Projet PCII : Course de voiture

I – Introduction

L'objectif du projet de PCII est de réaliser un jeu vidéo des années 80 de type « course de voiture » en vue à la première personne. Avant de réaliser ce projet nous avons effectué le tutoriel de projet afin de découvrir et d'adopter certaines méthodes essentielles à la réalisation d'un projet, aussi bien au niveau du code que du rapport. L'objectif de ce tutoriel était de réaliser une version simplifiée de *Flappy Bird*. Le but de notre jeu de course de voiture est de rester le plus longtemps possible dans la course pour gagner un maximum de points. Le joueur dispose d'un temps imparti pour atteindre un point de contrôle lui octroyant du temps de jeu supplémentaire jusqu'au prochain point de contrôle. Il est possible d'effectuer des déplacements horizontaux à l'aide des touches directionnelles. Au fil de la course le joueur peut rencontrer des obstacles à éviter ou encore des concurrents à dépasser. Enfin, plus le joueur se situe proche de la route principale plus il voit son accélération augmentée, lui permettant ainsi de rejoindre le prochain point de contrôle plus rapidement. Le score final dépend de la distance parcourue et du nombre de concurrents dépassés. La partie s'arrête lorsque le temps est écoulé. Notre jeu vidéo est réalisé en Java. Voici un aperçu du style de jeu duquel ce projet est inspiré :

Figure 1: Jeu vidéo d'arcade « Out Run » commercialisé en 1986

II – Analyse globale

Les principales fonctionnalités du jeu à implémenter sont listées ci-dessous.

- Véhicule représenté par une image
- Un horizon représenté par une ligne horizontale
- Mécanisme de contrôle au clavier pour gérer les déplacements horizontaux
- Génération aléatoire d'une piste infinie limitée par l'horizon pour représenter la route
- Apparition de points de contrôle
- Apparition d'obstacles et de concurrents
- Mécanisme de gestion de l'accélération du véhicule en fonction de sa distance à la route

III – Plan de développement

Lors de la première séance nous avons travaillé sur les points suivants :

- Lecture et analyse du sujet (*Carlo et Ugo, 25 mn*)
- Conception, développement du squelette du code suivant le motif MVC (*Ugo, 30 mn*)
- Conception, développement et test de la génération infinie de la route (*Carlo, 90 mn*)
- Implémentation du KeyBoard Listener (*Ugo, 45 mn*)
- Conception, développement et test du thread faisant se déplacer la route (*Carlo, 60 mn*)
- Conception développement et test des éléments de décor : ligne d'horizon et chaîne de montagne (*Ugo, 120 mn*)
- Conception, développement et test de l'apparition des points de contrôle (*Carlo, 60 mn*)
- Documentation du code (*Carlo et Ugo, tout au long de l'écriture du code, 215 mn*)

Le diagramme de Gantt correspondant à la première séance de travail est présenté ci-dessous.

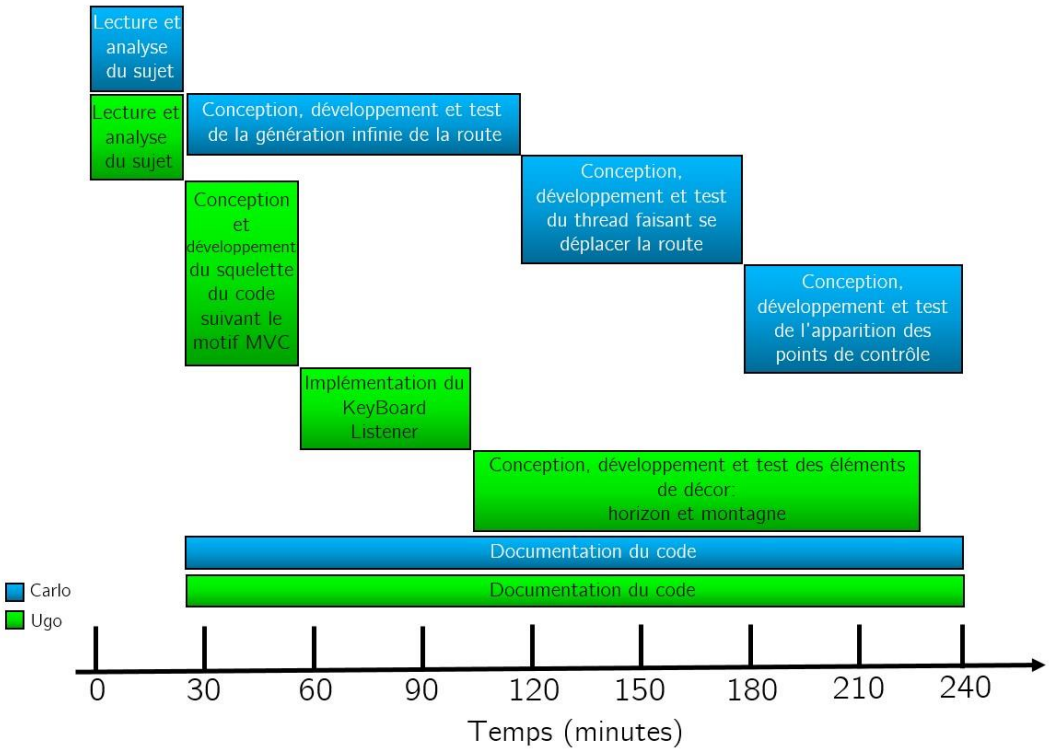


Figure 2: Diagramme de Gantt de la séance 1

Lors des deux séances qui ont suivis nous avons travaillé sur les points suivants :

- Amélioration du code de la première séance (*Carlo et Ugo, tout au long de l'écriture du code, 480 mn*)
- Amélioration de la fluidité du déplacement de la voiture (*Ugo, 30 mn*)
- Implémentation du déplacement du décor et de la route (*Ugo, 40 mn*)
- Conception de : l'accélération, la vitesse de la voiture et le défilement de l'écran (*Carlo, 300 mn*)
- Conception du temps et de son interaction avec les points de contrôle (*Ugo, 240 mn*)
- Documentation du code (*Carlo, 180 mn et Ugo, 170 mn*)

Le diagramme de Gantt correspondant est présenté ci-dessous.

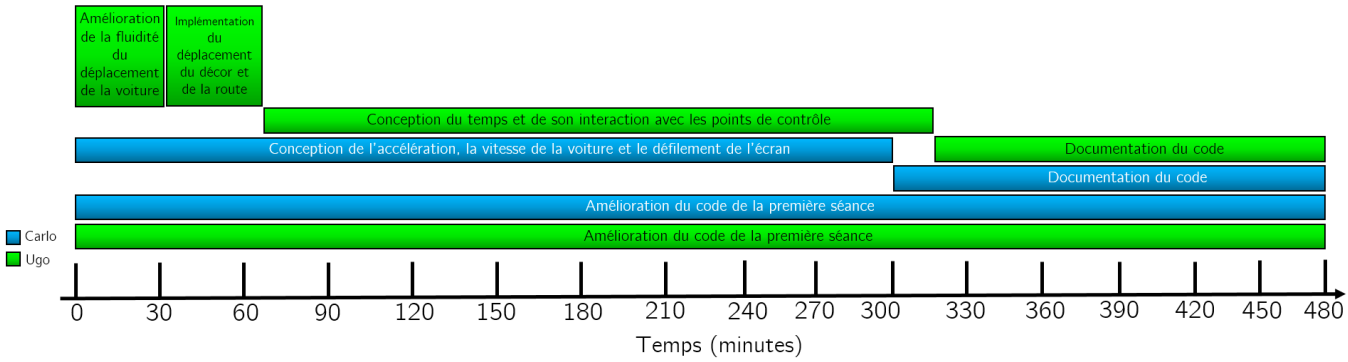


Figure 3: Diagramme de Gantt des séances 2 et 3

Lors des vacances nous avons traité les points suivants :

- Amélioration de la cohésion entre les différentes fonctionnalités implémentées (*Carlo et Ugo, 160 mn*)
- Conception des obstacles, de leur collision (*Carlo, 50 mn*)
- Conception du mouvement des obstacles (*Ugo, 60 mn*)
- Implémentation de l'image de la voiture (*Carlo 30 mn*)
- Documentation du code (*Carlo et Ugo, tout au long de l'écriture du code, 240 mn*)

Le diagramme de Gantt correspondant est présenté ci-dessous.

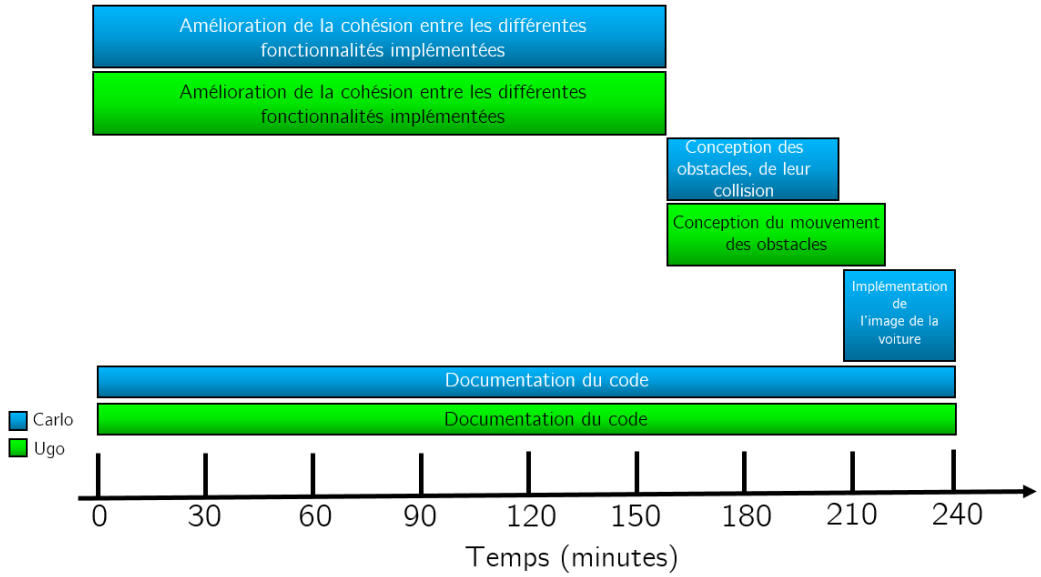
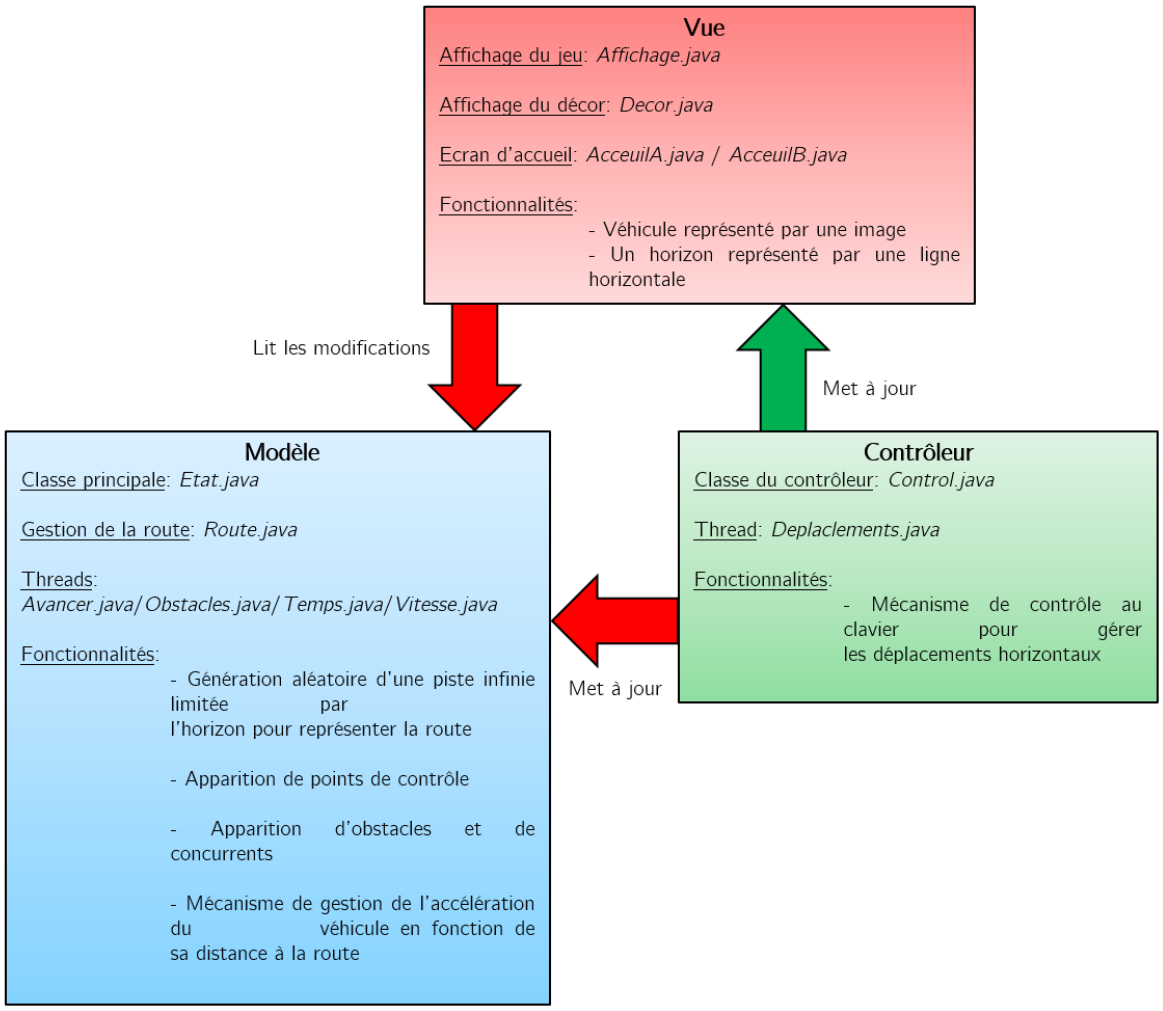


Figure 4: Diagramme de Gantt du travail effectué pendant les vacances

IV - Conception générale



V – Conception détaillée

- Implémentation de l'interface graphique :

  Les classes concernées par cette fonctionnalité sont : *Affichage.java*, *Decor.java*, *AccueilA.java*, *AccueilB.java*.
  On utilise la classe *Affichage.java* pour l'interface graphique du jeu. Dans sa méthode *paint* héritée de la classe *JPanel* on appelle un ensemble de sous méthodes, chacune chargée de l'affichage d'un élément de l'interface :

  - *paintRoute* : affiche la route. On récupère les points visibles grâce à la méthode getRouteGauche de la classe *Route*. Cette méthode renvoie les points de l'extrémité gauche de la route ; pour afficher également ceux de l'extrémité droite, on utilise de l'attribut *Ecart* de la classe *Route* représentant la largeur de la route. Un point est considéré comme visible lorsque le point qui le succède est situé au-dessus la hauteur de l'écran moins la valeur de l'attribut score de la classe *Route* qui est incrémenté par la méthode *setScore* de la classe *Route* appelée par la classe *Avancer*

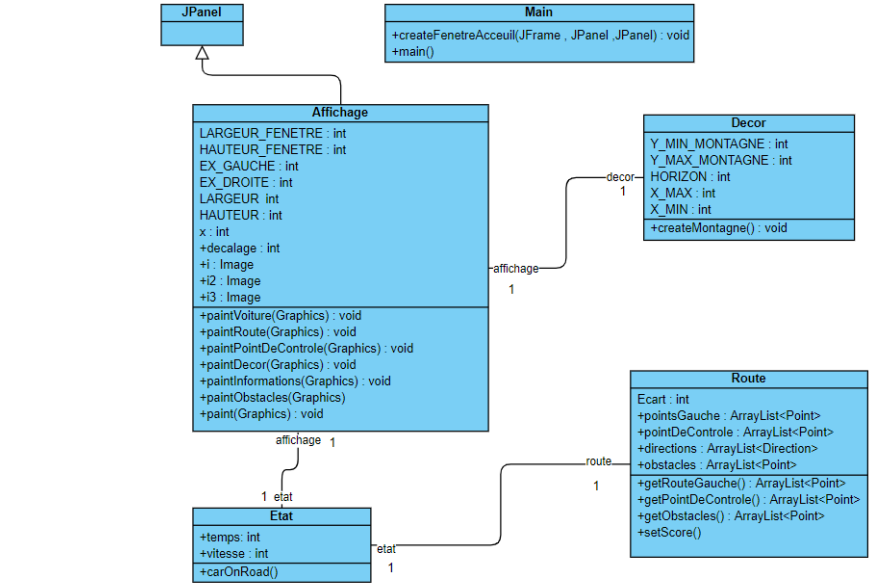
(un Thread). Cette incrémentation du score sert à représenter le défilement vers le haut de la route. Lorsqu'on un point de la route n'est plus considéré comme visible on le retire de la liste représentant les points de la route *pointsGauche* qui est un attribut de la classe *Route*. Ensuite pour gérer le défilement horizontal de la route on utilise l'attribut *decalage* de la classe *Affichage* qui est incrémenté par les méthodes de la classe *Etat* chargée de modifier la valeur de l'ordonnée de la voiture. L'attribut *decalage* est ajouté aux abscisses des différents éléments à ajouter. Cette méthode n'est pas utilisée que pour l'affichage de la route mais aussi pour les autres sous méthodes d'affichage.

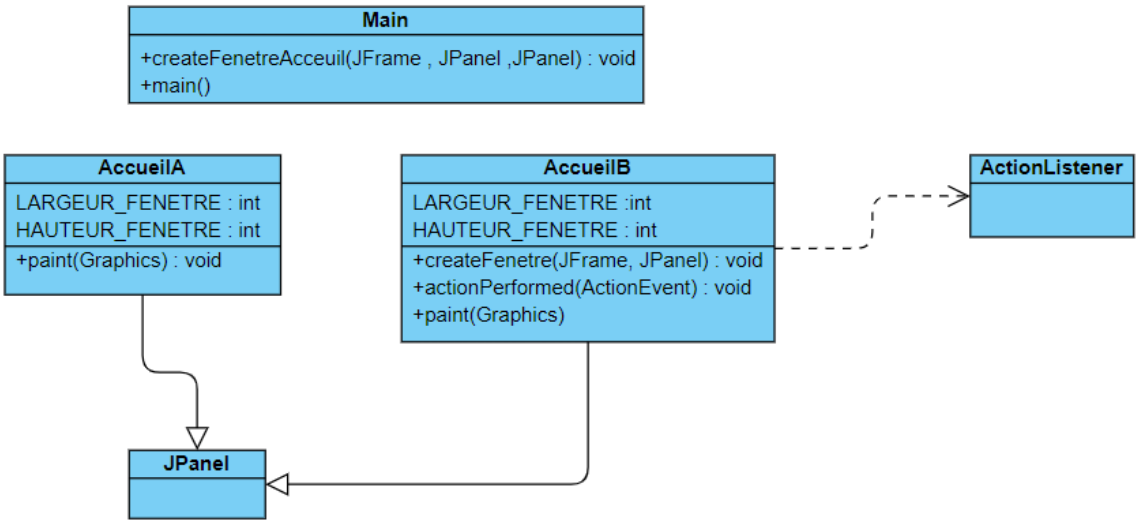
- *paintDecor* : affiche le décor. Les éléments du décor sont : les montagnes et la ligne d'horizon. Ces deux éléments sont gérés par la classe *Decor* au sein de laquelle on retrouve : la constante *HORIZON* représentant la ligne de l'horizon ; et la méthode *createMontagne* qui génère aléatoirement des points dont les valeurs de leurs abscisses sont bornées par les constantes de classe *X_MAX* et *X_MIN*, et dont les valeurs de leurs ordonnées sont bornées par les constantes de classe *Y_MIN* et *Y_MAX*. Les différents points représentant la montagne sont stockés au sein de l'attribut *pointList* de la classe *Decor*.

  Pour délimiter la distance minimale et maximale que la voiture peut parcourir horizontalement on a décidé d'utiliser deux lignes verticales situés à chacune des deux extrémités et que le joueur ne peut pas franchir. On utilise la méthode *clearRect* pour enlever l'affichage de la route là où se trouve l'affichage du décor.
- *paintInformations* : affiche les informations relatives à la partie de joueur. Le temps, représenté par l'attribut temps de la classe *Etat*. La vitesse, représentée par l'attribut vitesse de la classe *Etat*. Le score, dont la valeur est renvoyée par la méthode *getScore* de la classe *Route*. Les informations sont représentées sous forme de carrés noirs situés aux extrémités de l'écran.
- *paintObstacles* : qui affiche les obstacles récupérés par la méthode *getObstacles* de la classe *Route*. On affiche les obstacles grâce à la méthode *drawOval*.
- *paintVoiture* : affiche la voiture à partir des valeurs données par les constantes de classe : *HAUTEUR_FENETRE* (hauteur de la fenêtre) ; *HAUTEUR* (hauteur de la voiture) ; *x*, l'abscisse de la voiture. La voiture est représentée par une image contenue dans les attributs *i*, *i2*, *i3* de type *Image*. Les images sont situées dans le package *vue* et on les récupère grâce à la méthode *getImage* et à la classe *Toolkit* dans le constructeur. On choisit l'image en fonction de la direction de la route, on récupère cette information grâce à la liste *directions* qui est un attribut de la classe *Route*. Si la voiture n'est pas sur la route on montre que l'image où la direction de la route est droite. On vérifie que la voiture est sur la route grâce à la méthode *carOnRoad* de la classe *Etat*. La voiture reste au milieu de l'écran

tout le long de la partie, ce sont les autres éléments de l'interface qui changent leur position.

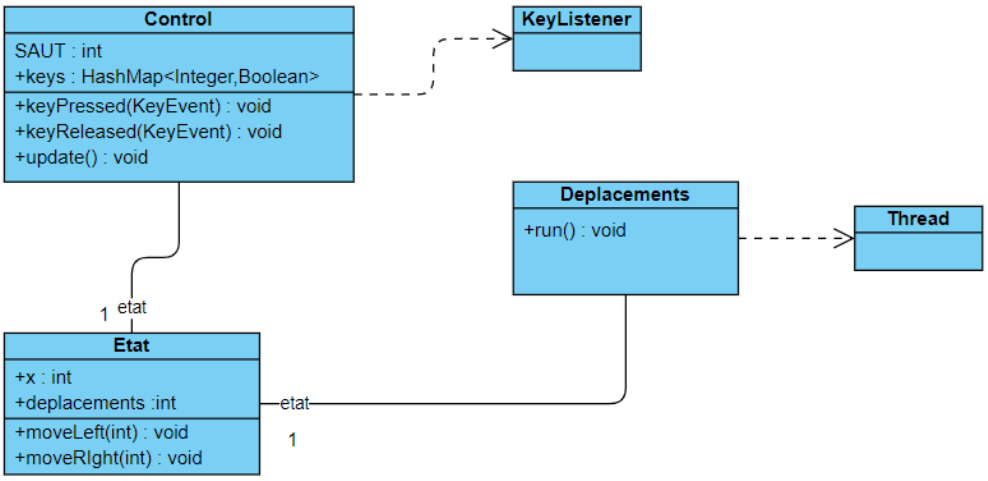
On utilise les classes *AcceuilA* et *AcceuilB* pour l'interface graphique de l'accueil. L'écran d'accueil est séparé en 2, la classe *AcceuilA* se charge de la partie haute, et la classe *AcceuilB* se charge de la partie basse. *AccueilA* et *AcceuilB* hérite de la classe *JPanel* et *AcceuilB* implémente la classe *ActionListener*. On utilise *ActionListener* pour implémenter des boutons à partir desquels le joueur choisit la difficulté du jeu. Lorsque le joueur appuie sur le bouton la version du jeu correspondante se lance et la gestion du lancement de la partie est effectuée grâce à la méthode *actionPerformed* héritée de la classe *ActionListener* qui lance les Thread et appelle la méthode *createFenetre*. Les instances de *AcceuilA* et *AcceuilB* sont appelles dans le *main* de la classe *Main* grâce à la méthode *createFenetreAcceuil*.





- Gestion des déplacements de la voiture :

L'état de la voiture et de la partie sont contenues au sein de la classe *Etat* du modèle. L'attribut *x* de la classe *Etat* représente la valeur actuelle de l'abscisse du joueur. La constante *SAUT* de la classe *Controle* représente la valeur d'un déplacement de la voiture. La classe *Controle* implémente la classe *KeyListener* et est donc chargée de répondre à l'interaction du joueur avec le clavier. Pour cela on se sert de l'attribut *keys* de la classe *Control* de type *HashMap<Integer,Boolean>* qui associe à chaque touche avec laquelle le joueur peut interagir (les flèches directionnelles droite et gauche) un booléen. Si le joueur appuie sur la touche alors son booléen est mis à vrai grâce à la méthode *keyPressed* de la classe *Control* et si le joueur relâche la touche alors son booléen est mis à faux grâce à la méthode *keyReleased* de la classe *Control*. Ces deux méthodes sont appelées par la méthode *update* de la classe *Control* elle-même appelée par la méthode *run* du thread *Déplacements*. On a décidé de partir sur cette implémentation pour éviter qu'il y ait des pauses lors des changements de direction et donc pour rendre le jeu plus fluide.
La modification de l'abscisse de la voiture est effectuée grâce aux méthodes *moveLeft* et *moveRight* de la classe *Etat* qui changent la valeur de l'attribut *x*, en vérifiant que les bornes mentionnées auparavant ne sont pas franchies. Ce sont ces méthodes qui sont chargées aussi de changer la valeur de l'attribut *decalage* de la classe *Affichage*. Ces deux méthodes sont appelées au sein de la méthode *update*.



- Gestion de la vitesse :

Il faut qu'on traite séparément le calcul de l'accélération et le calcul de la vitesse, représentés respectivement par les attributs *acceleration* et *vitesse* de la classe *Etat*. Lorsque l'accélération est constante positive alors la vitesse augmente jusqu'à la limite fixée par la constante *VITESSE_MAX* de la classe *Etat* ; et lorsque l'accélération est constante négative alors la vitesse diminue jusqu'à atteindre la limite 0 qui conduit à la fin de la partie. L'accélération est positive lorsque la voiture est sur la route et elle négative lorsque la voiture n'est plus sur la route.

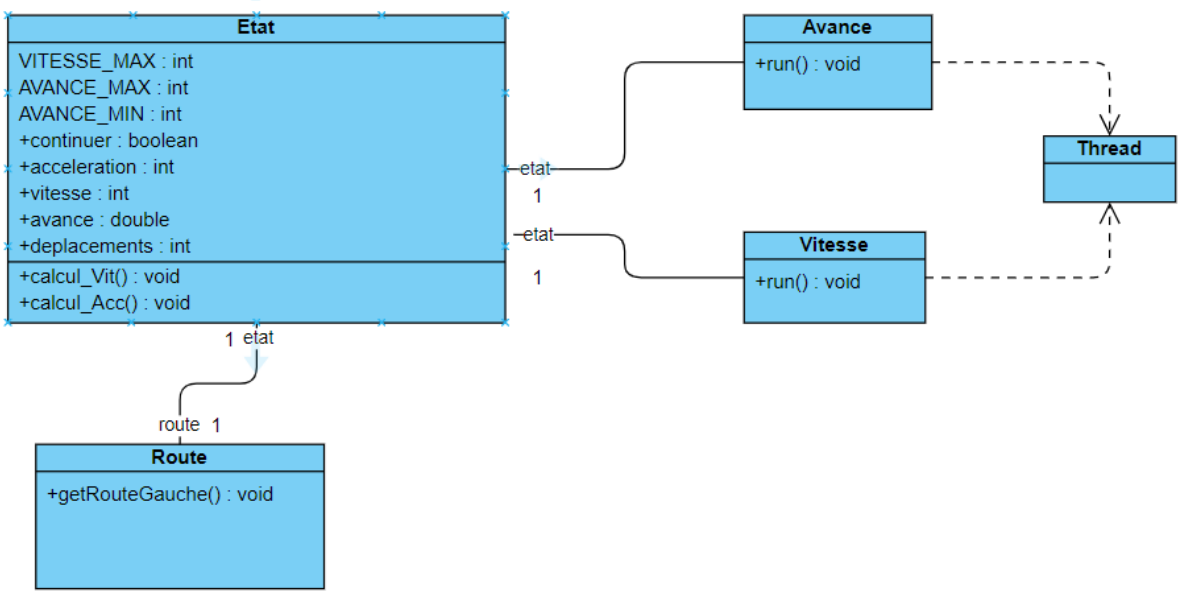
- Calcul de l'accélération :

  Ce calcul est effectué par la méthode *calcul_Acc* de la classe *Etat*. On récupère les points de la route grâce à la méthode *getRouteGauche* de la classe *Route*. On récupère les ordonnées des points qui encadrent la voiture pour ensuite, grâce au calcul de la pente, récupérer l'abscisse des deux extrémités de la route situées à la même ordonnée que la voiture. Si auparavant la voiture était sur la route et l'est toujours alors on renvoie 1 ; sinon si auparavant la voiture était en dehors de la route et elle l'est toujours on renvoie -1. Sinon s'il y a eu un changement d'état (auparavant pas sur la route et maintenant si, ou l'inverse) on renvoie 0.

- Calcul de la vitesse :

  Avant de détailler le calcul il faut qu'on introduise un nouvel attribut de la classe *Etat* : *avance*. Si la vitesse de la voiture augmente alors la vitesse de défilement de l'interface graphique augmente aussi ; et si la vitesse de la voiture diminue alors le défilement de l'interface graphique diminue aussi. On utilise l'attribut *avance* pour représenter ces changements de vitesse du défilement de l'écran. Cet attribut est borné par les constantes *AVANCE_MAX* et *AVANCE_MIN* de la classe *Etat*. Cet attribut sera passé en paramètre à la méthode *sleep,* elle-même appelée par la méthode *run* des classes *Avancer*, chargée du défilement de l'écran en modifiant la valeur de l'attribut score de la classe *Route* ; et *Vitesse*, utilisée pour appeler la méthode *calcul_Vit* de la classe *Etat*. *Avancer* et *Vitesse* implémente la classe Threads.
  Le calcul de *vitesse* et *avance* est effectué par la méthode *calcul_Vit*. Pour commencer on récupère l'entier envoie par la méthode *calcul_Acc* et on calcule la nouvelle vitesse de la façon suivante : NouvelleVitesse = vitesse + (calcul_Acc*2) ;
  Ensuite on vérifie que cette nouvelle vitesse respecte les bornes imposées, si c'est le cas on change la valeur de l'attribut *vitesse* à NouvelleVitesse ; sinon on donne à *vitesse* la valeur de la borne dépassées. On vérifie aussi si la vitesse de la voiture a diminué ou augmentée ; si elle a diminué alors on incrémente la valeur de *avance* sinon on décrément la valeur de *avance*. On vérifie que la nouvelle valeur de l'attribut *avance* respecte les bornes imposées.
  Si la vitesse de la voiture augmente alors la vitesse de déplacement horizontal augmente aussi, et si la vitesse de la voiture diminue alors la

vitesse de déplacement horizontal de la voiture diminue aussi. L'attribut *deplacements* de la classe *Etat* représente la vitesse de déplacement horizontal de la voiture. Cet attribut sera appelé par la méthode *sleep*, elle-même appelée par la méthode *run* de la classe *Déplacements*. Sa valeur dépend de celle de l'attribut *avance*.
Si la vitesse devient nulle alors on arrete la partie et on affiche un message grâce à la classe *JOptionPane*, et on met l'attribut *continuer* à false ce qui conduit à un arrêt de tous les Threads.



- Conception de la route :

  Pour l'implémentation de la route on utilise l'algorithme suivant :
  - On va commencer par fournir les attribut de classe *Route* utilisés : *pointsGauche*, qui représente la liste de points de la route et que l'on va appeler List dans le pseudo-code ; *objy*, qui représente l'ordonnée du prochain point qui va être ajouté à List ; *limite*, qui est utilisée pour savoir si les limites aux extrémités donnée par les attributs *xmin* et *xmax* sont atteints ; *direction*, qui représente la direction actuelle de la route ; *x1*, qui représente l'abscisse du dernier point ajouté ; et *y*, qui représente l'ordonnée du dernier point ajouté (On ne va pas traiter l'ajout d'obstacles pour l'instant).

- *Pseudo-code* :

```
Init () : void
        On ajoute le premier point à List avec les coordonnées x1 et y initialisée
        dans le constructeur
        objy ← y – 100
        Tant que objy > abscisse de l'horizon
        (on ajoute des points au-delà de l'horizon pour prévoir l'accélération du
        défilement de la route)
                Méthode pour ajouter un point -> sous_init(): void
                limite ← faux
                Si direction := Droit
                        y ← objy
                        On ajoute à List le point à la coordonnée x1 et y
                Sinon Si direction := Droite
                        x1 ← x1 + (y - objy)
                        y ← objy
                        Si x1 + Ecart >= xmax
                                x1 ← xmax – Ecart
                                limite ← vrai
                                direction ← Gauche
                        Fin Si
                        On ajoute à List le point à la coordonnée x1 et y
                Sinon Si direction := Gauche
                        x1 ← x1 – (y-objy)
                        y ← objy
                        Si x1 <= xmin
                                x1 ← xmin
                                limite ← vrai
                                direction ← Droite
                        Fin Si
                        On ajoute à List le point à la coordonnée x1 et y
                Fin Si
                objy ← y – (valeur tirée au hasard entre ymin et ymax)
                Si limite := faux
                        r ← valeur tirée au hasard entre 0 et 4
                        Si r < 3
                                direction ← Droit
                        Sinon Si r := 3
                                direction ← Droite
                        Sinon
                                direction ← Gauche
                        Fin Si
        Fin de la méthode pour ajouter un point
```

- Conception des points de contrôle :

  - Conception des points de contrôle
    On va commencer par fournir les attributs de la classe *Route* utilisés : *pointsGauche*, qui représente la liste de points de la route que l'on va appeler *ListRoute* ; *pointDeControle* qui représente la liste de point de contrôle et que l'on va appeler *ListPDC* ; compteur, qui représente l'ordonnée du nouveau point de contrôle ; *sous_compt* , qu'on utilise pour savoir quand est-ce qu'on doit modifier la distance entre deux points de contrôle ; *add* qui représente la distance entre deux point de contrôle ; *xPrecedent* et *yPrecedent* qu'on utilise pour le calcul des coordonnées du nouveau point de contrôle grâce au calcul de la pente.

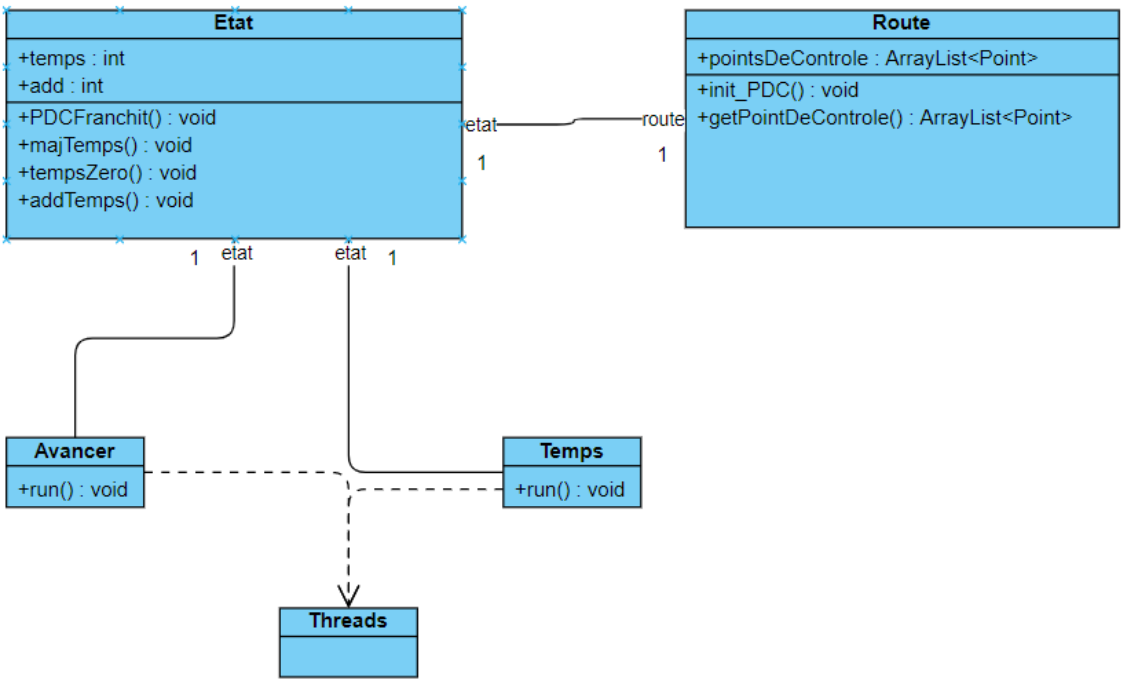
  - *Pseudo-code* :

```
init_PDC() : void
        On parcourt tous les points P de ListRoute, en commençant par le
        deuxième, grâce à une boucle for :
                Si l'ordonnée du point P est inférieure ou égale à compteur

                On calcule l'abscisse du nouveau point PDC de ListPDC
                grâce au calcul suivant

        P(xP,yP)
        Pprecedent(xPrecedent,yPrecedent)
        PDC(xPDC,compteur)
        Pente ← (yP-yPrecedent)/(xP-xPrecedent)
        Pente ← (compteur - yPrecedent)/(xPDC - xPrecedent)
        xPrecedent ← xPrecedent + ((compteuryPrecedent)/Pente)
        On ajoute PDC à ListPDC
        On prépare l'ajout du nouveau point de contrôle :

        sous_compt ← sous_compt+1
        Si sous_compt := 5
                sous_compt := 0
                Si add =\= 1000
                        add ← add + 100
                Fin du Si
        Fin du Si
        compt ← compt + add
        Fin du Si
        xPrecedent ← xP
                yPrecedent ← yP

        Fin de la boucle for
```

- Conception du temps et gestion du franchissement des points de contrôle :

Le temps est représenté par l'attribut *temps* de la classe *Etat*. Les méthodes de la classe Etat qui s'occupent de la gestion du temps sont :

- *majTemps* : qui met à jour l'attribut *temps* en lui soustrayant 0.0025 ; ce qui d'après des tests correspond à ce qu'on lui enlevé une milliseconde. La méthode est appelée toutes les millisecondes par la méthode *run* de la classe *Temps* qui implémente *Threads*. Le calcul est effectué tant que *temps* n'est pas nul, pour vérifier cela on appelle *tempsZero*. Si *tempsZero* renvoie *false* alors l'attribut *continuer* de la classe *Etat* est mis à *false* et un message de fin de partie est affiché grâce à la classe *JOptionPane*.
- *tempsZero* : qui renvoie *true* si *temps* est strictement supérieur à 0. Sinon la méthode renvoie *false* et l'attribut *temps* est mis à 0.
- *addTemps* : qu'on utilise pour incrémenter l'attribut *temps* lorsqu'un point de contrôle est franchi. On incrémente l'attribut *temps* grâce à l'attribut *add* qu'on décrémente progressivement jusqu'à arriver à une valeur minimum qu'on a fixé à 0.05.
- *PDCFranchit* : qui vérifie si un point de contrôle a été franchi. Pour cela on récupère les points grâce *à getPointDeControle* de la classe *Route*. Ensuite on vérifie si l'ordonnée de la voiture est à la même hauteur du point de contrôle et si la voiture est sur la route grâce à la méthode *carOnRoad* de la classe *Etat*. Si le point de contrôle est franchi, alors : on appelle la méthode *addTemps* de la classe *Etat* ; on retire le point de contrôle correspondant de la liste *pointsDeControle* de la classe *Route* ; et on ajoute un nouveau point de contrôle grâce à la méthode *init_PDC* de la classe Route.
  La méthode *PDCFranchit* est appelée dans la méthode *run* du *Thread Avancer*.



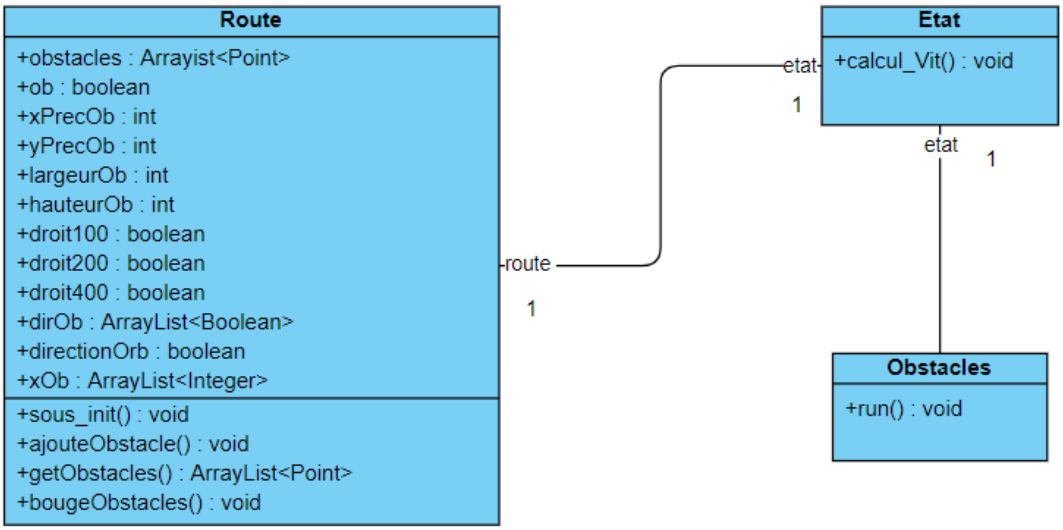
- Conception des obstacles et gestion de leur collision :

  - L'initialisation et l'ajout d'obstacles sont gérés dans la classe *Route* Les attributs de cette classe qu'on utilise sont :
    - *obstacles*, la liste d'obstacles
    - *ob*, qu'on utilise pour l'ajout d'un nouvel obstacle
    - *xPrecOb*,qui représente l'abscisse du dernier obstacle ajouté
    - *yPrecOb*, qui représente l'ordonnée du dernier obstacle ajouté
    - *largeurOb*, qui représente la largeur d'un obstacle
    - *hauteurOb*, qui représente la hauteur d'un obstacle
    - *droit100/droit200/droit400* : qui sont les 3 valeurs préfixées pour la longueur de la route lorsque sa direction est DROIT.
    - *dirOrb* : la liste qu'on utilise pour connaitre la direction horizontale de chaque obstacle.
    - *directionOrb* : qu'on utilise pour initialiser la direction horizontale des obstacles qu'on ajoute.
    - *xOb* : qu'on utilise pour connaitre la valeur minimum de l'abscisse des obstacles
  - Pour l'initialisation et l'ajout d'obstacles on utilise la méthode *ajouteObstacle* de la classe *Route*. On n'ajoute des obstacles que si la route est droite pour ne pas entraver la jouabilité. On place les obstacles aléatoirement et on ajoute les informations relatives à cet obstacle aux attributs chargée de son déplacement horizontal.
  - Pour la mise à jour de la position des obstacles on utilise la méthode bougeObstacles de la classe Route qui utilise l'algorithme suivant :

  - *Pseudo-code* :

```
bougeObstacle() : void
        Pour chaque point P(xP,yP) de la liste d'obstacles
                Si dirOrb(p) := true (si l'obstacle se déplace vers la gauche)
                        x ← xP-1
                        Si x <= xOb(p) (si on a atteint ou dépassé l'extrémité gauche
de la route)
                                dirOrb(p) ← false
                                x ← xOb(p)
                        Fin Si
                        obstacles(p) ← obstacles(NewP(x,yP))
                Sinon
                        x ← xP+1
                        Si x >= xOb(p) + Ecart ( si on a atteint ou dépassé l'extrémité
droite de la    route)
                                dirOrb(p) ← true
                                x ← xOb(p) + Ecart - largeurOrb
                        Fin Si
                        obstacles(p) ← obstacles(NewP(x,yP))
                Fin Si
Fin de la méthode
```

- Pour la gestion de la collision on l'effectue au sein de la méthode *calcul_Vit* de la classe *Etat*. On récupère les obstacles grâce à la méthode *getObstacles* de la classe *Route*. S'il y a eu collision alors on décrémente l'attribut *vitesse* de la classe *Etat* d'une valeur constante, et on incrémente l'attribut *avance* de la classe *Etat* d'une valeur constante.



VI - Résultats

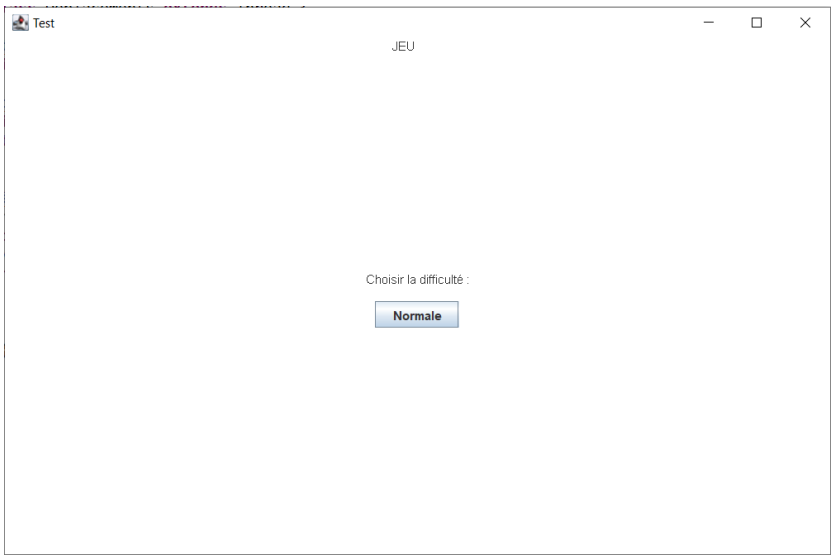


Figure 3 : Ecran d'accueil

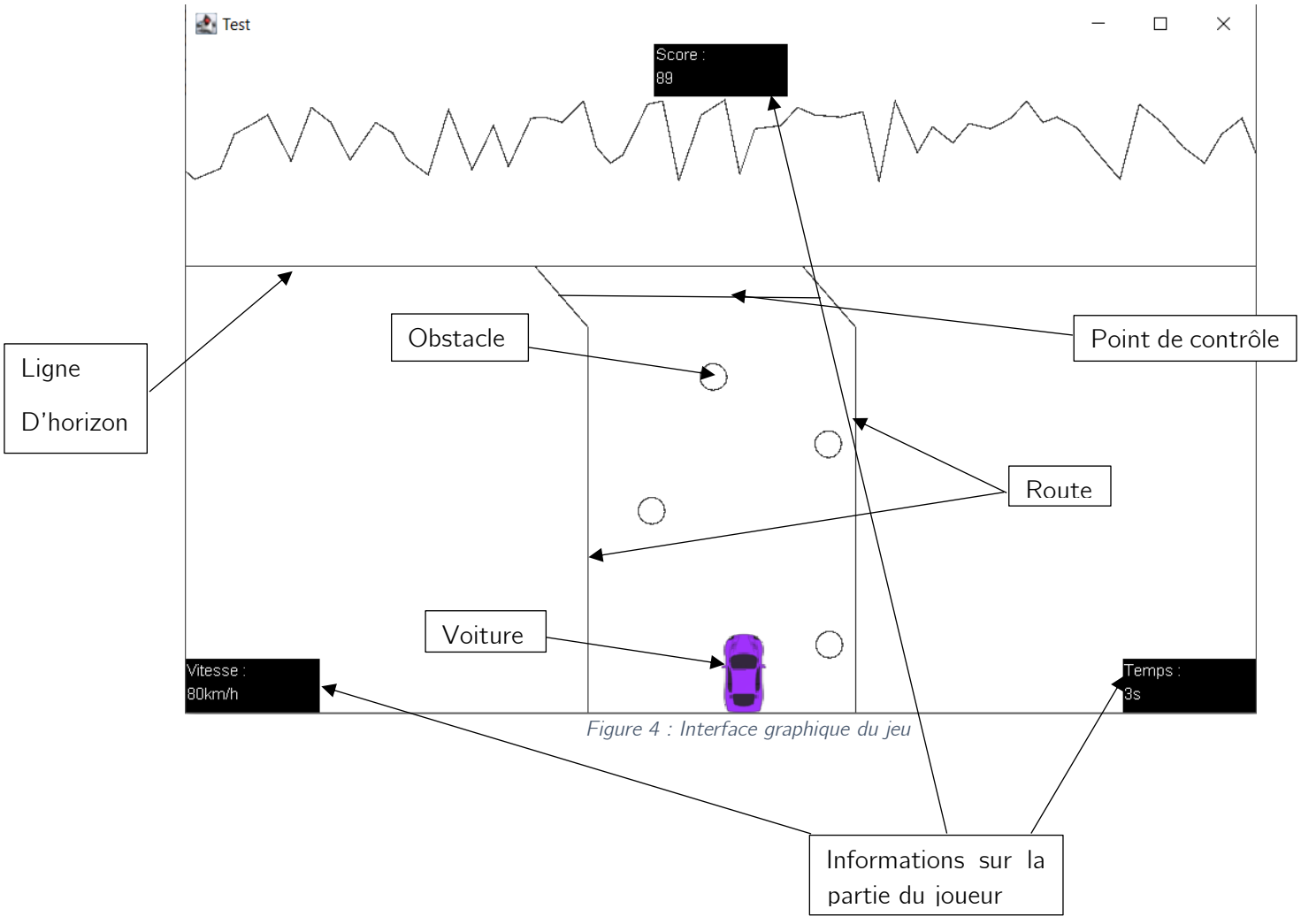


Figure 4 : Interface graphique du jeu

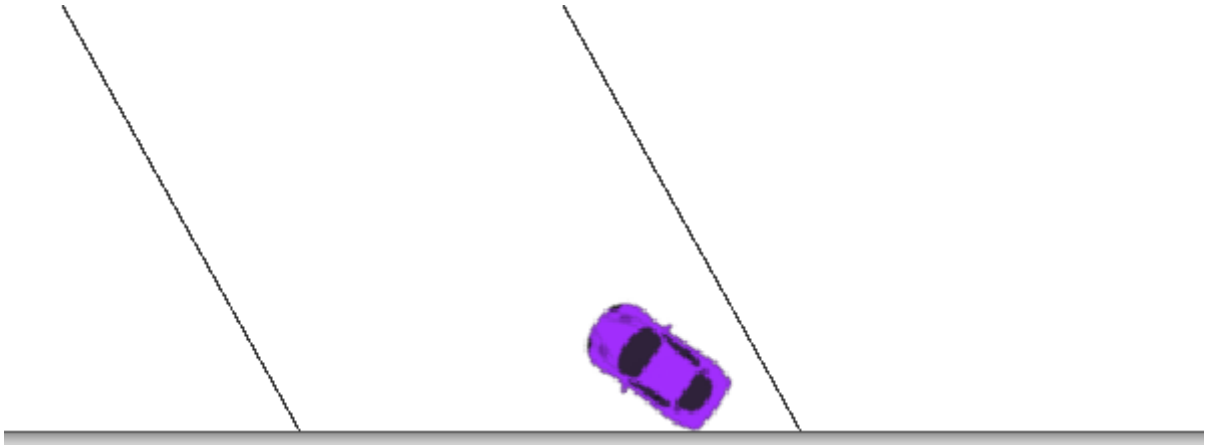


Figure 5 : affichage de la voiture lorsqu'elle se retrouve dans un virage

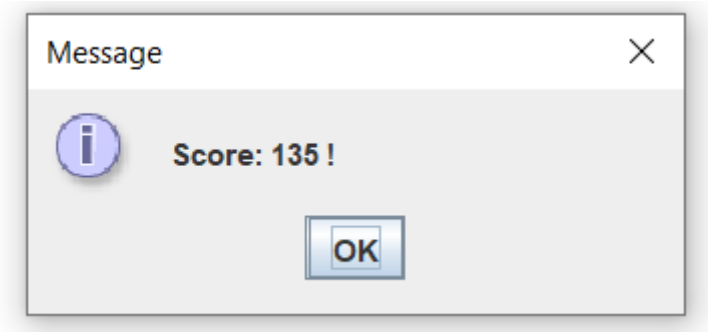


Figure 6 : Message qui s'affiche lorsque le joueur a perdu la partie

VII – Documentation utilisateur

Voici une explication détaillée des étapes à suivre pour jouer au jeu :

- 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`. Utilisez les flèches directionnelles gauche et droite pour vous déplacer.

VIII – Documentation développeur

Les classes principales du projet sont :

- *Etat.java* : qui implémente : la modification de la position de la voiture ; la gestion du temps et de la vitesse ; le franchissement d'un point de contrôle et la collision d'un obstacle ; les conditions de fin de partie.
- *Route.java* qui permet de modifier l'implémentation de la route et l'ajout de ses points, obstacles et points e contrôle. De plus cette classe s'occupe aussi de la mise à jour du score.
- *Control.java* : à laquelle on peut ajouter de nouvelles interactions avec le clavier.
- Si des modifications veulent être apportés à l'affichage toutes les classes du package *vue* peuvent être utiles.

Les autres classes sont soit des Threads, soit la classe *Main.java* dans laquelle se trouve la méthode *main* du projet.

Si des modifications veulent être apportées aux constantes du code : pour ce qui est de l'affichage le rôle des différentes constantes de chaque classe est clair ; pour ce qui est des modifications relatives à l'état de la partie (temps, vitesse, obstacles...) nous n'avons pas encore réussi à faire communiquer toutes ces constantes entre elles pour que le résultat final ne soit pas injouable. C'est pour cela aussi que nous avons décidé de partir sur un modèle où le joueur peut choisir sa difficulté à partir de paramètres que nous avons testés préalablement.

On va tout de même ici expliquer quelles sont ces constantes :

- *Saut* de la classe *Control.java* est la constante représentant la valeur du déplacement effectué par la voiture.
- *VITESSE_MAX* est l'attribut de la classe *Etat.java* représente la borne max de la vitesse.
- *AVANCE_MIN* et *AVANCE_MAX* sont des attributs de la classe *Etat.java* représentant les bornes max et min de la vitesse de défilement de l'écran.

- Les valeurs initiales des attributs *vitesse* et *avance* de la classe *Route.java* peuvent aussi être modifiées dans le constructeur.
- *INCR* de la classe *Route.java* est la constante contenant la valeur qui incrémente le score à chaque fois que la méthode *setScore* de la même classe est appelée.

Les fonctionnalités qu'il reste à implémenter sont des fonctionnalités qui servent à rendre le jeu plus divertissant et cohérent esthétiquement, comme l'ajout d'adversaires qui auraient une vitesse et une position sur la route fixée, ces adversaires devraient apparaitre aléatoirement lors du défilement vertical de la route et ils pourraient augmenter le score du joueur lorsque celui-ci les dépasse.

IX – Conclusion et perspectives

Nous avons réussi à implémenter toutes les fonctionnalités essentielles du jeu. Il nous reste à trouver un moyen pour que toutes ces fonctionnalités communiquent entre elles de façon cohérente et que l'une ne vienne pas entraver le fonctionnement d'une autre.
La principale difficulté a été d'implémenter les points de contrôle qui au début conduisaient toujours à une erreur lors de l'exécution du programme. Maintenant c'est la fonctionnalité qui déplace les obstacles qui quelques fois conduit à une erreur qui bloque le mouvement de tous les obstacles du jeu.