthodes pourront toujours être améliorée, enrichie afin de rendre le jeu toujours plus beau.

Un point important du jeu est le calcul des défaites. cela se fait dans la méthode testPerdu de la classe ModeleFlappy du package Modele, et est détaillée précédemment. Cette méthode calcul la distance au parcours pour estimer une défaite ou non du joueur. Cela modélise assez bien les défaites dans le jeu Flappy Bird, mais reste assez primitif. Un gros point d'amélioration serait de faire perdre le joueur comme dans Flappy Bird, lors d'une sortie de la fenêtre, ou lors d'une collision avec un tuyau.

9 Conclusion

Le jeu a donc atteint l'objectif attendu. Il est cependant toujours possible d'améliorer la jouabilité, de rendre le jeu plus beau avec un fond plus riche, un oiseau et non-plus un oval, une amélioration de l'affichage du score, un écran de fin, on écran de début avec un rappel du meilleur score et un bouton play. Il existe encore beaucoup d'idées pour améliorer le jeu qui est tout de même déjà intéressant. En effet, la création du jeu a permis de voir l'utilisation des affichages graphiques, l'utilisation des thread, et un peu de synchronisation. Tout cela est très enrichissant.