RAPPORT PCII

YOHANN BLACKBURN
SAMIR RABIAI

GROUPE 1

1. Introduction

Dans ce projet nous allons implémenter un jeu de course de moto. Le jeu se déroule de la façon suivante : le joueur utilisera les touches de son clavier (flèche gauche et droite) pour diriger le véhicule. Si ce dernier est sur la piste, il accélèrera graduellement mais s'il sort de celle-ci, il se mettra à ralentir jusqu'à ne plus bouger, ce qui marquera une fin de partie. Le joueur devra éviter les différents obstacles de la course ainsi que les concurrents afin de pouvoir arriver au prochain checkpoint avant la fin du chrono, et s'il n'y arrive pas, cela marquera aussi une fin de partie.

Voici une capture d'écran d'un jeu qui possède à peu près la même philosophie, nous devrions obtenir une interface graphique contenant : notre score, notre vitesse, les obstacles, les concurrents, notre véhicule, ainsi que les décors de notre modèle qui réagissent aux interactions du joueur.

Fig. 1 : Capture d'écran d'un jeu de course de voiture en 3D isométrique

2. Analyse globale

L'implémentation de ce projet se fera sur une durée de sept semaines durant laquelle nous implémenterons les différentes fonctionnalités qui nous permettront d'obtenir un jeu plus ou moins similaire à celui montré ci-dessus. Le développement du jeu se fera en *Java* tout en suivant le modèle *MVC* qui nous permettra de structurer le projet en séparant bien ces fonctionnalités en trois parties distinctes : le modèle, la vue, et le contrôleur.

Maintenant que nous avons compris à quoi devrait ressembler le jeu, nous pouvons extraire les différentes fonctionnalités nécessaires à la création de celui-ci, par exemple, il nous faudra implémenter une interface graphique permettant d'afficher les différents composant de notre jeu, il nous faudra générer une piste infini ainsi que notre véhicule, implémenter les interactions que le joueur pourrait avoir le véhicule, les états initiaux et finaux du jeu, le calcul de l'accélération du véhicule, l'ajout d'un chronomètre (et du score) qui nous permettra de déterminer si le joueur a perdu (et nous montrera son score si c'est le cas), l'affichage du chronomètre et du score, et plusieurs autre petites ou grandes fonctionnalités qui seront listés dans la partie *Plan de développement.*

3. Plan de développement

Fonctionnalités nécessaires :

- Dessin de notre véhicule, tâche facile
- Dessin de l'horizon de du fond, tâche facile
- Implémentation et dessin d'une piste de course infinie, tâche moyenne
- Dessin de courbe (calcul de point de contrôle pour Bézier), tâche difficile
- Import et utilisation de sprite, tâche difficile
- Création de l'état initial du jeu, tâche facile
- Mise en place des déplacement du véhicule et des interactions du joueur avec le jeu, tâche facile
- Création de véhicules concurrents dans notre modèle, tache plutôt facile
- Affichage des véhicules concurrents, tâche facile
- Détection de collision avec les concurrents, tâche moyenne
- Mise en place du chronomètre et du score + affichage de ces derniers, tâche moyenne
- Méthode de mis a jour du temps restant, tâche moyenne
- Gestion de vitesse (sorties de route), tâche difficile
- Fonctionnalités de fin de partie (vitesse = 0 ou temps restant = 0), tâche difficile

Fonctionnalités complémentaires (Optionnelles, comme son nom l'indique mais quand même intéressantes) :

- Création et affichage d'obstacles aléatoires dans notre modèle, tâche facile,
- Détection de collision avec les obstacles, tache difficile,
- Mise à jour de l'affichage pour prendre en compte les collisions, tache facile,
- Changement dessin véhicule en fonction des actions faites, plutôt facile,
- Défilement du décors pour créer une sensation de virage, tache moyenne,
- Fonctionnalité du calcul d'accélération du véhicule, tâche facile
- Fonctionnalité pour faire monter le véhicule (modification de l'état du véhicule),
- Mettre à jour l'affichage, tache plutôt facile,
- Ajouter des décors autour de la piste, tache plutôt facile,
- Dessin de la piste avec prise en compte de la profondeur, tâche difficile,
- Ajout écran d'accueil, tache moyenne,
- Mémorisation des meilleurs scores, tache facile,
- Fonctionnalité de contrôle de véhicule, tache plutôt facile

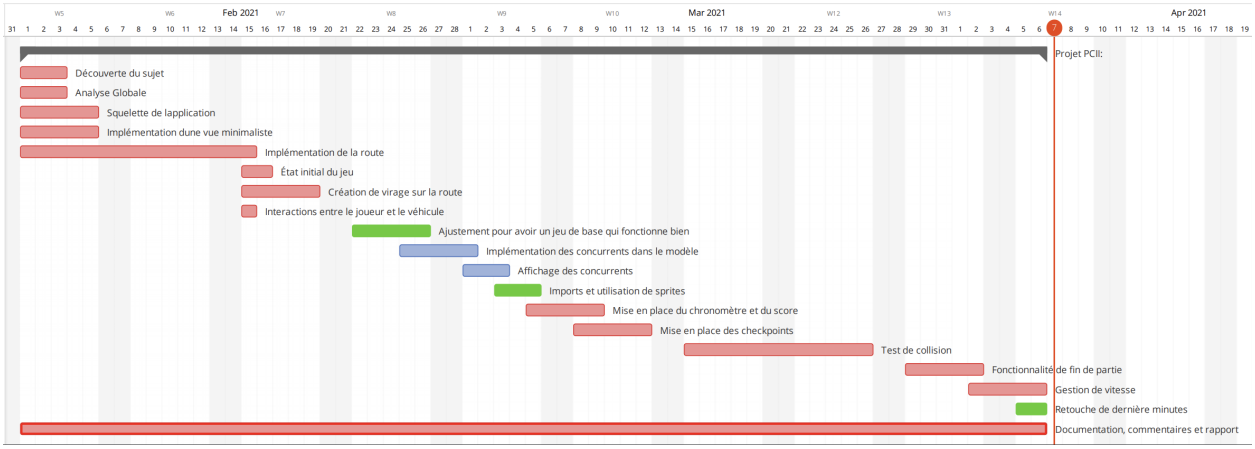


Fig. 2 : Diagramme de Gantt

4. Conception générale

Lors de ce projet, nous avons adopté le modèle *MVC* pour le développement de notre interface graphique. Celui-ci permettra de séparer nos différentes classes en trois packages distincts ce qui nous fournira une meilleure lisibilité ainsi qu'une séparation claire et nette de leur rôle dans ce projet.

Le package *Model,* contiendra, comme son nom l'indique, toutes les classe qui porteront sur le modèle, c'est-à-dire toutes les classes qui feront les calculs nécessaires au bon fonctionnement du jeu, celui-ci sera composé des classes :

- *Game*, qui sera notre classe principale, comportera la majorité des fonctionnalités nécessaire lié au déplacement du véhicule,
- *Car,* qui sera notre véhicule, comportera les fonctionnalités liées à la création de ce dernier,
- *Opponent,* qui implémentera nos concurrents, comportera les fonctionnalités liées à ces derniers,
- *Road,* qui sera notre piste, comportera les fonctionnalités liées à la création d'une piste infinie, et l'auto génération de point ainsi qu'une auto suppression de ces derniers,
- *ThreadAvancer,* qui sera une *Thread*, et nous permettra de faire avancer notre environnement (donnera l'illusion que la voiture avance).

Le package *View* contiendra toutes les classes qui nous permettront d'afficher sur une fenêtre, celui-ci sera composé des classes :

- *View,* qui se chargera de la création de notre fenêtre d'affichage,
- *ViewWorld,* qui se chargera de tous les calculs concernant l'affichage de notre véhicule, de notre route, de nos concurrents etc…,
- *ThreadAffichage,* qui sera chargé de tout simplement redessiner notre après un certain temps.

Le package *Controler* contiendra toutes les classes qui porteront sur les traitements des inputs du joueurs, celui-ci sera composé des classes :

- *Controler,* qui sera un *KeyListener* ce qui permettra de prendre en charge les différents inputs d'après le clavier du joueurs.

Le package ressource quant à lui, contiendra toutes les images/sprites dont nous aurons besoin pour faire notre jeu.

5. Conception détaillée

Pour la fenêtre contenant notre véhicule, les concurrents et le décors, nous allons utiliser l'API *Java Swing* et la classe *JPanel*. La classe *ViewWorld* nous permettra de définir les dimensions de la fenêtre, se sera dans celle-ci que nous ferons tous les calculs liés à l'affichage. La classe Game sera la classe principale de notre modèle, se sera dans celle-ci que nous ferons tous les calculs liés au jeu.

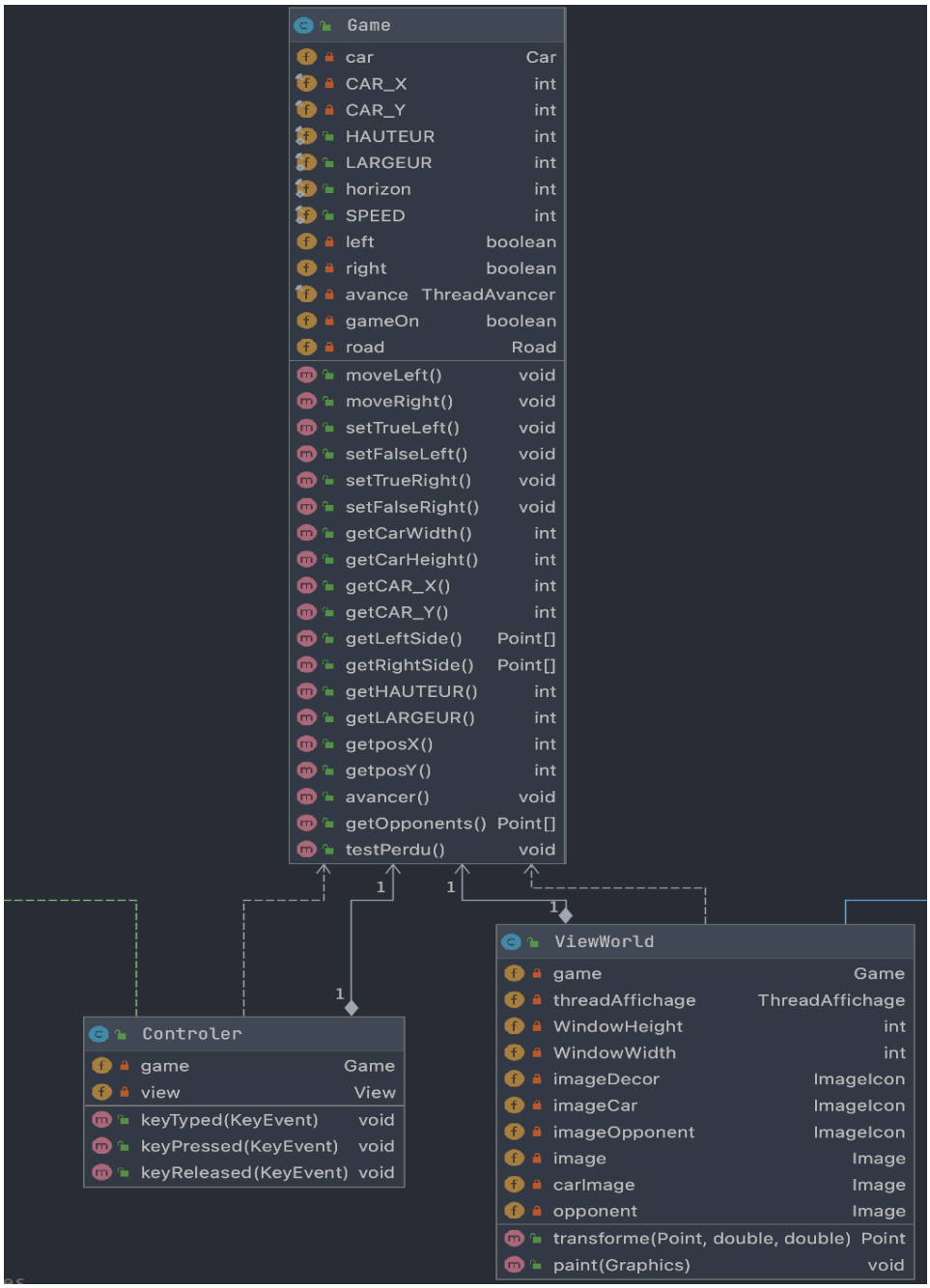


Fig. 3 : Diagramme de classe représentant les différentes classes du modèle MVC

Pour le déplacement de la voiture nous utiliserons de la programmation événementielle avec la classe *KeyListener* ainsi que des booléens définis dans la classe principale *Game*. En effet, la classe *Game* possédera deux booléens qui seront par défaut à *false* et lorsque le joueur appuiera sur la flèche de gauche ou de droite de son clavier, la classe *Controler* se chargera de mettre ces derniers à *true* a l'aide des methodes *setTrueLeft()* et *setTrueRight()* de la classe *Game*. Ces booléens seront utilisés dans les méthodes *moveLeft()* et *moveRight()* de la classe *Game*, ce qui permettra de bouger la voiture d'une certaine constante si ces booléens sont à true.

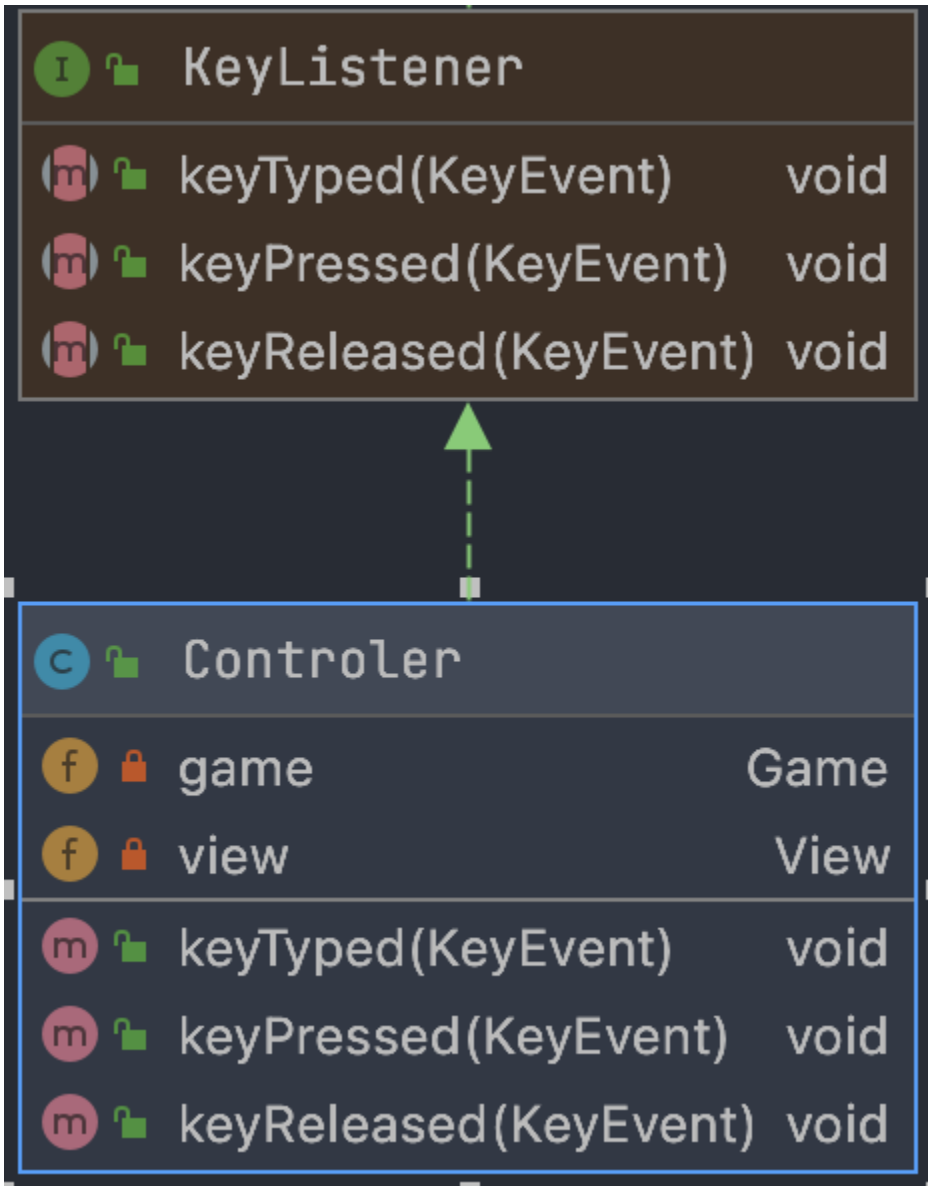


Fig. 4 : Diagramme de classe présentant l'implémentation de la classe KeyListener dans la classe Controler

La gestion du déplacement de la voiture sera en fait une illusion, la voiture ne bougera jamais en avant, nous ferons "*bouger le monde"* du haut de la fenêtre vers le bas pour donner l'impression au joueur que son véhicule avance à une certaine vitesse. Cette fonctionnalité sera implémentée dans la

classe *ThreadAvancer* qui hérite de la classe *Thread.* Elle prendra un paramètre l'état dans lequel se trouve le jeu, dont sa méthode principale sera *run()*. Elle permettra d'exécuter certaines tâches quand la thread sera démarrée notamment comme faire avancer constamment le véhicule ou encore de le faire bouger vers la droite ou la gauche dépendant de quel touches de son clavier le joueur utilise.

Fig. 5 : Diagramme de classe de l'implémentation de la classe ThreadAvancer

Pour la route, une classe *Road* a été créée. Celle-ci aura comme attributs deux *ArrayList* de *Point* qui nous permettront de dessiner deux lignes qui seront séparées par une constante *roadWidth* ce qui créera une route. On utilisera la constante *SPEED* de la classe *Game* pour faire avancer celle-ci à une certaine vitesse. Le constructeur *Road* nous permettras d'initialiser les points qui constitueront notre route dont le premier sera en y = 0. Ces points vont être reliés par la méthode *paint(Graphics g)* de la classe *ViewWorld* dans laquelle nous créerons la classe *QuadCurve2D* pour dessiner des courbes de bézier entre nos points avec le point de contrôle étant toujours égale à $(point[i+1].x + point[i].x) / 2$ en x et $(point[i+1].y + point[i].y) / 2$ en y.

Fig. 6 : Diagramme de classe de l'implémentation de la classe Road

Pseudo code l'initialisation de nos points :

```
Methode Road() :
      //Permet d'initialiser les points de la partie gauche de la route
      ajoute Point(centreRoad - roadWidth/2, 0) dans leftSide
      ajoute Point(100+rand(0,200), horizon) dans leftSide
      ajoute Point(100+rand(0,200), Game.HAUTEUR) dans leftSide
      ajoute Point(100+rand(0,200), Game.HAUTEUR+horizon) dans leftSide
      //Partie droite
      Pour tous les point p dans leftSide
            rightSide <- Point(p.x + roadWidth, p.y)
```

Pour nous permettre de ne pas avoir une route toute le temps droite mais aussi des virages jouables, la position en x de nos points doit varier, c'est pour cela que nous ajoutons toujours une nombre aléatoire entre 0 et 200.

La méthode *setPosY(int stop)* de *Road* nous permet de faire 'avancer' notre voiture. En effet, nous allons rajouter une certaine constante *SPEED* défini à notre position en y. Cette méthode sera appelée dans une méthode de la classe *Game* : *avancer()* qui sera à son tour appelée dans la méthode *run()* de la classe *ThreadAvancer*. *stop* nous servira bien plus tard dans notre programme, son but est de gérer la variation de vitesse de la voiture en fonction de sa sortie (ou non) de la route.

Notre route va normalement devoir être infinie. Cette fonctionnalité sera implémentée dans la méthode *updateRoad(posY)* de la classe *Road*. Elle sera appelée dans les méthodes *getLeftSide(posY)* et *getRightSide(posY)* de la classe *Road*. Cette méthode nous permettra de générer des points si besoin et de les supprimer si un point n'est plus visible.

Peusdo code de *updateRoad(posY) :*

```
Methode updateRoad(posY) :
```

```
Si leftSide.get(2)-posY <= 0 alors
      retire leftSide(0)
      Retire rightSide(0)
Si leftSide.get(leftSide.size-2)-posY <= horizon alors
      X <- 100+rand(0,200)
      Rajoute point(x, leftSide.get(leftSide.size-1).y + horizon)
        dans leftSide
      Rajoute point(x+roadWidth, leftSide.get(leftSide.size-1).y +
        horizon) dans leftSide
```

Pour nous permettre d'avoir un jeu un peu plus intéressant, nous avons implémenté une classe *Oppenent* qui nous permettra de 'faire la course' contre des concurrents. La classe *Car* contiendra une *ArrayList* d'*Oppenent* que l'on mettra à jour au fur et à mesure que nous jouerons, en effet, la méthode *updateOpponents()* se basera grandement sur le pseudo code ci-dessus.

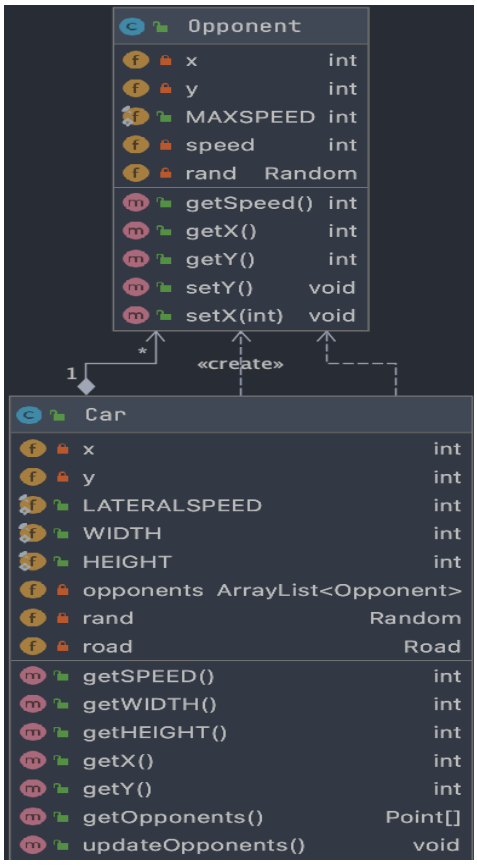


Fig. 7 : Diagramme de classe de l'implémentation de la classe Opponent

Pour la mise en place du chronomètre et du score, nous avons utilisé la classe *Timer* de *Java* ainsi que la classe *TimerTask*, nous avons donc initialisé dans la classe *Game* un attribut *timeLeft* de type *int*, un attribut *timer* de type *Timer* et un attribut *task* de type *TimerTask*. Ce dernier représente la tâche effectuée par *timer*, cette tâche consiste à décrémenter timer chaque seconde.
Code pour la tâche à effectuer (dans la déclaration du *TimerTask*):

```
Private TimerTask task = new TimerTask() {
      @Override
      Public void run(){
            timeleft--;
      }
};
```

Dans le constructeur de la classe *Game* :

```
timer.scheduleAtFixedRate(task,1000,1000)
```

scheduleAtFixedRate(TimerTask task, long delay, long period) execute à répétition la tâche *task* pendant période de temps avec un intervalle de valeur *delay*
Dans le futur, nous allons modifier task afin d'inclure la détection de sortie de route et le changement de vitesse de la voiture

L'attribut *timer* nous sert de compte à rebours, sa valeur est mise à jour à chaque fois que nous dépassons un checkpoint. La méthode UpdateTimer s'occupe de cette tâche :
Pseudo-code pour *updateTimer* :

```
Si checkpoint < 5 // Ordonnée du bas de notre voiture alors
      Incrémenter le nombre de checkpoint dépassés
      Si : nombre de checkpoint dépassés + 5 < maxTime/2 alors
            timeLeft=timeLeft+ ((maxTime/2) - nombreCheckPointsDépassé)
      Sinon
            timeLeft = timeLeft + 5 // Au minimum, on ajoute 5 secondes
```

Nous avons choisi d'implémenter le système de checkpoint dans la classe *Road*, car nous avons utilisé l'attribut *posY* présent pour mettre à jour l'ordonnée de ce dernier. *checkPointY* de type int est l'attribut qui représente la position du checkpoint.
La classe Game récupère grâce à la méthode *getCheckpointY()* de *Road*
Code de la methode *getCheckpointY() :*

```
public int getCheckpointY(){
      updateCheckPointY();
      Return checkpoint - posY;
```

```
}
```

updateCheckPointY() est une méthode qui vérifie si le checkpoint n'apparaît plus dans la fenêtre (est en dessous) et attribue une nouvelle valeur à *checkpointY*
Code de la fonction *updateCheckPointY()* :

```
Public void updateCheckPointY() {
      If (checkpointY - posY <= 0) {
            checkpointY = posY + 3*Game.HAUTEUR;
}
```

La partie est finie quand le timer est à 0 ou quand notre voiture touche une des voitures adverses. Ces conditions sont vérifiées dans la méthode *testPerdu()* présente dans la classe *Game*. La détection de collision se fait à partir du principe que chaque voiture est de forme rectangulaire. On vérifie donc si les extrémités de notre voiture sont entrées en contact avec les extrémités des voitures adverses. Le schéma ci-dessous illustrera plus clairement notre idée.

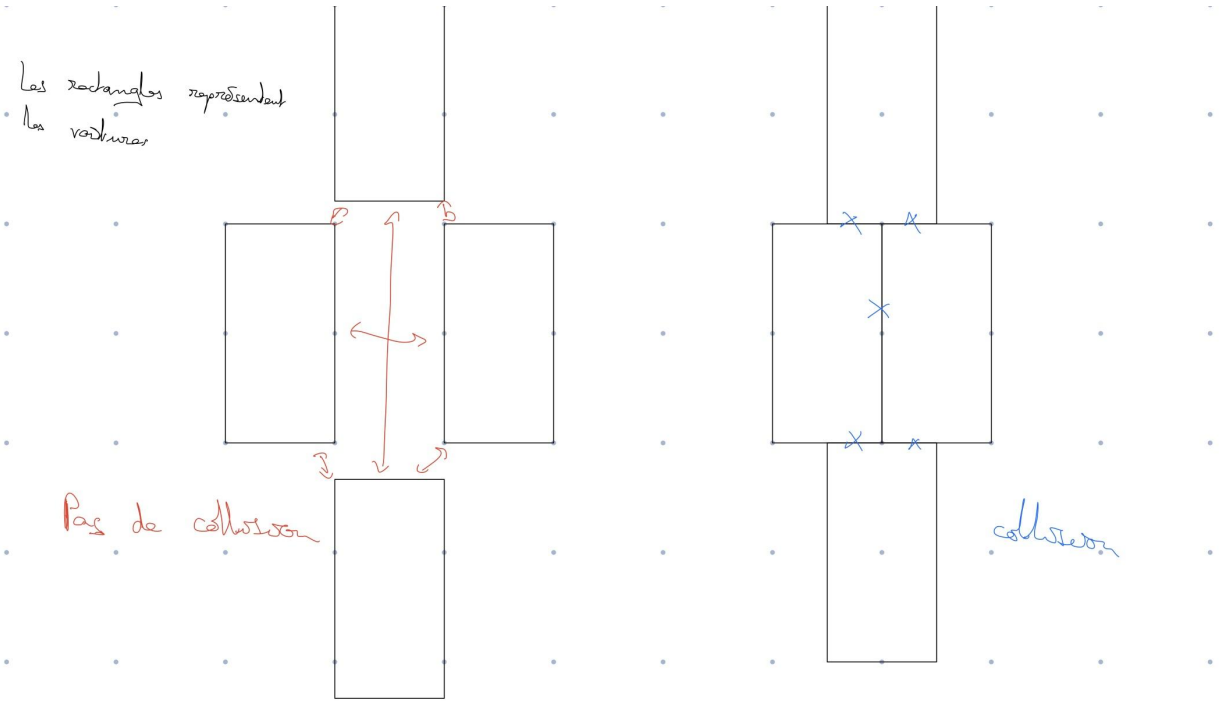
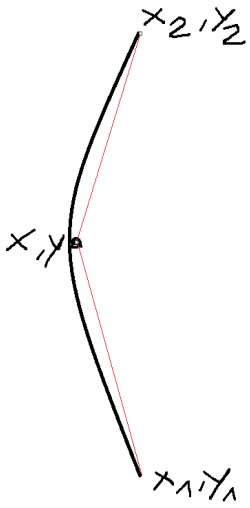


Fig. 8 : Schéma de détection de collision entre les véhicules

La méthode *testPerdu()* sera appelée dans la méthode *run()* de la classe *ThreadAvancer* ce qui nous permettra d'arrêter les opérations, en interrompant le *ThreadAvancer avance,* lorsque celle-ci détectera une collision ou verra qu'il n'y aura plus de temps restant sur le *timer.*

Enfin, la gestion de vitesse se fera dans la classe *Game* dans la méthode *checkIfOut()*. Le principe est simple : pour chaque paire de points consécutifs dans midLane de la classe Road, on crée un point

"d'ancrage" avec des coordonnées x = (x1 + x2) /2 et y =(y1 + y2)/2. Ceci va nous permettre de mieux nous rapprocher des courbures de la route.



Nous comparerons ensuite la position relative de notre voiture par rapport aux droites que nous avons créées. Si la voiture est de distance *roadWidth/2* ou plus de la droite alors nous ne sommes pas sur la route.

Pseudo-code de la méthode *checkIfOut()* :

```
Récupérer tableau de points de midLane
pt1= premier point du tableau
pt2 = deuxième point du tableau
midl = nouveau point (x = (pt1.x + pt2.x) /2, y = (pt1.y + pt2.y)/2)
Pour chaque point de midLane
        Si CAR_Y entre midl.y et pt1.y alors
                Si pt1.x < midl.x alors
                        Si CAR_X à droite de pt1.x-roadWidth/2 et à gauche de midl.x + roadWidth/2
                                Alors return false

                Si pt1.x > midl.x alors
                        Si CAR_X à droite de midl.x-roadWidth/2 et à gauche de pt1.x + roadWidth/2
                                Alors return false
        Si CAR_Y entre midl.y et pt2.y alors
                Si midl.x < pt2.x alors
                        Si CAR_X à droite de midl.x-roadWidth/2 et à gauche de pt2.x + roadWidth/2
                                Alors return false

                Si pt2.x > midl.x alors
                        Si CAR_X à droite de pt2.x-roadWidth/2 et à gauche de midl.x + roadWidth/2
                                Alors return false

        pt1 = midLane[i]
        pt2 = midLane[i+1]
        midl = nouveau point (x = (pt1.x + pt2.x) /2, y = (pt1.y + pt2.y)/2)
return true;
```

Cette méthode nous renvoie un booléen qui va nous permettre de contrôler la vitesse de notre voiture. Et ceci grâce à l'attribut *stop* (de type int) présent dans la classe Game. Dans notre attribut *task* (de type TimerTask). Nous allons rajouter une fonctionnalité qui vérifie la valeur rendue par la méthode *checkIfOut()* et qui incrémente ou décrémente *stop* en fonction du résultat rendu.
Pseudo-code de cette fonctionnalité

```
Si checkIfOut vrai alors
      Si stop < road.getSpeed // stop plus petit que la vitesse max de la voiture
            stop++
Sinon
      Si stop > 0 alors
            Stop--
```

L'attribut *stop* va ensuite servir lors de la mise à jour des positions de la voiture (dans *setPosY(int stop)* de la classe Road) et la mise à jour de la position des adversaires (dans *setY(int stop)* de la classe Opponent), Au lieu d'ajouter *speed* à l'ordonnée, on ajoute *speed* - *stop*.

6. Resultats

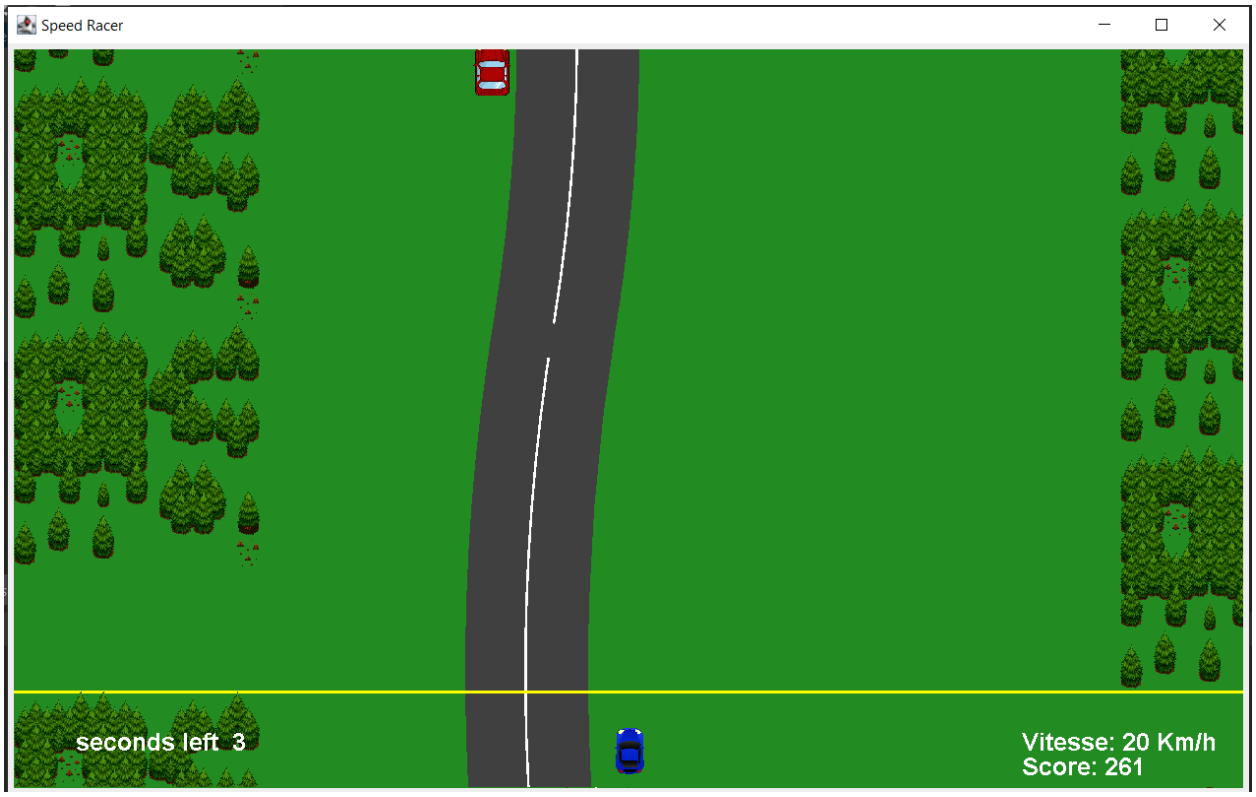


Fig. 9 : Capture d'écran de notre jeu

7. Documentation d'utilisation

- Pré-requis : Java (JDK 11 au moins) avec un IDE.
- Mode d'emploi : Importer le projet dans une IDE, sélectionner la classe possédant le Main puis "Run as Java Application".
- Une fenêtre s'ouvrira avec la voiture, la route et le décor, appuyer sur les flèches gauche et droite du clavier pour déplacer la voiture.

8. Documentation développeur

Le jeu reste encore très simple malgré les fonctionnalités déjà nombreuses que nous avons implémentées. De plus, il reste encore quelques bugs et des modifications qui n'ont pas nécessairement été faites. Ils restent encore beaucoup de fonctionnalités que nous trouvons intéressantes et que nous voulons encore implémenter, cependant à ce jour nous n'avons pas encore eu les connaissances ou le temps nécessaires pour les faire, notamment pour le passage en une vue 3D. Dans un prochain patch nous envisageons de régler les bugs déjà présents mais aussi d'ajouter certaines fonctionnalités et bien sûr les bugs qui vont avec !

9. Conclusion et perspective

A la fin de ce projet, nous avons appris à respecter le modèle MVC mais aussi à utiliser les *Thread*. Nous avons réussi à implémenter plusieurs fonctionnalités que nous estimons complexes mais nous sommes encore très motivés pour en implémenter d'autres encore plus ardues notamment le passage en 3D.