université
PARIS-SACLAY

Rapport de projet PCII

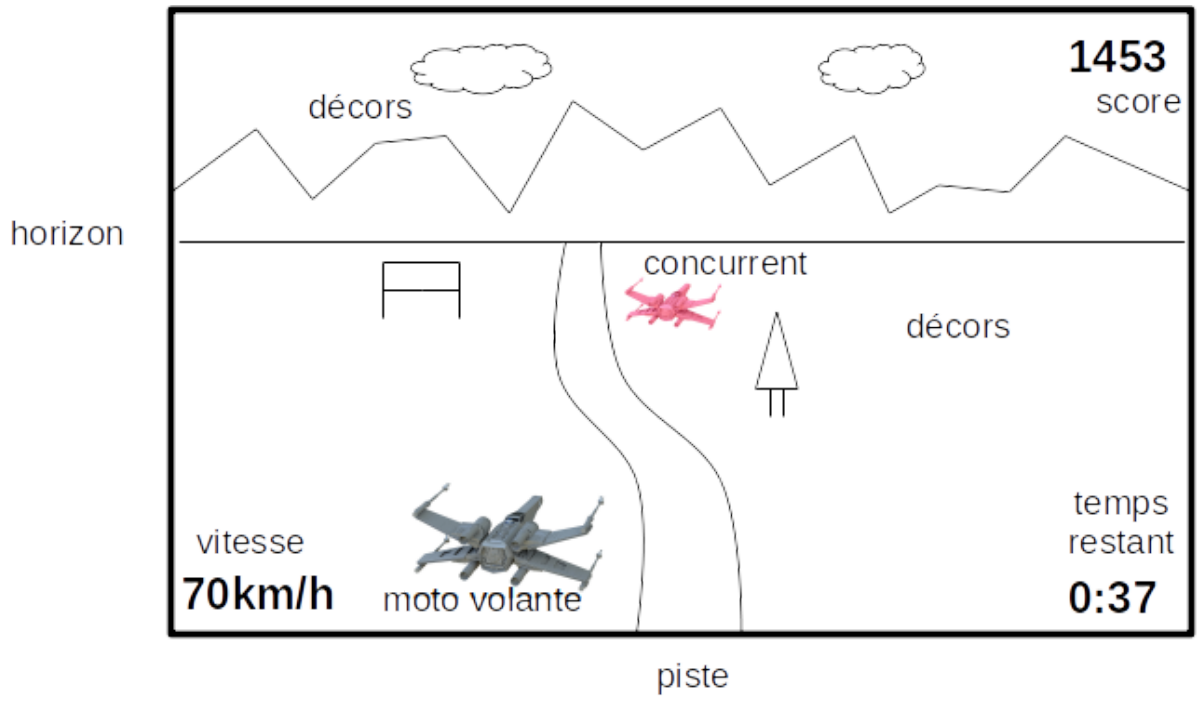
Réalisé par
Hongyu YAN, Shiqing HUANG

Sous la direction de
Thi Thuong Huyen Nguyen

Année universitaire 2020-2021
Licence Informatique

Introduction

L’objectif du projet est de réaliser un jeu vidéo des années 80 de type «course de voiture» en vue à la première personne. Le jeu permet au joueur de rouler sur une moto sur coussin d’air qui peut se déplacer horizontalement et voler vers le haut pour contrôler la vitesse, éviter les obstacles et surpasser ses concurrents. À la fin de la partie, nous donnerons le score du joueur et le meilleur score de l'histoire. La figure suivante montre un diagramme schématique du jeu:



Analyse globale

Pour réaliser le jeu, plusieurs fonctionnalités principales sont attendues:

1. l’interface graphique avec le véhicule, l' horizon et la piste
2. le défilement automatique de la piste
3. l’apparition des points de contrôles à intervalles réguliers
4. la réaction de le véhicule aux clavier de l’utilisateur
5. le mécanisme de calcul de l’accélération du véhicule en fonction de la position par rapport à la piste
6. l’affichage le temps restant et le kilométrage

Ensuite plusieurs fonctionnalités supplémentaires:

1. Courbe de Bézier
2. Ajout de concurrents et un bonus de 10 points au score pour chaque concurrent dépassé.
3. Le véhicule peut monter à l’aide de la touche « haut », il s’éloigne de la piste et va donc perdre de la vitesse. Il redescend tout seul vers le sol lorsqu’il perd de la vitesse.
4. Ajout d’un écran d’accueil
5. Mémorisation des meilleurs scores
6. Le dessin du véhicule change en fonction des actions du joueur

Ils seront divisés en 4 parties pour réaliser :

Nous nous intéressons d'abord uniquement à un sous-ensemble de fonctionnalités qui sont prioritaires et simples à réaliser:

- Création d'une fenêtre dans laquelle est dessiné le véhicule, l' horizon et la piste
- Déplacement du véhicule à droite et à gauche lorsqu'on tape les flèches du clavier

Nous nous consacrons ensuite à certaines fonctions qui sont plus difficiles et nous demande de maîtriser les connaissances multi-thread:

- Animation de la piste à vitesse constante
- Création du décors
- Mouvements du décors de fond selon les touches du clavier
- Décompte des kilomètres et affichage

Et puis, nous voulons réaliser les fonctionnalités qui sont compliqué et nous oblige à maîtriser certaines connaissances en physique et en mathématiques:

- Ajout d'obstacles et détection de collisions qui ralentissent le véhicule
- Ajout de points de contrôles (représentés par une ligne horizontale sur la piste)
- Calcul de l'accélération du véhicule en fonction de la position par rapport à la piste
- Lorsque la vitesse du joueur atteint 0 ou lorsque le temps alloué atteint 0, la partie se termine et le score est affiché

Enfin, nous voulons améliorer les fonctionnalités du jeu et l'interface :

Ces améliorations ne sont pas prioritaires et sont optionnelles, mais elles sont beaucoup plus difficiles et intéressantes.

- Ajout d'images dans le décors
- Ajout d'un écran d'accueil
- Ajout d'adversaires
- Création d'une sensation de profondeur
- Documentation du projet v1

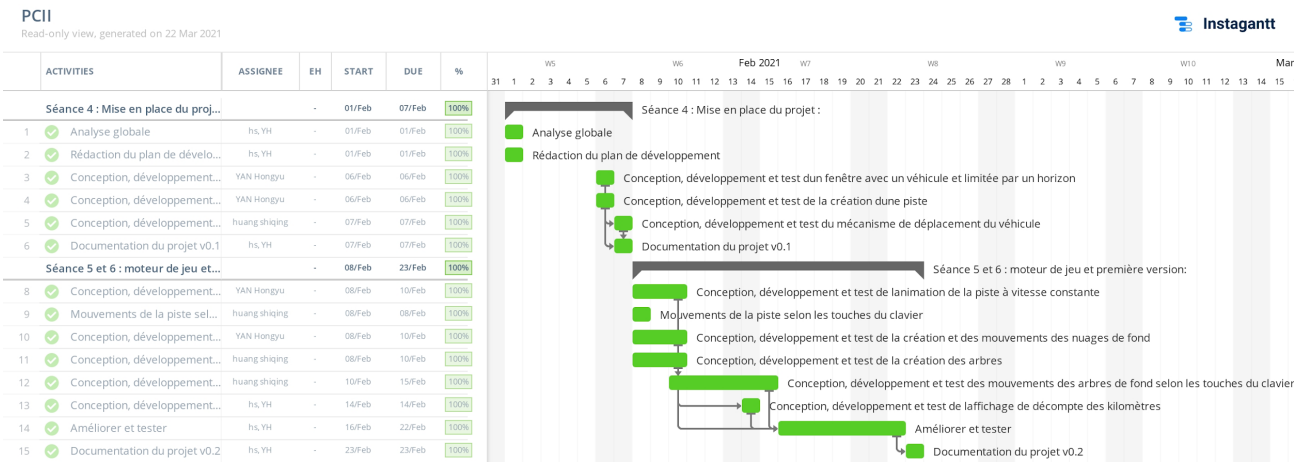
Plan de développement

- Séance 4 : Mise en place du projet

Tâche	assignée
Analyse globale	YAN, HUANG
Rédaction du plan de développement	YAN, HUANG
Conception, développement et test d'un fenêtre avec un véhicule et limitée par un horizon	YAN
Conception, développement et test de la création d'une piste	YAN
Conception, développement et test du mécanisme de déplacement du véhicule	HUANG
Documentation du projet v0.1	YAN, HUANG

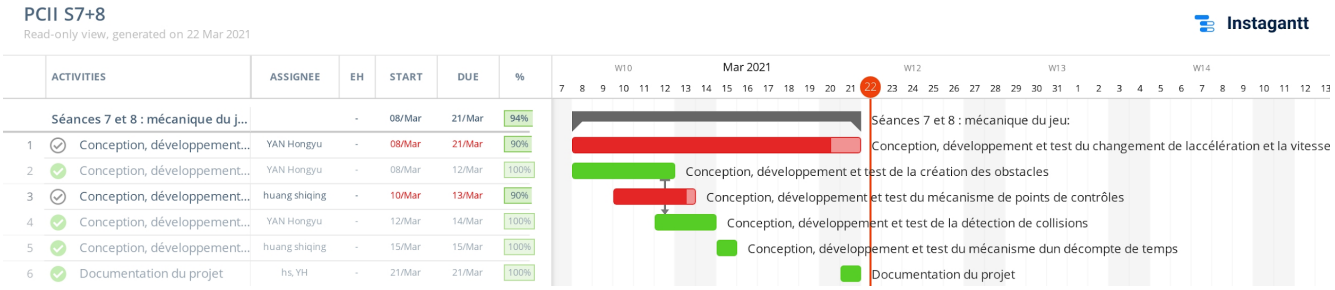
- Séance 5 et 6 : moteur de jeu et première version

Tâche	assignée
Conception, développement et test de l'animation de la piste à vitesse constante	YAN
Mouvements de la piste selon les touches du clavier	HUANG
Conception, développement et test de l'affichage de décompte des kilomètre	YAN, HUANG
Conception, développement et test de la création et des mouvements des nuages de fond	YAN
Conception, développement et test de la création des arbres	HUANG
Conception, développement et test des mouvements des arbres de fond selon les touches du clavier	HUANG
Améliorer et tester	YAN, HUANG
Documentation du projet v0.2	YAN, HUANG



- Séances 7 et 8 : mécanique du jeu

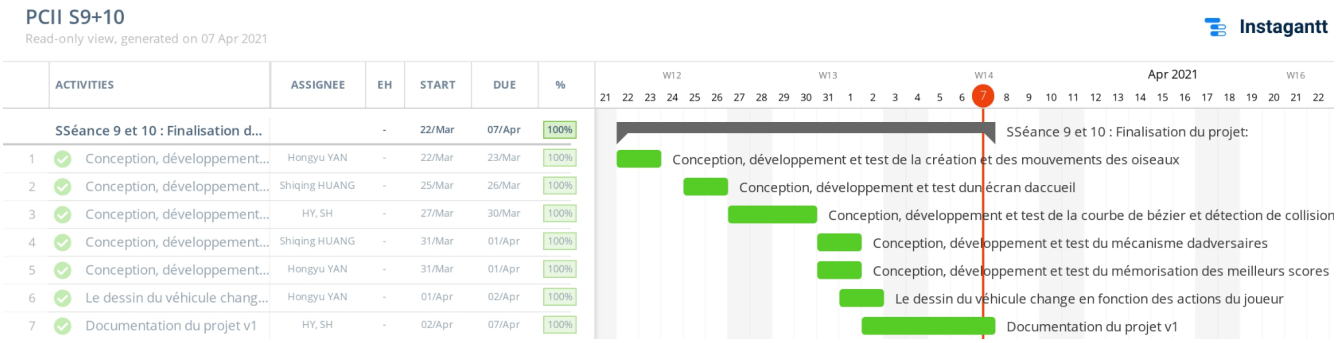
Tâche	assignée
Conception, développement et test du changement de l'accélération et la vitesse	YAN
Conception, développement et test de la création des obstacles	YAN
Conception, développement et test de la détection de collisions	YAN
Conception, développement et test du mécanisme de points de contrôles	HUANG
Conception, développement et test du mécanisme d'un décompte de temps	HUANG
Documentation du projet v0.9	YAN, HUANG



(Dans la séance 9 et 10, nous avons terminé le travail qui restait auparavant.)

- Séance 9 et 10 : Finalisation du projet

Tâche	assignée
Conception, développement et test de la création et des mouvements des oiseaux	YAN
Conception, développement et test d'un écran d'accueil	HUANG
Conception, développement et test de la courbe de bézier et détection de collision	YAN, HUANG
Conception, développement et test du mécanisme d'adversaires	HUANG
Conception, développement et test du mémorisation des meilleurs scores	YAN
Le dessin du véhicule change en fonction des actions du joueur	YAN
Documentation du projet v1	YAN, HUANG

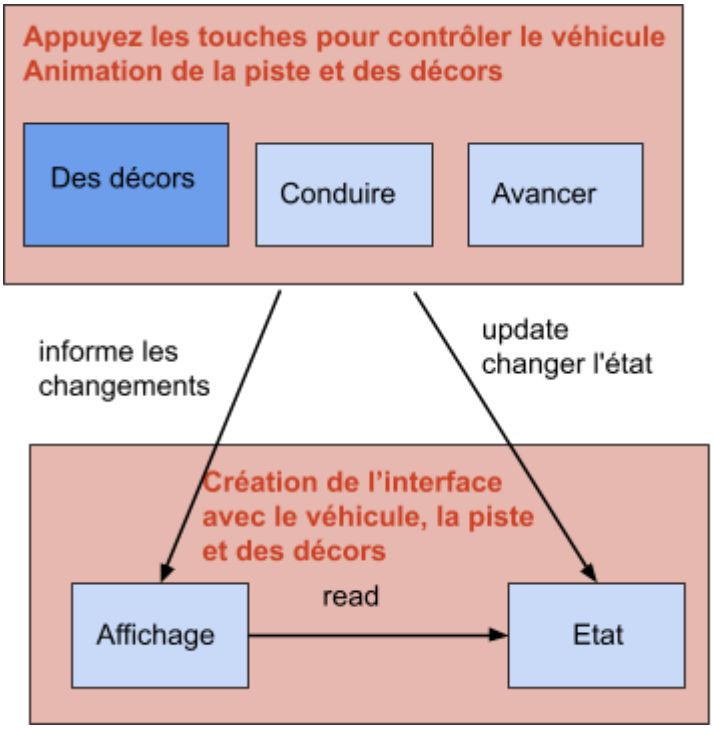


Conception générale

L'interface graphique s'est construite autour du modèle MVC. Le package *vue* s'occupe de dessiner l'interface, c'est a dire, une fenêtre avec du véhicule, une piste et des décors dedans.

Le package *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.

Le package *control* effectue les changements dans l'état et informe la *vue* des changements. Il modifie les états des dessins afin de faire déplacer le véhicule, avancer la piste et les décors, puis ils informent la vue des changements pour redessiner.



Conception détaillée

Pour la fenêtre avec une voiture, j'utilise l'API Swing et la classe JPanel. Je définis les dimensions de la voiture et de la fenêtre dans des constantes.

Constante(s) de la classe *Affichage*:

- LARG : Largeur de l'interface
- HAUT : Hauteur de l'interface
- WIDTH : Longeur du véhicule
- HEIGHT : Largeur du véhicule
- HAUT_VEH : Hauteur du véhicule
- HORIZON : Hauteur de l'horizon

Voici un diagramme de classe pour afficher une fenêtre avec un véhicule dedans.

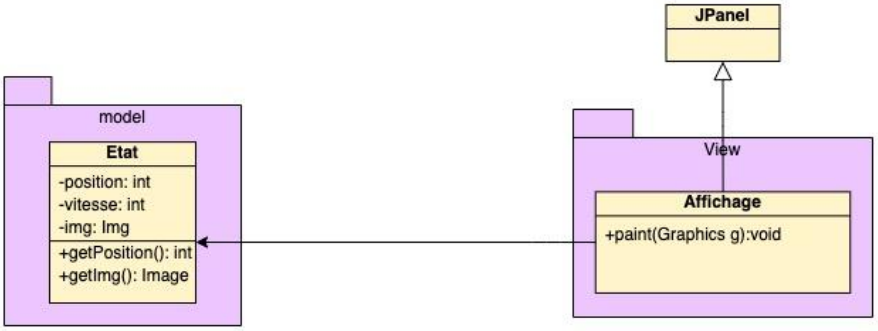


Figure-1 du afficher une fenêtre avec un véhicule

Pour le déplacement du véhicule, nous utilisons la programmation événementielle avec la classe KeyListener et la distance est définie dans une constante.

Constante(s) de la classe *Etat* :

- DEPLACE : La valeur du déplacement de quelques pixels du véhicule
- V_MAX : Vitesse maximum du véhicule
- Y_VEH : L'ordonné initial du vhécule
- Y_ciel : La distance maximale vers le ciel

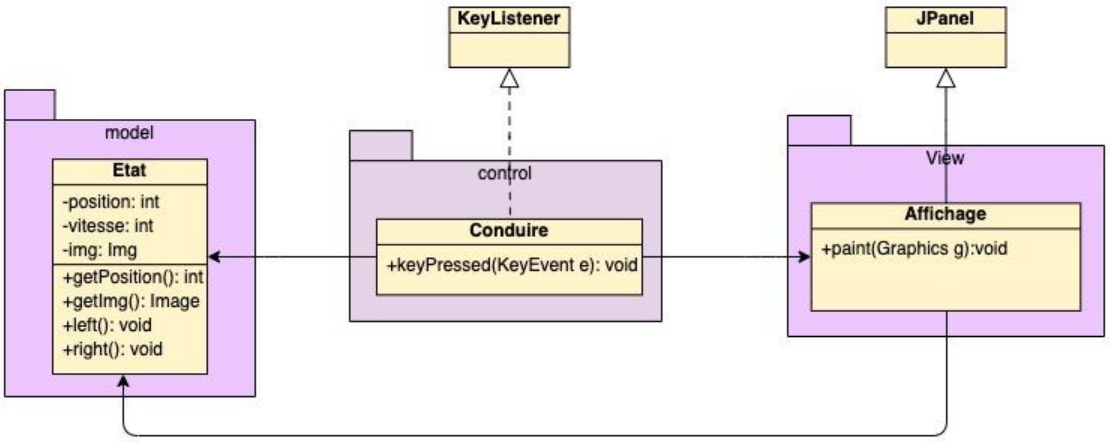


Figure-2 du déplacement du véhicule

Pour le défilement automatique de la piste, je dois d'abord générer aléatoirement les coordonnées et j'ai proposé un algorithme.

```
piste()  : void

    x <- new Random().nextInt(50) + Affichage.LARG/2 - 50

    y <- Affichage.HAUT

    Ajouter Point(x,y) dans la liste des points

    tant que  y >= Affichage.HORIZON faire

        x <- new Random().nextInt(50) + Affichage.LARG/2 - 50

        y <- y - new Random().nextInt(30) + 30;

        Ajouter Point(x,y) dans la liste des points

    fin tant que
```

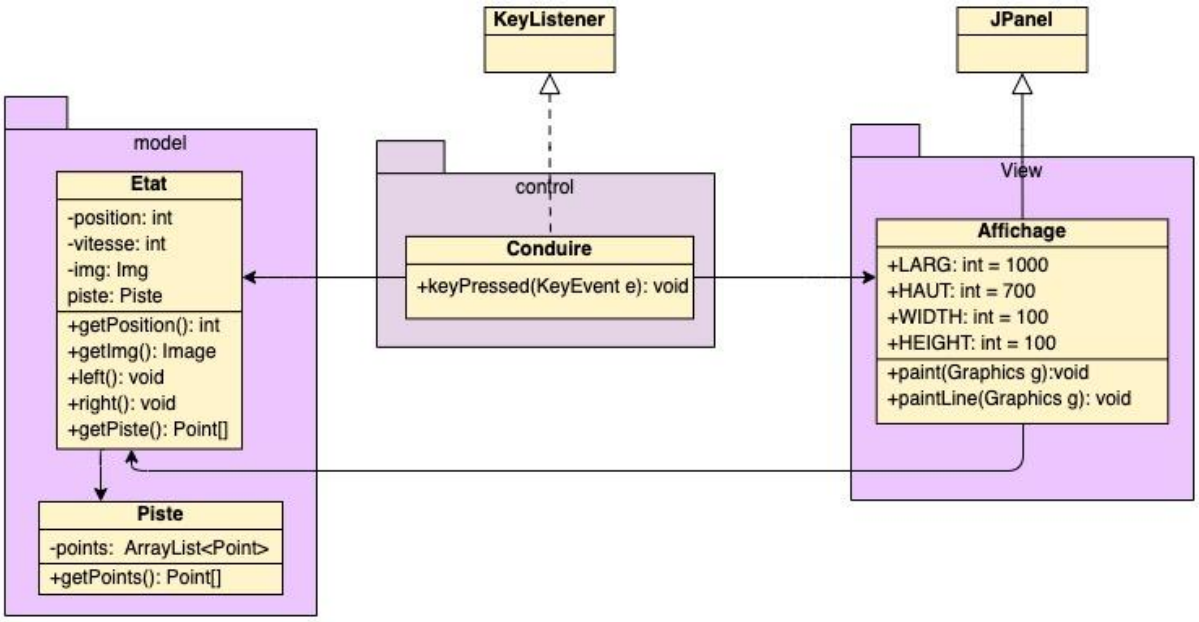


Figure-3-1 du défilement automatique de la piste

Ensuite j’ai besoin d’un autre thread pour avancer la piste, elle devrait être infinie et calculée à partir d’une ligne brisée verticale limitée par l’horizon et générée aléatoirement. Et la valeur du mouvement est définie dans une constante.

Constantes de la classe *Piste* :

- AVANCE : la valeur de l’incrément de la position de la piste

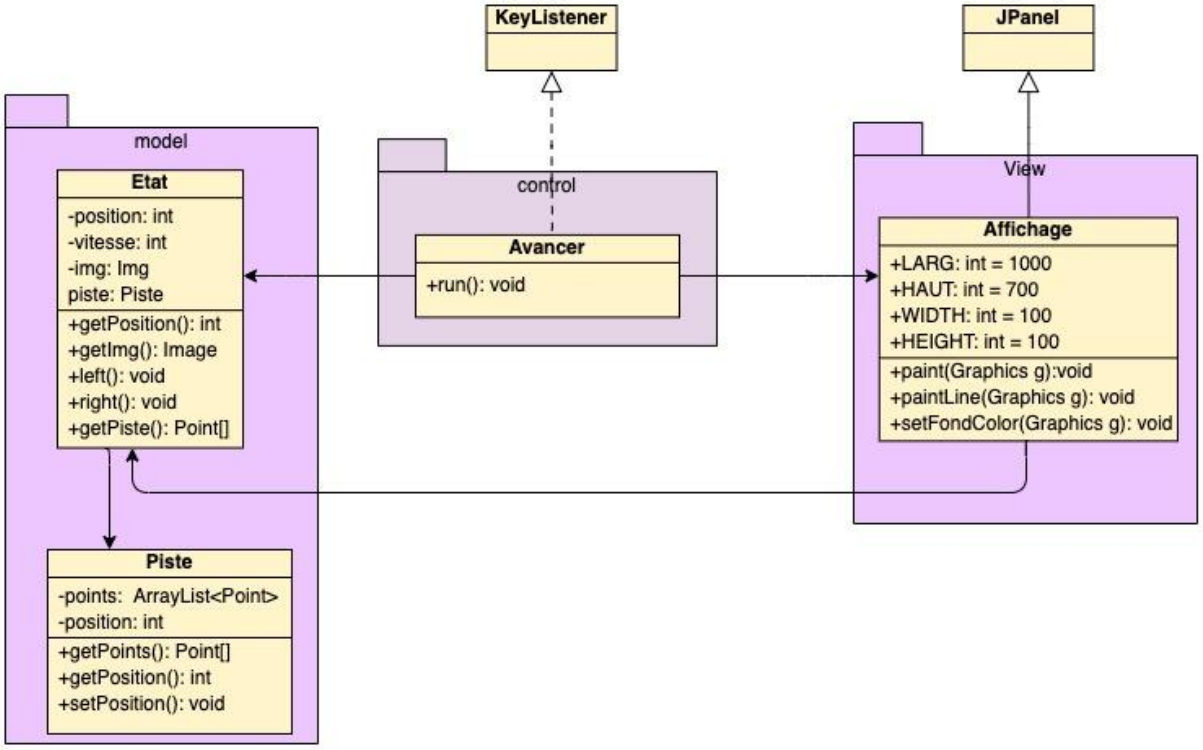


Figure-3-2 du défilement automatique de la piste

Nous avons un attribut *position* dans la classe *Piste* afin de savoir la kilomètre et on l'affiche dans la classe Affichage en utilisant la fonction *drawString.*

Maintenant, il est temps de rejoindre le décor. Afin d'améliorer la qualité de l'affichage, nous utilisons des images plutôt que des dessins faits à la main. On télécharge une image du nuage dans le projet. Nous créons ensuite une classe *Nuage* qui implémente un *thread*. La vue du MVC doit alors afficher les nuages qui se déplacent dans le ciel, de droite à gauche. Le constructeur de la classe *Nuage* choisit une valeur aléatoire pour le délai et la hauteur pour que les nuages aient des vitesses et des hauteurs différentes. Et munir la classe *Avancer* d'un générateur aléatoire de nuages pour avoir des nuages qui défilent sur l'écran.

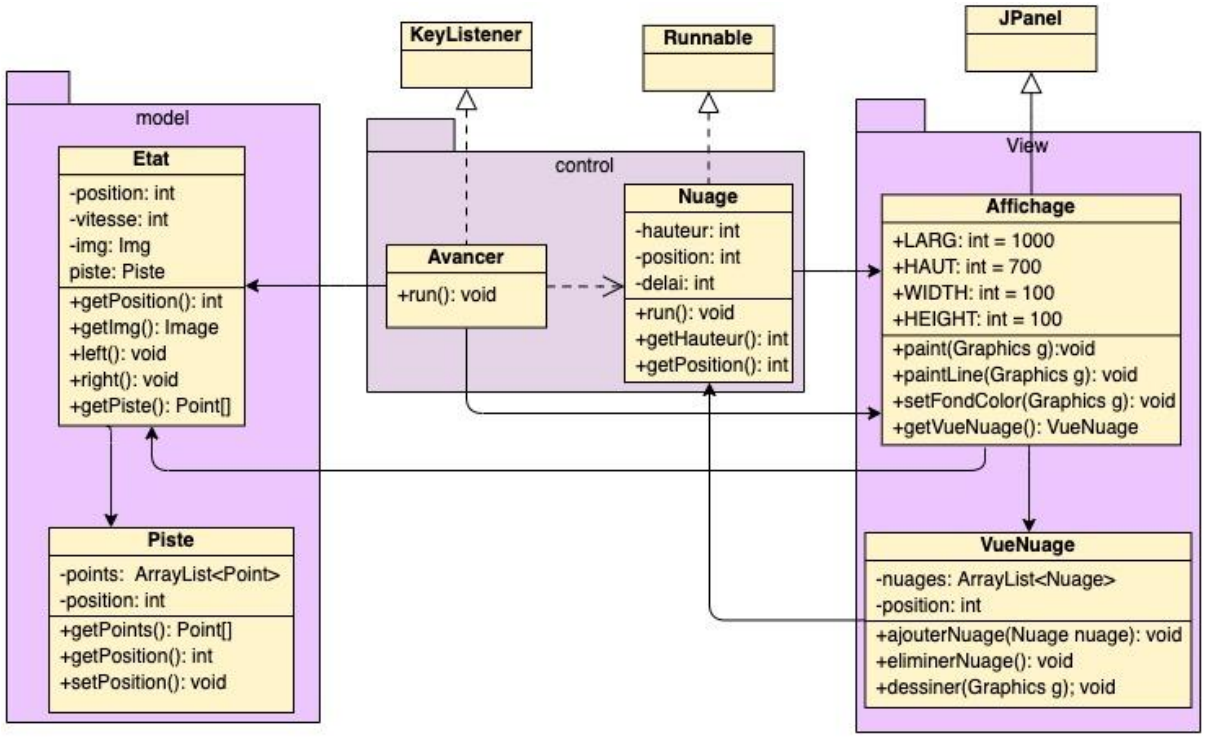


Figure-4 de la création et des mouvements des nuages

Ensuite, nous voulons ajouter des arbres des deux côtés de la route. Nous créons ensuite une classe *Tree* qui implémente un *thread*. La vue du MVC doit alors afficher les arbres. Dans la classe Tree, la hauteur initiale des arbres est fixée égale à l'horizon, mais la position de l'arbre est aléatoire, afin de faire apparaître l'arbre à partir de différentes positions sur l'horizon. Afin d'éviter que l'arbre n'apparaisse sur la route, il faut limiter la valeur de la position de l'arbre. Et munir la classe *Avancer* d'un générateur aléatoire de arbres pour avoir des arbres qui défilent sur l'écran.

Pour limiter la position de l'arbre:

```
Point[] points <- piste.points;
double x1 <- 0.0, y1 <- 0.0, x2 <- 0.0, y2 <- 0.0, x_piste;
double k, b;
Pour i de 0 à points.length - 1 faire
        Si (points[i].y >= Affichage.HORIZON
           && points[i+1].y <= Affichage.HORIZON) alors
        /*(x1, y1),(x2, y2) sont 2 points sur le côté gauche de la route,
        Affichage.HORIZON est entre ces deux points*/
        x1 <- points[i].x; x2 <- points[i+1].x;
        y1 <- points[i].y; y2 <- points[i+1].y;
k <- (y2 - y1) / (x2 -x1);
b <- (x2*y1 - x1*y2)/(x2 - x1);
x_piste <- (Affichage.HORIZON - b)/k;
Faire
   this.position <- new Random().nextInt(300) + Affichage.LARG/2 - 150 ;
Tant que position + WIDTH_TREE >= x_piste
        && position <= x_piste + WID_PISTE
```

(Lorsque nous sommes passés aux courbes de Bézier pour composer des routes, cette méthode n'était plus applicable. La méthode de calcul *x_piste* modifiée spécifique sera présentée ci-dessous.)

Pour permettre à la piste et aux arbres de suivre le mouvement à gauche ou à droite de la moto et de se déplacer à droite ou à gauche. Pour la piste, nous avons ajouté deux fonctions left () et right () dans la classe *Piste*. Lorsque *KeyPressed* est surveillé, ces deux fonctions sont appelées. Pour les arbres, lorsque la position de la route change, l'arbre change avec le changement de route.

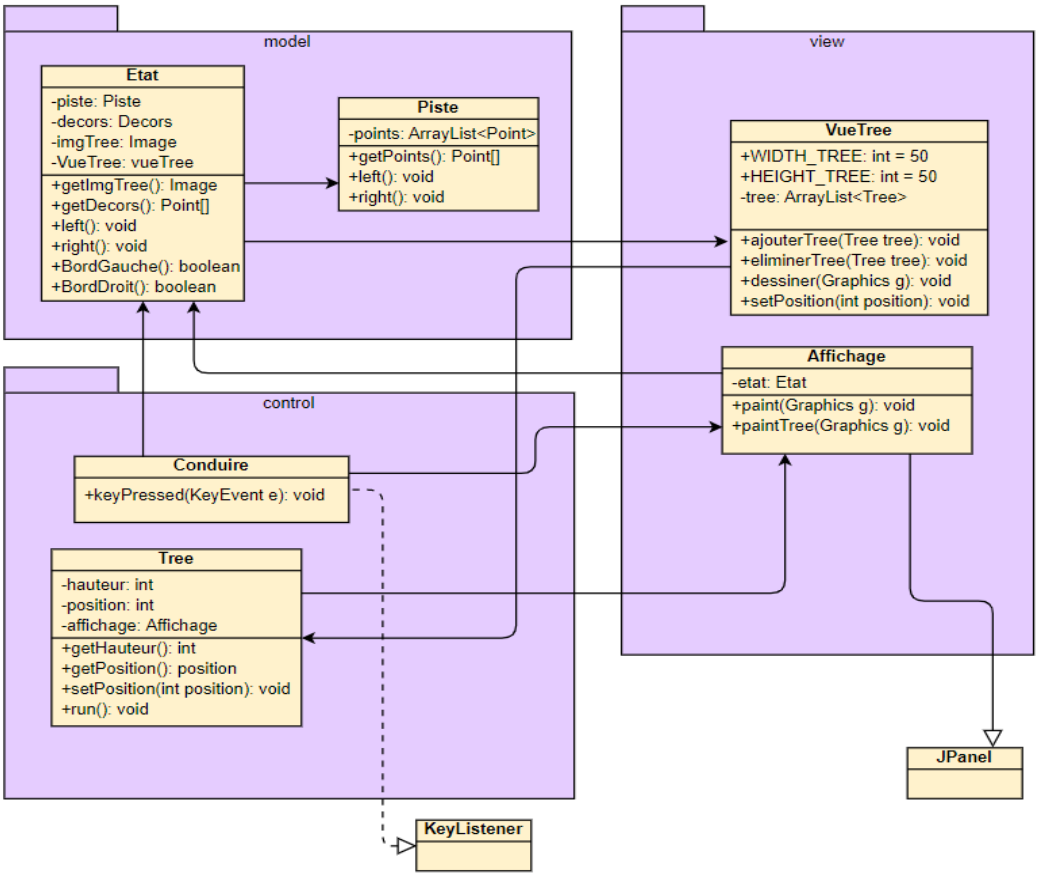


Figure-5 de la création et des mouvements des arbres

(Ce qui précède est la conception originale des arbres. Après avoir prévu d'ajouter plus d'obstacles, nous avons constaté qu'ils étaient presque similaires, nous avons donc changé la structure du code.)

Pour l'ajout d'obstacles, afin de réduire la redondance du code, nous créons une classe *Decors* qui est père de la classe Arbre et la classe Barrage et une classe VueDecors avec des fils *VueArbre* et *VueBarrage*. Alors dans la classe *Etat*, nous avons une liste qui contient tous les décors au sol (qui vont avancer avec la piste). Nous pouvons parcourir la liste quand on en a besoin.

C'est l'heure de détecter des collisions permettant de ralentir le véhicule. Nous avons une méthode collision dans la classe. On parcourt la liste des décors et si on a une collision, on perd une vitesse fixe.

Pour détecter la collision (cf le dessin ci-dessous):

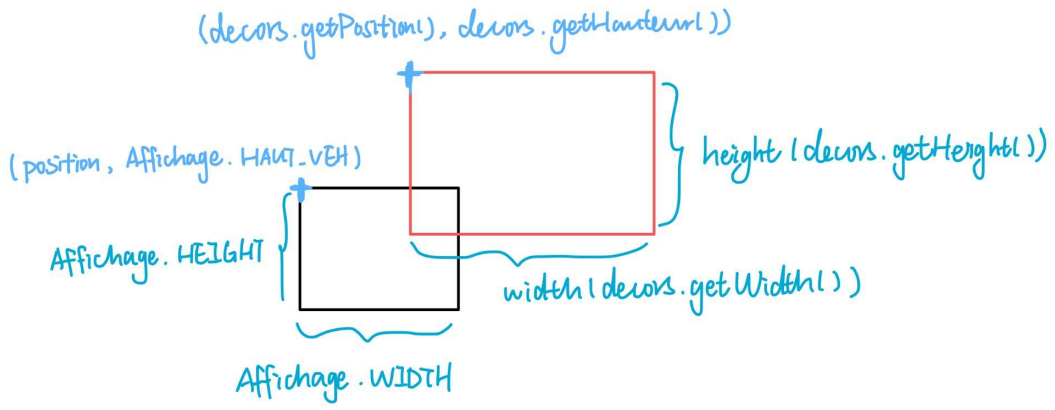
Nous détectons d'abord s'il y a une collision dans le sens vertical:

```
o.getHauteur() <= (Affichage.HAUT_VEH + Affichage.HEIGHT) &&

(o.getHauteur() + height) >= Affichage.HAUT_VEH
```

Et puis, nous détectons s'il y a une collision dans le sens horizontal:

```
o.getPosition() <= this.position+Affichage.WIDTH && o.getPosition()+width >= this.position
```



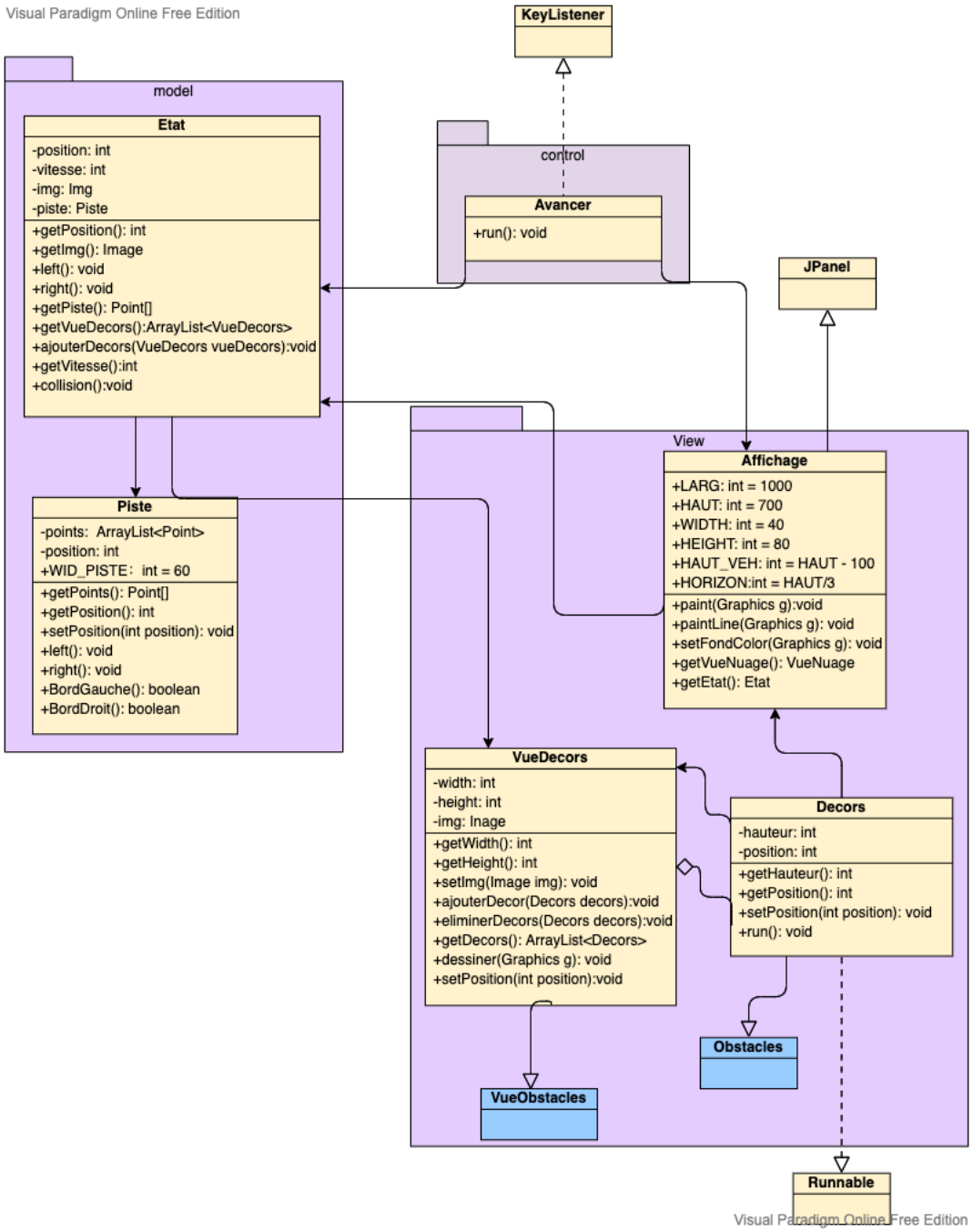


Figure-6 des décors et la détection des collisions

Maintenant, nous voulons ajouter des points de contrôle au jeu, pour cela nous avons ajouté une classe pour définir les états du point de contrôle. Nous jugeons dans la classe *Avancer* si le point de contrôle doit être dans la plage visible de la fenêtre. Si le point de contrôle est dans la plage visible de la fenêtre, les données du point de contrôle doivent être mises à jour avec le déplacement de la route, et elles doivent être affiliées dans la fenêtre de la classe *Affichage*.

Nous nous sommes souvenus de l'heure de début du jeu ***DEBUT*** lors de la création de Piste. Lorsque l'heure actuelle moins l'heure de début du jeu est égale au temps restant du jeu (ajout continu de bonus), nous arrêterons le jeu.

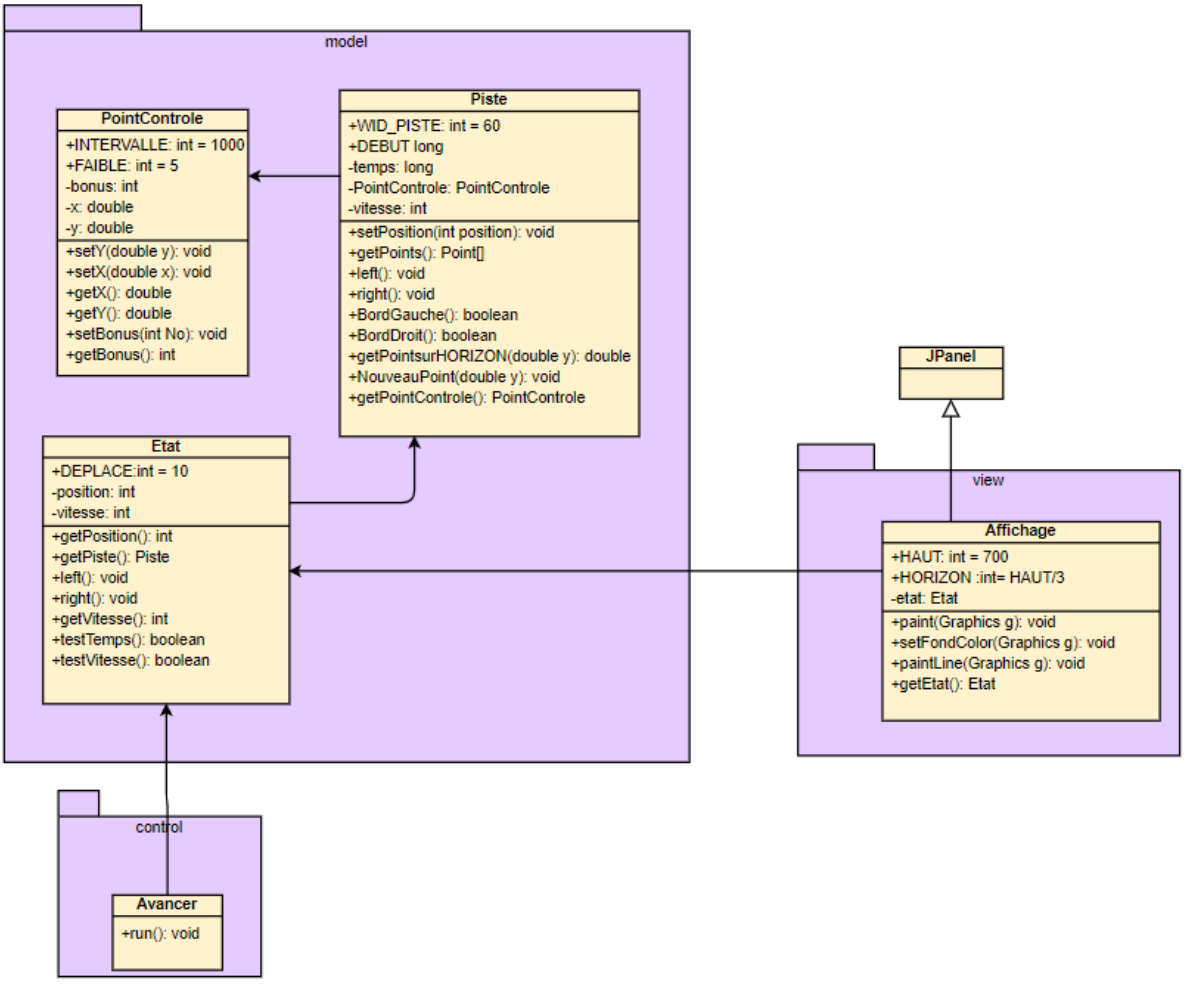


Figure-7 le mécanisme de points de contrôles

Pour réaliser le mécanisme de calcul de la vitesse du véhicule, nous avons une méthode *getPointSurPiste()* afin d'obtenir l'abscisse gauche de la piste. Et puis , en utilisant une méthode *setAcceleration()* pour accomplir un mécanisme de calcul de l'accélération du véhicule en fonction de la position par rapport à la piste. Si le véhicule est sur la piste, l'accélération est 8. Sinon s'il est près de la piste, l'accélération est 8 sinon l'accélération est -8. Nous pouvons obtenir du temps en utilisant System.*currentTimeMillis*().

1. *Si le véhicule est sur la piste :*

   ***position du véhicule** + Affichage.**WIDTH** >= **piste**.getPointSurPiste(**Y_VEH**) &&*

   position du véhicule** <= **piste**.getPointSurPiste(**Y_VEH**) + Piste.**WID_PISTE

2. *Si le véhicule est en gauche de la piste :*

   ***position du véhicule** + Affichage.**WIDTH** < **piste**.getPointSurPiste(**Y_VEH**))*

   s'il est près de la piste (le véhicule est en gauche de la piste) :

   *(**piste**.getPointSurPiste(**Y_VEH**) - **this**.**ancienPos**) >= (**piste**.getPointSurPiste(**Y_VEH**) - **this**.**position**))*

3. *Si ce ne sont pas les deux cas ci-dessus, alors le véhicule est en droite de la piste :*

   s'il est près de la piste (le véhicule est en droite de la piste) :

   *(**this**.**ancienPos** - **piste**.getPointSurPiste(**Y_VEH**)) >= (**this**.**position** - **piste**.getPointSurPiste(**Y_VEH**))*

On sait la formule ci-dessous en connaissant l'ancienne vitesse, l'accélération et le temps :

$$v = vi + at$$

a = accélération, t = temps

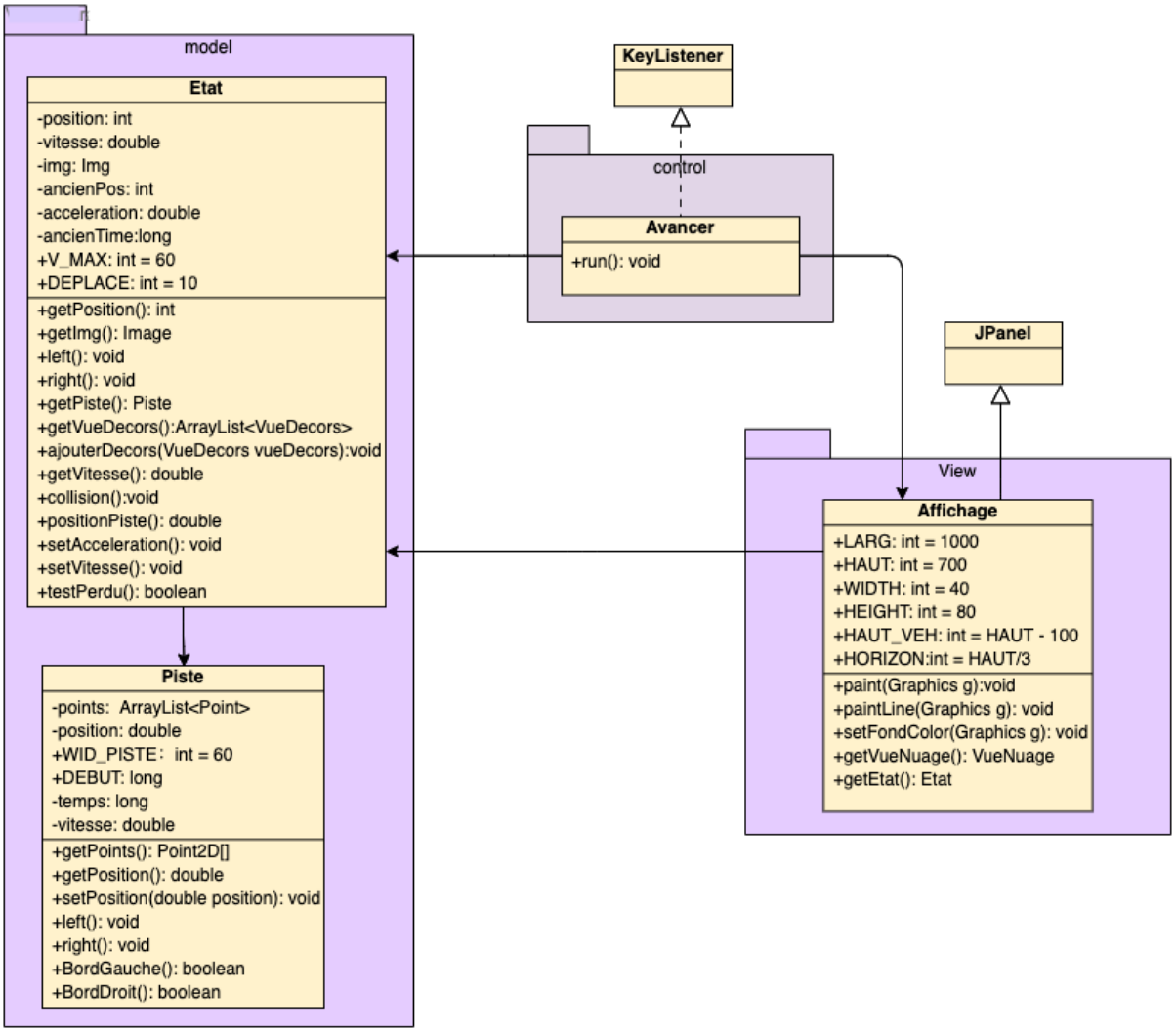


Figure-8 le mécanisme d'accélération et vitesse

Maintenant, pour avoir une piste plus élégante, nous allons utiliser des *courbes de Bézier*. Une courbe de Bézier est caractérisée par 3 points: les deux extrémités plus un point de contrôle, qui définissent un polygone dans lequel la courbe doit être englobée.

La formule du troisième point milieu en connaissant les deux points:

point milieu = (point1 + point2)/2

Le point milieu et le prochain(ancien) point milieu seront les deux extrémités, et le point 2(point 1) sera le point de contrôle.

La bibliothèque AWT définit les classes `QuadCurve2D` et `CubicCurve2D` qui implémentent cet algorithme. Pour l'utiliser, nous devons créer un objet de la classe

interne XXX.Double ou XXX.Float puis utiliser la méthode *setCurve* pour lui donner les points caractéristiques, comme sur l'exemple ci-dessous :

```
QuadCurve2D courbe = new QuadCurve2D.Double();
debut = new Point2D.Double(x1,y1);
ctrl = new Point2D.Double(x1,y1);
fin = new Point2D.Double(x1,y1);
courbe.setCurve(debut,ctrl,fin);
```

Pour dessiner cette courbe, il faut utiliser la classe *Graphics2D* :

```
Graphics2D g2 = (Graphics2D)g;
g2.draw(courbe);
```

Lorsque nous sommes passés à l'utilisation des courbes de Bézier pour composer la route, nous avons également réécrit la fonction adaptée à la recherche de points sur la courbe.

Tout d'abord, nous trouvons à quel segment de la courbe se situe la valeur y cible. Utilisez ensuite le point de départ, le point de contrôle, le point final et la valeur y cible comme paramètres pour trouver la valeur de t.

dans la classe *Piste*:

1.Calculer t:

```
computeT(Point2D p0, Point2D p1,Point2D p2, double p):
  //Résolvez une équation quadratique pour trouver t
  double[] tt <- equation2(p0.getY(),p1.getY(),p2.getY(),p);
  Si tt[0] != -1 et tt[1] != -1 alors
    //Si l'équation a deux solutions, les deux valeurs t sont placées respectivement dans la formule de la courbe de Bézier.
    //Comparez la valeur y obtenue et prenez une solution plus proche de la valeur y du point cible p.
    double pointTest <- getPointOnQuadraticCurve(p0, p1, p2, tt[0]);
    Si Math.abs(pointTest - p) < 0.01 alors return tt[0];
    Sinon return tt[1];
  Sinon return tt[0];
```

2.Résolvez des équation: *yP = (1-t)2 yP0 2t(1-t) yP1 t2 * yP2*

```
equation2(double y0, double y1, double y2, double yp):
  double a <- y0 - y1 * 2 + y2,
         b <- 2*(y1 - y0),
         c <- y0 - yp;
  double[] tt <- new double[2];

  //Si a = 0, la solution est - c / b
  Si a = 0 et b != 0 alors
     tt[0] <- - c / b; tt[1] <- - 1;
  Sinon
     double sq <- Math.sqrt( b * b - 4 * a * c );
     tt[0] <- (sq - b)/ (2 * a);
     tt[1] <- (-sq - b) / (2 * a);

     //Déterminer si t atteint la limite [0,1]
     Si (tt[0] <= 1 et tt[0]>= 0)
        et (tt[1] <= 1 et tt[1]>= 0) alors  return tt;
     Sinon
        Si tt[0] <= 1 et tt[0]>= 0 alors tt[1] <- -1;
        Sinon tt[0] <- tt[1]; tt[1] <- -1;
        Fin Si
Fin si
return tt;
```

3. Calculer x: *Calculer la valeur x de la courbe de Bézier en fonction de la valeur t*

```
getPointOnQuadraticCurve(Point2D p0, Point2D p1, Point2D p2, double t):
  return  (1-t) * (1-t) * p0.getX() + 2 * t * (1-t) * p1.getX() + t * t * p2.getX();
```

Pour rendre notre interface plus joli, nous avons ajouté des oiseaux. Nous avons trouvé un GIF animé d'un oiseau sur internet et on a utilisé un outil de manipulation d'images séparant ce GIF animé en 6 images fixes. La structure de l'oiseau est similaire au nuage, composée d'un thread Oiseau et d'une classe *VueOiseau*.

Mais la classe *Oiseau* a un autre attribut *etat* qui permet de savoir dans quelle position est l'oiseau.

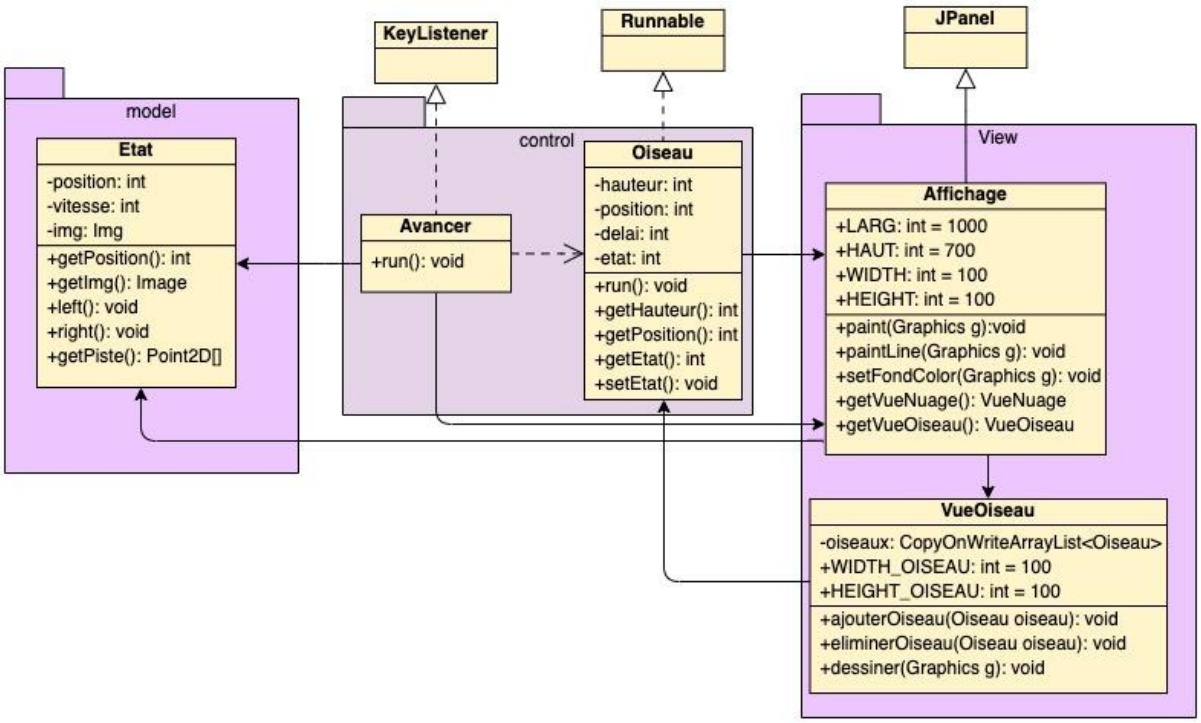


Figure-9 du mécanisme des oiseaux

Afin d'ajouter du plaisir au jeu, nous avons ajouté des adversaires. La structure du code de cette fonction est similaire aux décors, principalement composés de la classe *Adversaire* et de la classe *VueAdversaire*.

La différence est que l'adversaire doit apparaître au milieu ou près de la route et qu'il a une vitesse relative avec la route.

```
Faire
  this.position = new Random().nextInt(300) + Affichage.LARG/2 - 150 ;

  //L'adversaire doit apparaître au milieu ou près de la route
Tant que ( (this.position + VueAdversaire.WIDTH_ADV + 10) < x_piste
        && this.position > (x_piste + Piste.WID_PISTE + 10));
```

Nous fixons une vitesse relative fixe afin que le joueur puisse ou non dépasser l'adversaire.

```
this.hauteur += affichage.getEtat().getVitesse() - 25;
```

Lorsque l'ordonnée de l'adversaire dépasse la plage visible de la fenêtre, elle sera supprimée, et cela signifie que le joueur a complètement surpassé l'adversaire à ce moment. Nous donnerons au joueur 10 points en récompense.

```
Si this.hauteur > Affichage.HAUT
   //Lorsque le joueur dépasse l'adversaire, donnez un bonus de 10 points.
  this.affichage.getEtat().setScore(10);
  this.affichage.getVueAdversaire().eliminerAdversaire(this);
Fin si
```

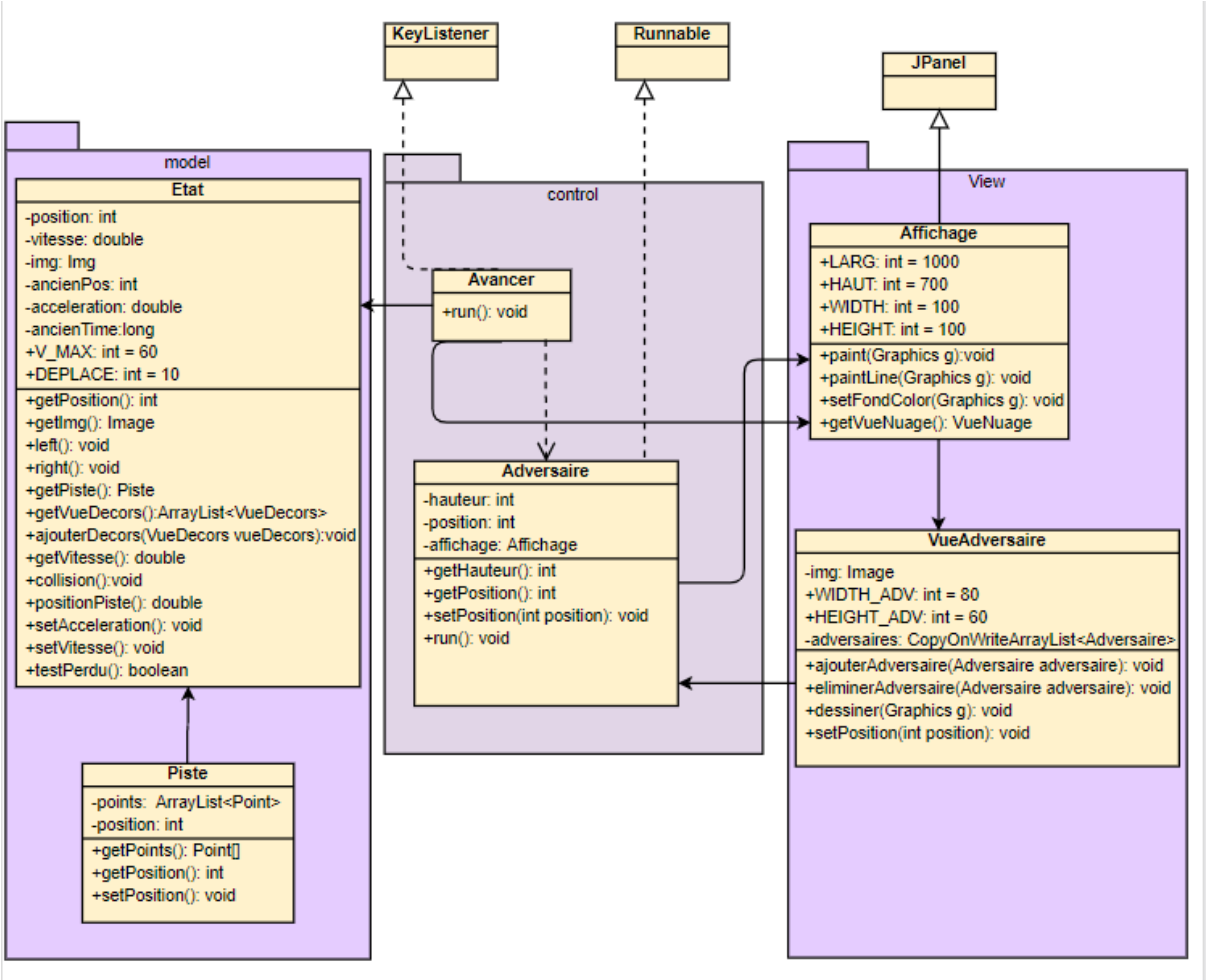


Figure-10 le mécanisme des adversaires

Pour que le véhicule puisse monter à l'aide de la touche « haut ». Nous avons ajouté un attribut boolean *up*, quand on appuie la touche « haut », *up* devient 'true', quand on la relâche, *up* devient 'false'.

Dans la fonction up(), on diminue l'ordonne du véhicule si up est true et l'ordonne minimale est ***Y_ciel*** et la vitesse minimale pour voler est 15. Le véhicule s'éloigne de la piste et va donc perdre de la vitesse. Alors on diminue une valeur fixe dans up(). Il redescend tout seul vers le sol lorsque sa vitesse est réduite à 8.

Afin de donner au joueur l'impression que le véhicule est volé vers le ciel, nous avons une classe Ombre. L'ombre bouge avec le véhicule et il sera agrandi /réduit quand le véhicule se déplace vers le ciel/sol.

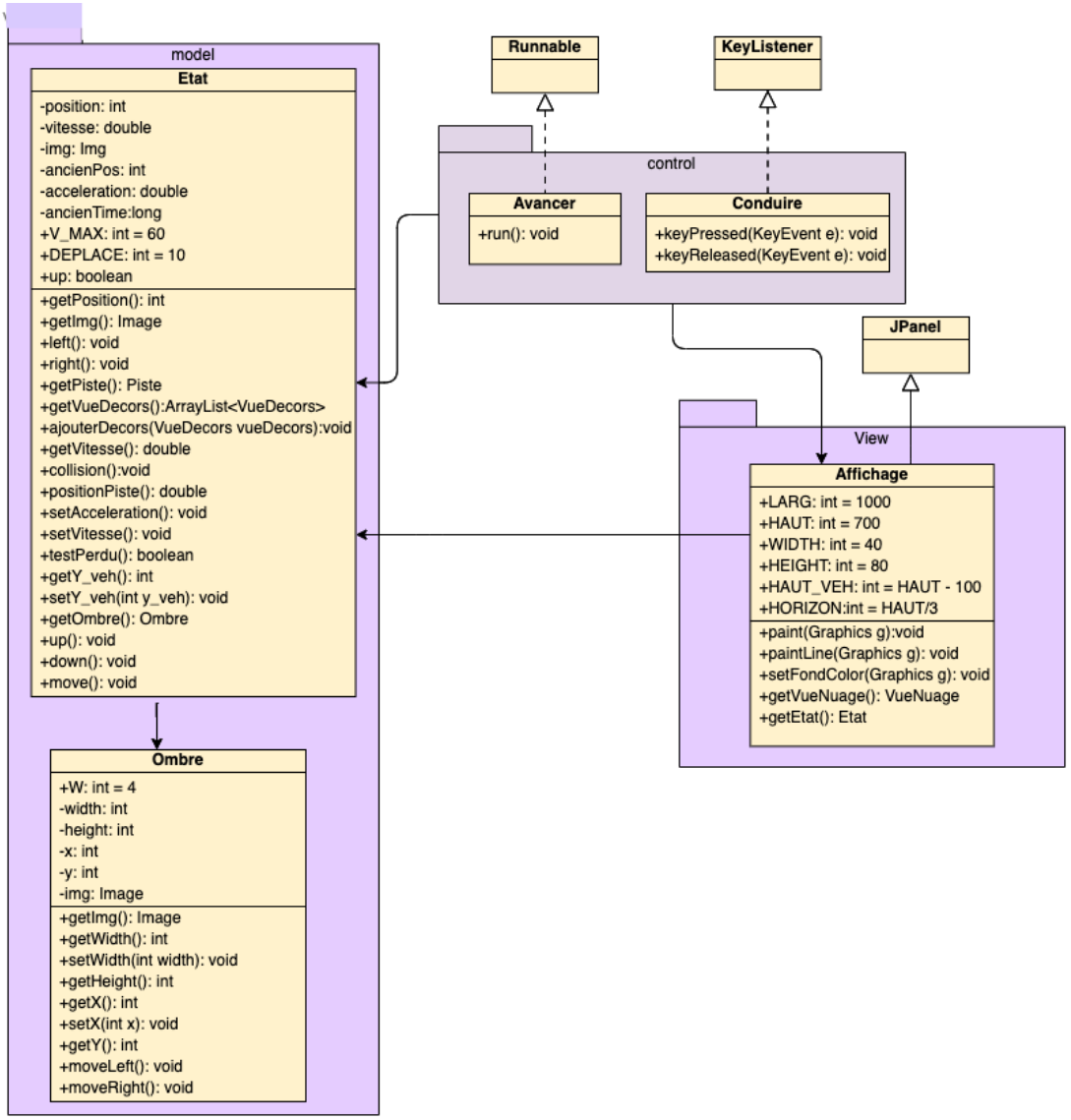


Figure-11 le mécanisme de voler avec la touche “haut”

Afin d'augmenter la jouabilité du jeu, nous avons ajouté une fonction *gameOver()* dans la classe *Etat* permettant de mémoriser le meilleur score.

On utilise *createNewFile()* pour créer un nouveau fichier 'score.txt' si le joueur joue pour la première fois et écrit le score actuel dans ce fichier. Dans le jeu suivant, nous lisons le score dans le fichier et le comparons avec le score actuel. Si le score actuel est plus élevé, l'écrivez dans le fichier et écrasez le score d'origine.

Pour lire le score:

```
BufferedReader in =new BufferedReader(new FileReader(file));

String contentLine = in.readLine();

int scoreCourrent = Integer.parseInt(contentLine);
```

Pour écrire le score:

```
FileWriter out = new FileWriter(file);

out.write(String.valueOf(this.bestScore));
```

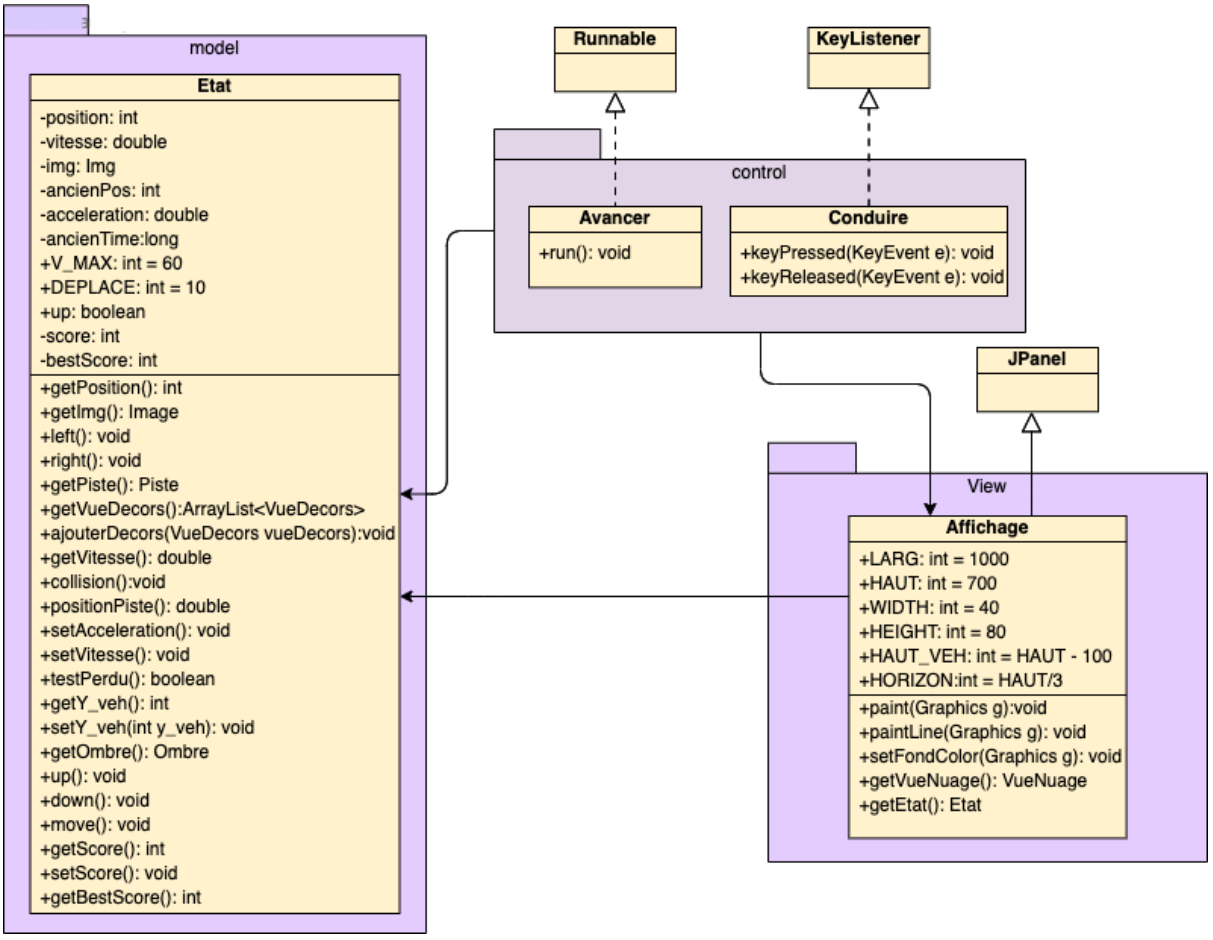


Figure-12 la mémorisation du meilleur score

Enfin, nous avons ajouté un écran de bienvenue à notre jeu. Nous avons ajouté un *JLabel* dont le contenu est une image à la classe *Bienvenus*. Et lui a ajouté un *MouseListener*. Lorsque nous détectons que le joueur clique, ce JPanel sera supprimé du *JFrame*, *Affichage* sera ajouté et les Threads seront lancés.

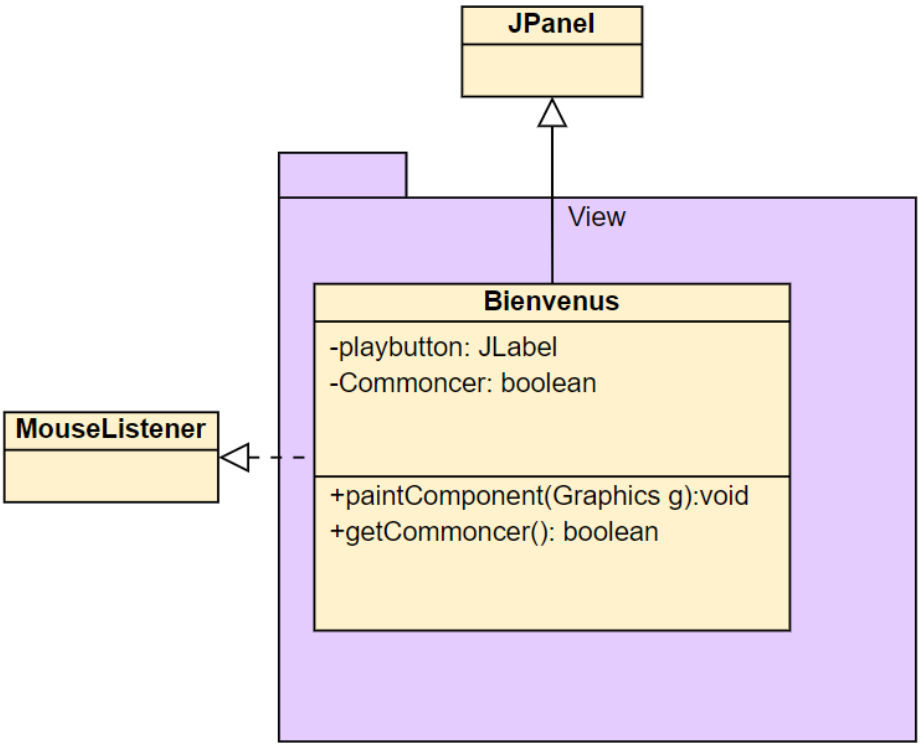
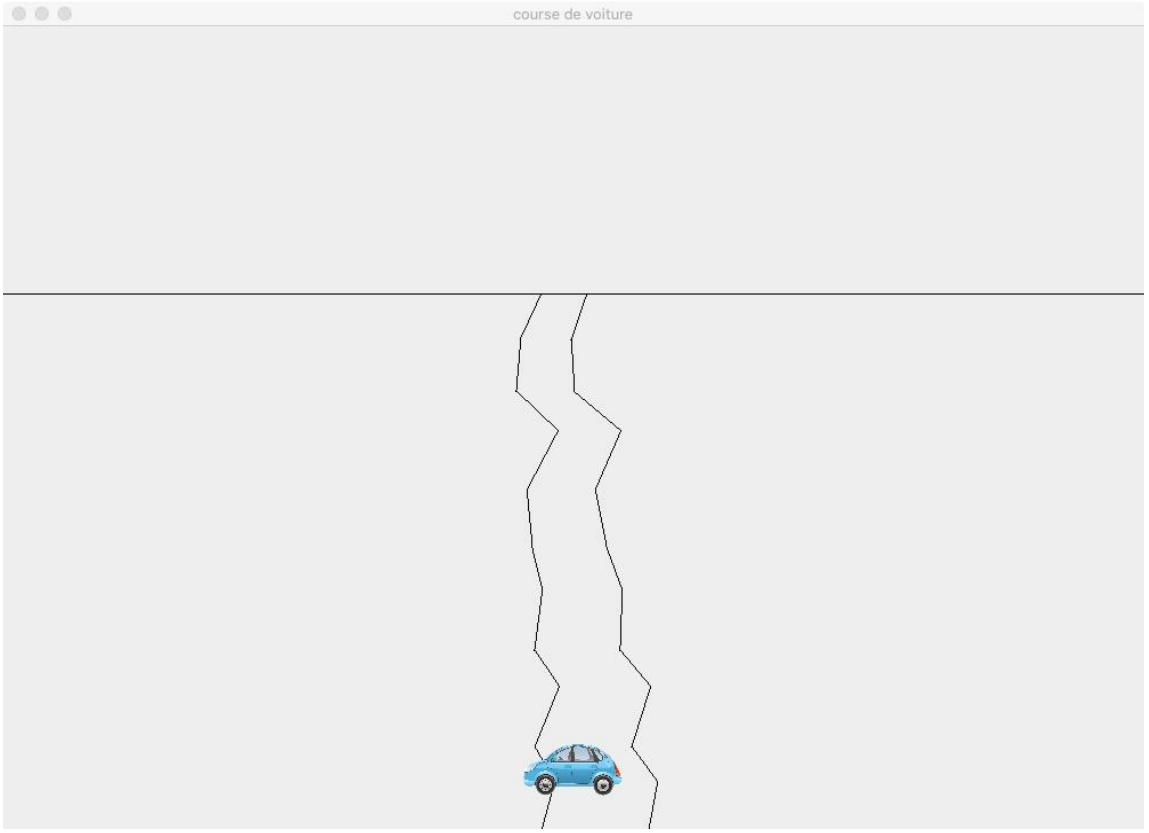
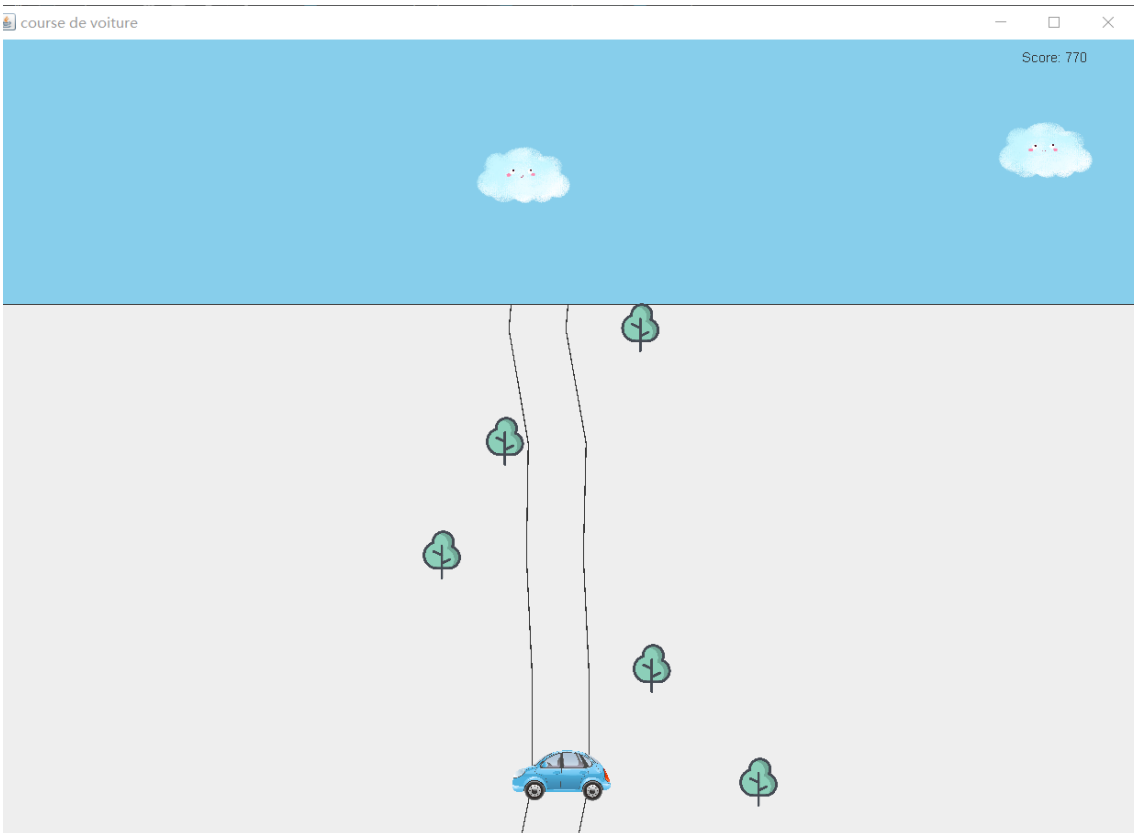
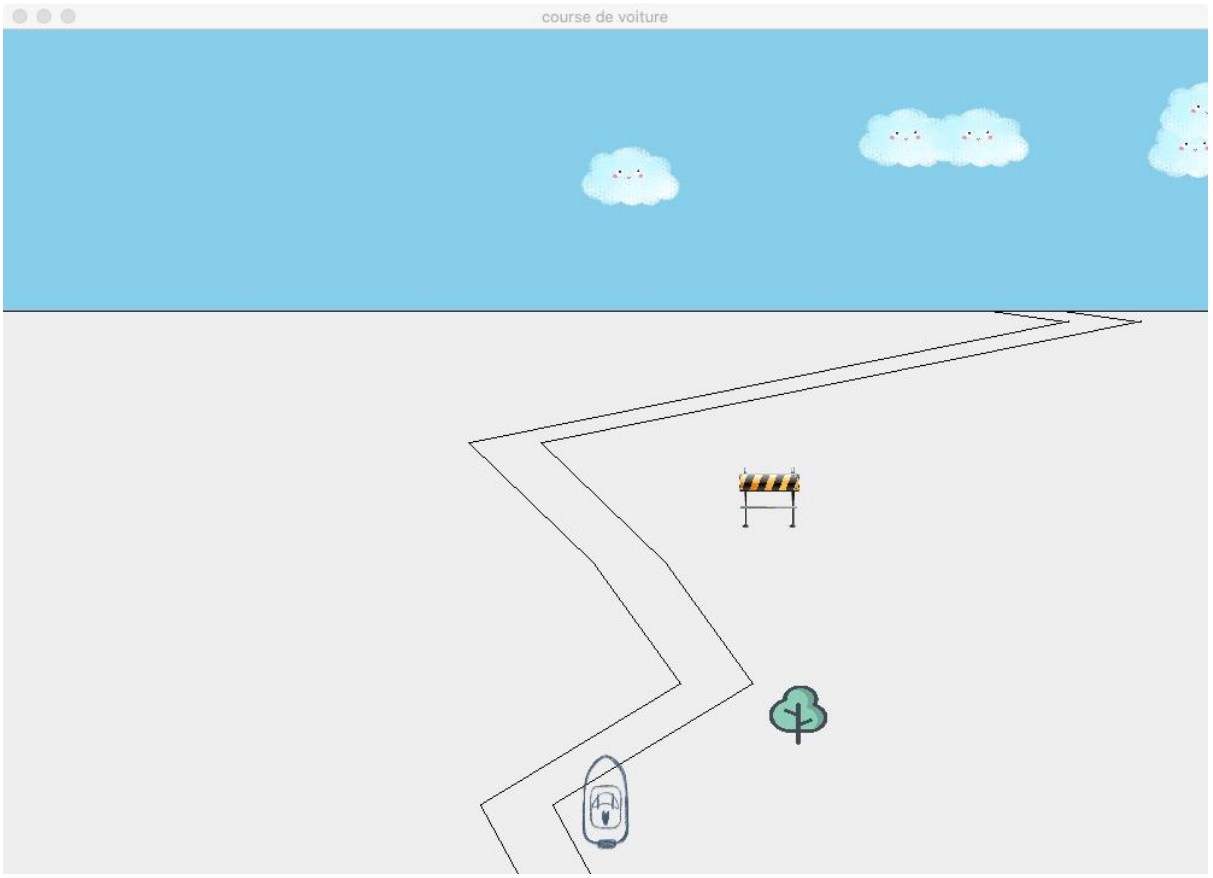


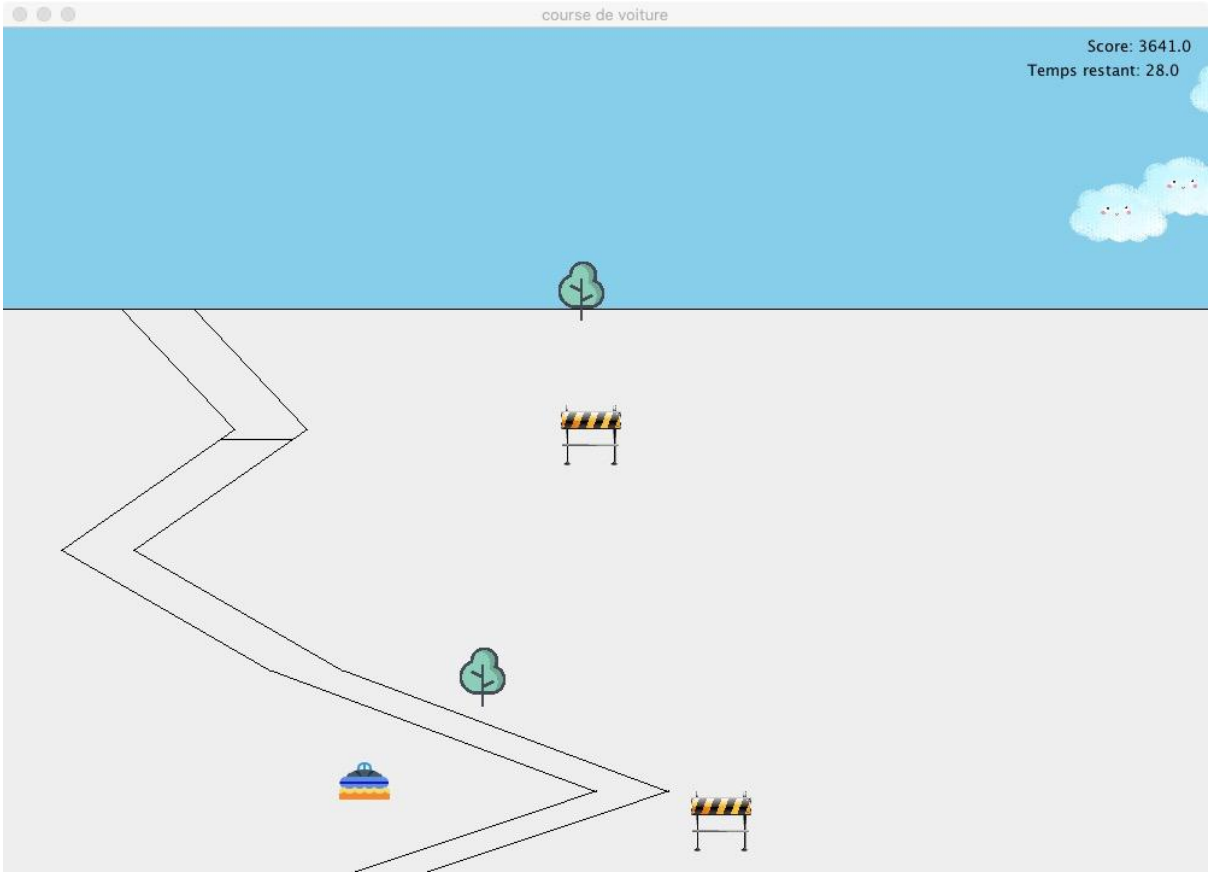
Figure-11 la création d'un écran d'accueil

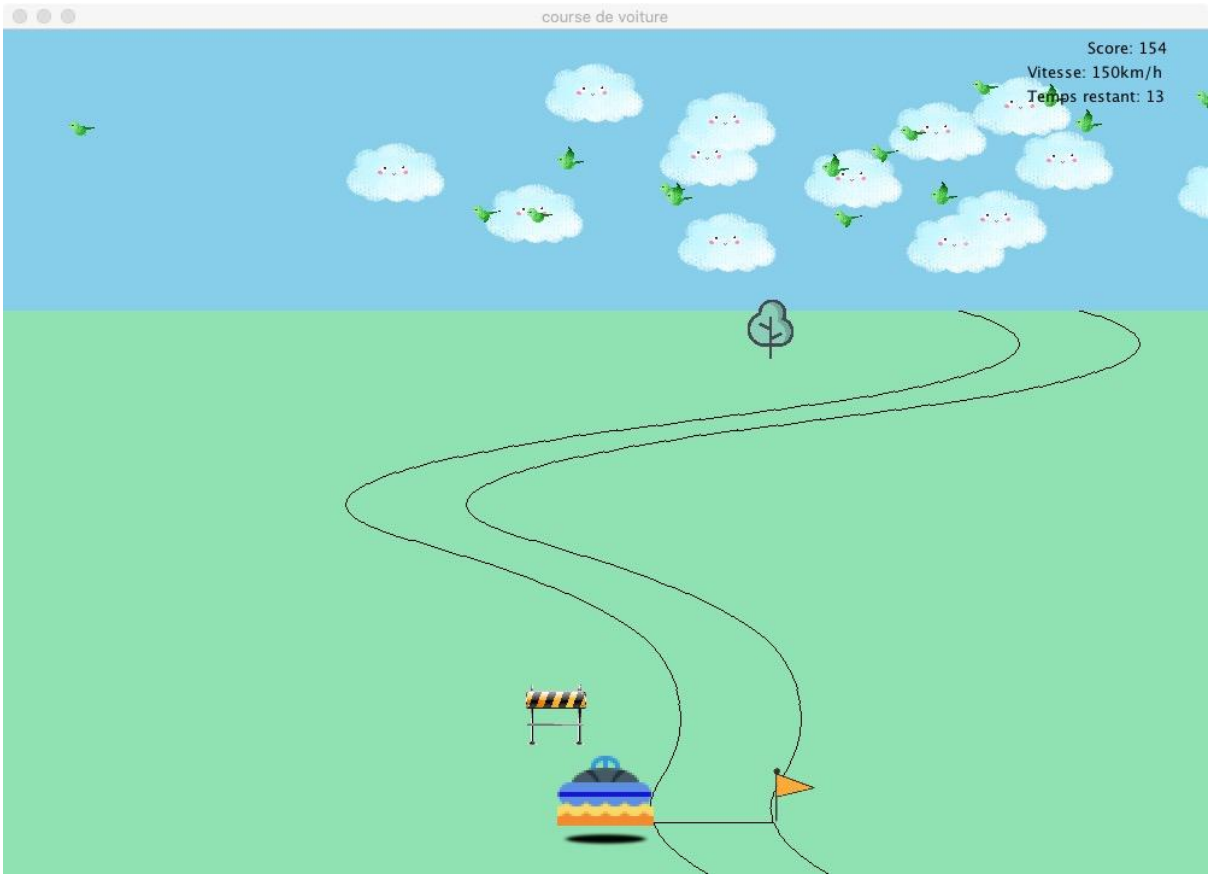
Résultat



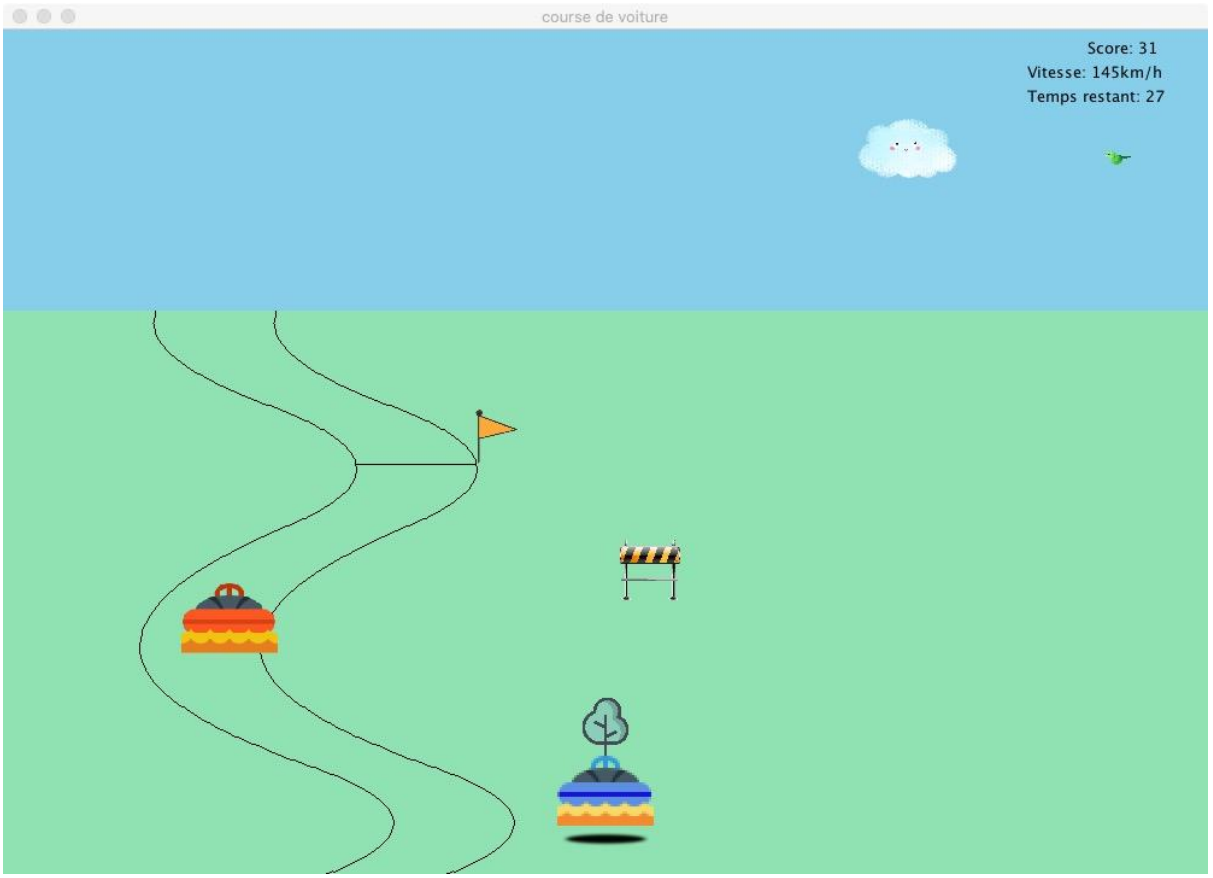


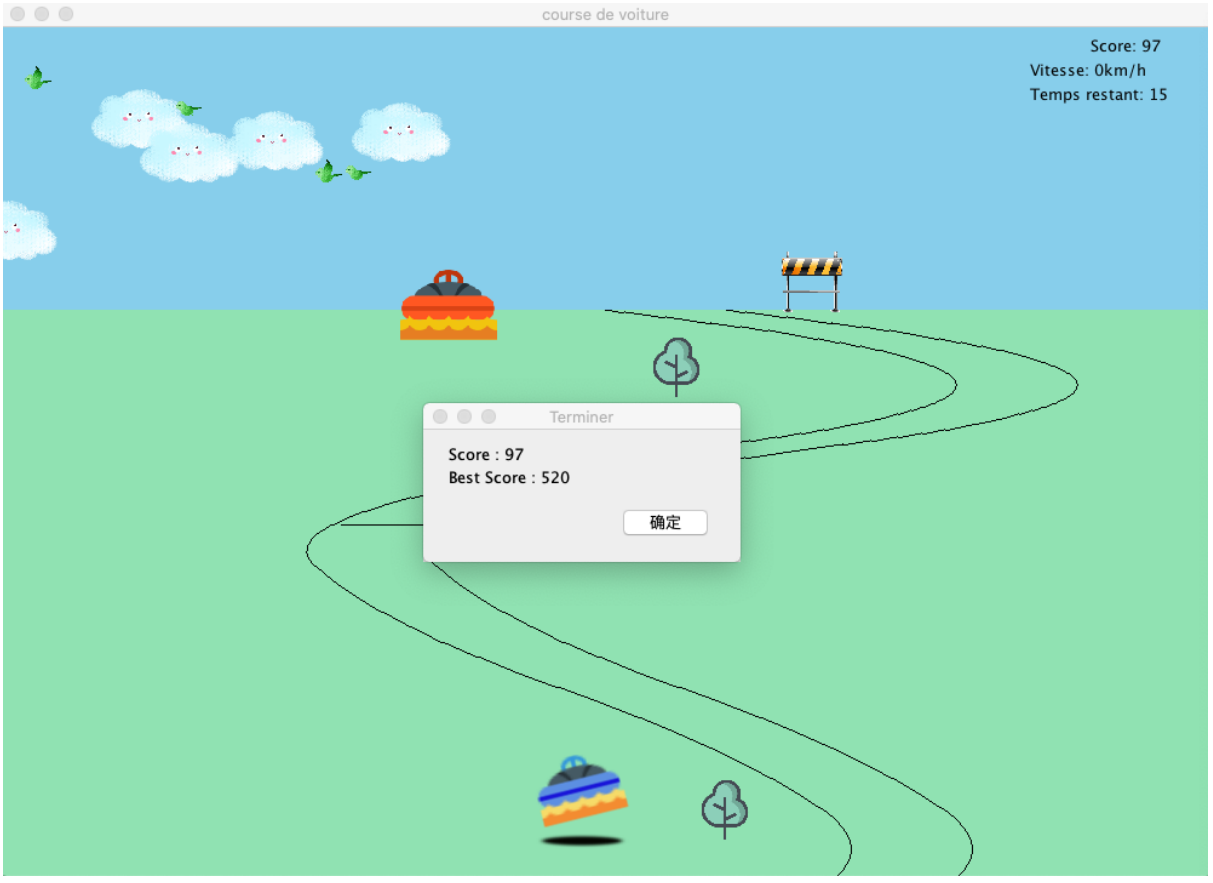

course de voiture

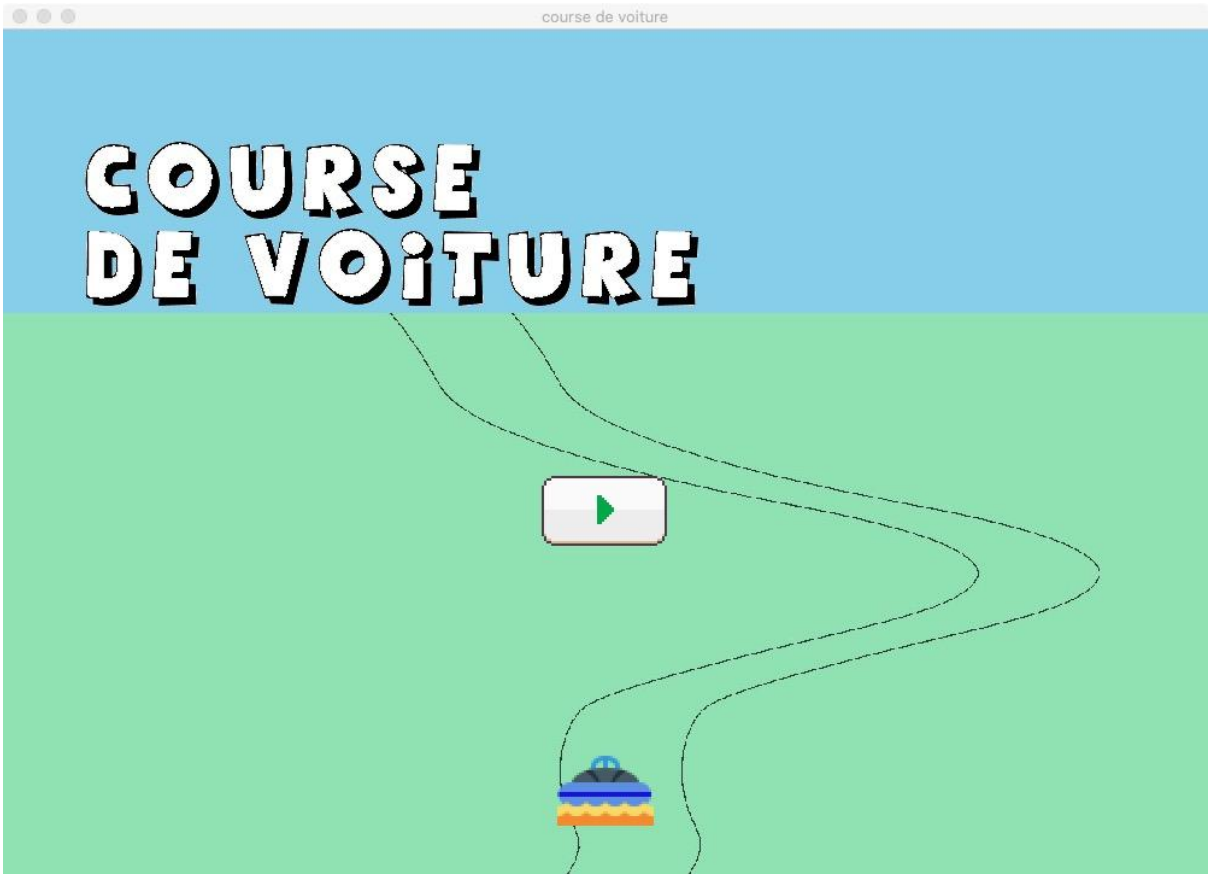

course de voiture
Score: 3641.0
Temps restant: 28.0


course de voiture
Score: 154
Vitesse: 150km/h
Temps restant: 13


course de voiture
Score: 158
Vitesse: 0km/h
Temps restant: 11


course de voiture
Score: 31
Vitesse: 145km/h
Temps restant: 27


course de voiture
Score: 97
Vitesse: 0km/h
Temps restant: 15
Terminer
Score : 97
Best Score : 520
確定



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 « *FenetrePrincipal* » puis « Run as Java Application » (dans le cas Eclipse) ou « Run '*FenetrePrincipal.main()'* » (dans le cas IntelliJ IDEA).
  - Appuyez sur les touches gauche, droite et haut du clavier pour contrôler le mouvement du véhicule. Faites attention à ne pas heurter les obstacles et conduisez autant que possible sur la piste, en dépassant les concurrents.

Documentation développeur

C’est la classe FenetrePrincipal qui contient la méthode `main`. Nous utilisons le modèle MVC pour organiser les codes afin de les rendre plus lisibles et propres.

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 et celle du véhicule.

Nous avons une classe père *VueDecors* afin d’afficher les décors qui avancent avec la piste. Elle a des fils *VueArbre* et *VueBarrage* pour dessiner des arbres et des barrages, les décors peuvent causer la perte de vitesse du véhicule. La classe *VueNuage* et *VueNuage* sont pour afficher les décors du fond. Nous avons aussi des adversaires définissant dans la classe *VueAdversaire* et un écran d'accueil *Bienvenus*.

Le package `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.

Vous pouvez contrôler la vitesse du déplacement à gauche ou à droite du véhicule en modifiant la constante ***DEPLACE*** de la classe *Etat*. Il est possible de changer la largeur de la piste avec la constante ***WID_PISTE***. ***V_MAX*** vous permet de définir la vitesse maximale du véhicule et ***Y_ciel*** détermine la plus haute altitude que le véhicule peut voler. Nous enregistrons l'heure de ***DEBUT*** du jeu et le temps restant dans la classe *Piste* pour déterminer si le jeu est terminé.

Le package `control` vous permet de savoir comment gérer les événements et la manière dont l’état du modèle change. L’animation de la piste et des décors et le mouvement du véhicule sont définis ici.

Pour l’instant, nous n’avons pas créé une sensation de profondeur, il faut maîtriser des connaissances mathématiques, nous pouvons calculer les projections sur le plan de l’écran de différents points dans un espace en trois dimensions.

Conclusion et perspectives

Nous avons réalisé une fenêtre dans laquelle est dessiné le véhicule, l'horizon, la piste et des décors. Nous avons un mécanisme du déplacement du véhicule à droite, à gauche et à haut lorsqu'on tape les flèches du clavier. La vitesse n'est pas gérée directement par le joueur mais elle dépend de la proximité à la piste, qui sert aussi d'alimentation. Lorsque le joueur est sur la piste, sa vitesse augmente (jusqu'à un maximum). Mais dès qu'il s'écarte de cette zone d'alimentation, il perd progressivement de la vitesse. La collision avec les obstacles cause une perte de vitesse aussi. Lorsque la vitesse ou le temps restant atteint 0, on perd le jeu, tous les threads s'arrêtent.

Ce qui était difficile, c'est que la piste devrait être infinie et calculée à partir d'une ligne brisée verticale limitée par l'horizon, afin de le résoudre, nous avions caché la partie au-delà de l'horizon à l'aide de la fonction ***fillRect***. Il faut aussi bien définir les coordonnées des points pour construire une ligne correcte.

Et nous avons toujours une erreur par rapport au décors. Même si nous avons écrit la limite de l'abscisse du décors généré dans le code, il y a encore des décors qui apparaissent sur la route. Nous avons constaté que la raison de l'erreur est que nous avons modifié la méthode de génération de piste afin que la limitation de génération du décors ne s'applique pas à la situation actuelle.

Nous faisons bien avancer le projet conformément à notre plan. Nous avons ajouté des obstacles et réduit la vitesse de la moto lorsqu'elle détecte que la moto est en collision avec l'obstacle. Nous avons également réalisé la fonction de points de contrôle, où des points de contrôle apparaissent à intervalles réguliers. Notre jeu a donc commencé à être limité par le temps.

La partie la plus difficile était de résoudre le bug, en particulier l'exception occasionnelle. En raison de la contingence de l'erreur, nous pensons qu'elle est causée par le multithreading, c'est à dire que quand un thread modifie la liste, il a un autre thread la parcourant en même temps, donc nous utilisons ***CopyOnWriteArrayList*** au lieu de ***ArrayList***.

Nous voulions colorer la route, mais nous n'avons pas pu trouver comment remplir la zone courbe. Et nous n'avons pas pu réaliser le dessin de la piste rétrécit au bout pour créer une sensation de profondeur.

Malgré tout, grâce à ce projet, nous avons acquis les connaissances de programmation d’interface en java à l'aide de la bibliothèque swing et de bien organiser les codes avec le modèle MVC. Nous avons aussi gagné la capacité à effectuer un travail qui répond à un cahier des charges, la capacité à travailler en groupe et la capacité à chercher les informations sur internet.