MARCÉ Taïssir & RAZAFIMBELO Fanoa

Projet PCII

Introduction

Nous allons à travers ce projet réaliser un jeu vidéo de type « course de voiture » à la première personne. Le but de ce jeu est de contrôler un véhicule aérien qui doit esquiver les obstacles et passer le plus de points de contrôle possible pendant un temps limité. Nous allons, tout d'abord, utiliser la bibliothèque Swing afin de programmer l'interface graphique et après avoir affiché la moto, implémenter la gestion des déplacements par le biais des flèches directionnelles du clavier. Par la suite, l'environnement de la moto sera implémenté en commençant par l'animation de la route. Enfin, après l'implémentation des mécaniques de jeu, le décor et l'aspect graphique de l'application seront améliorés.

Analyse globale

Les principales fonctionnalités à développer sont l'interface graphique avec la création de la moto et la gestion du clavier qui va permettre de la déplacer dans la fenêtre de jeu.

Ces fonctionnalités sont prioritaires car elles permettent de mettre en place l'environnement de jeu et sont plutôt simples à implémenter.

Dans le prolongement de ces fonctionnalités vient la création de l'environnement de jeu. Nous commençons, tout d'abord, par l'animation de la route à vitesse constante qui est une fonctionnalité prioritaire dans la mesure où toute la mécanique du jeu repose sur son animation dans la fenêtre.

Les fonctionnalités qui ne sont pas prioritaires mais que nous avons tout de même implémentées sont la stylisation de la route avec les courbes de bézier et l'affichage d'un oiseau animé à travers la fenêtre. L'animation de l'oiseau reste simple à réaliser mais pour les courbes de Bézier il faut faire attention à avoir une ligne continue entre les différentes courbes qui composent la route.

D'autres fonctionnalités prioritaires sont la création de points de contrôle et de crédit de temps. Ces fonctionnalités permettent par la suite de définir une condition de fin de partie.

Enfin, l'amélioration de l'aspect graphique n'est pas prioritaire mais permettra une meilleure expérience lors du lancement de l'application.

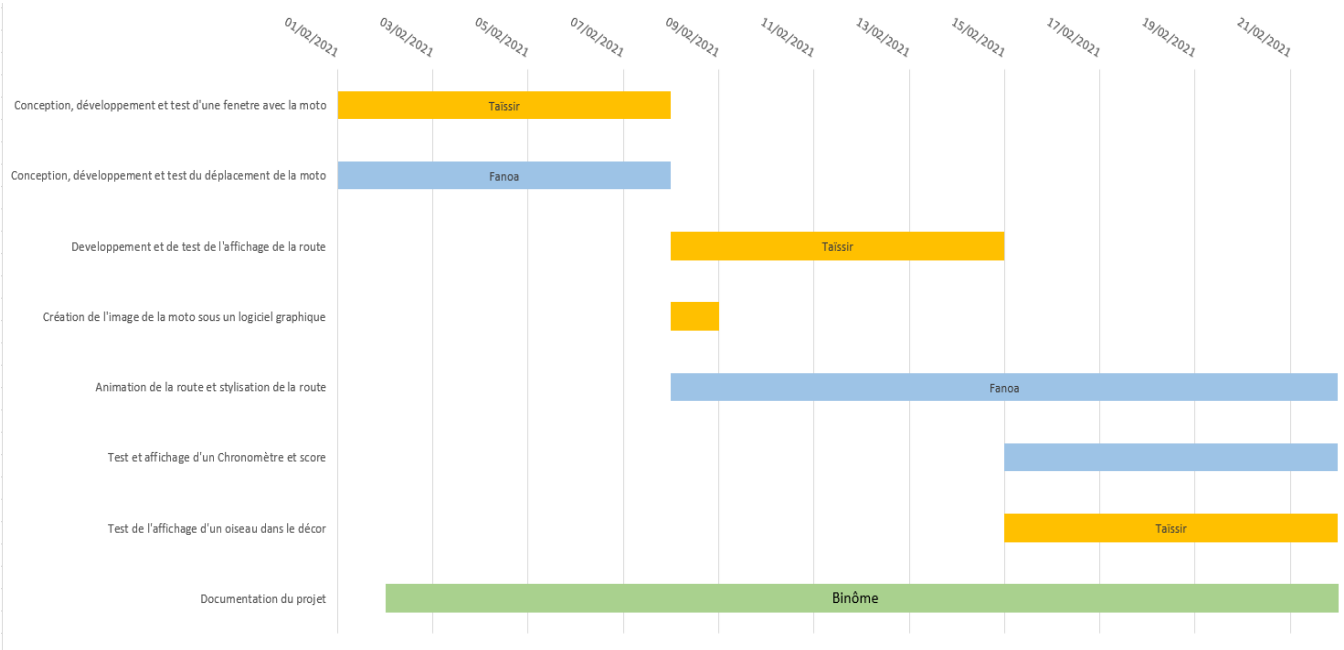
Plan de développement

Le plan de développement de l'application se divise en deux périodes : avant les vacances et après les vacances, partiels

Liste des tâches : (Avant les vacances)

- Analyse du problème
- Conception, développement et test d'une fenêtre avec le moto
- Conception, développement et test du mécanisme de déplacement de la moto
- Développement et test de l'affichage de la route
- Création de l'image de la moto sous un logiciel graphique
- Animation de la route et stylisation de la route
- Test et affichage d'un Chronomètre et score
- Test de l'affichage d'un oiseau dans le décor

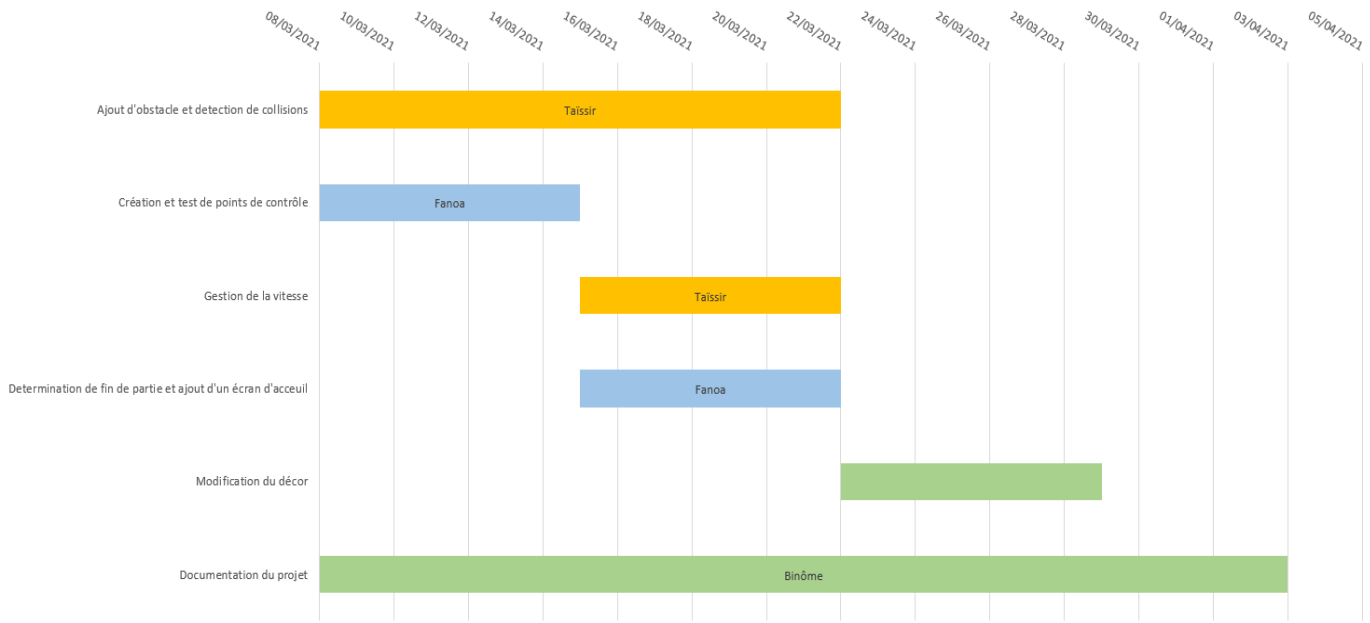
Diagramme de Gantt des différentes tâches du projet avant les vacances:



Liste des tâches : (Après les vacances)

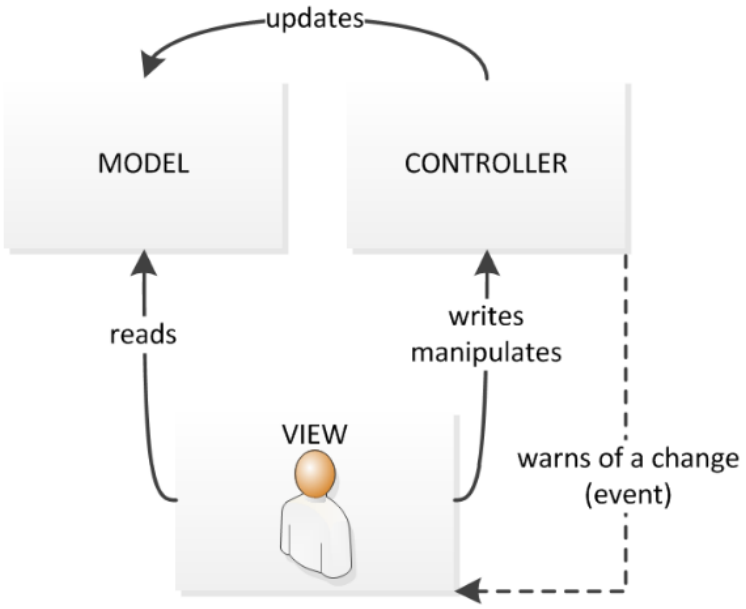
- Ajout d'obstacle et détection de collisions
- Création et test de points de contrôle
- Gestion de la vitesse
- Détermination de fin de partie et ajout d'un écran d'accueil
- Modification du décor
- Documentation du projet

Diagramme de Gantt des différentes tâches du projet après les vacances et partiels :



Conception générale

Nous avons adopté le motif MVC, Model View Controller pour le développement de notre interface graphique.



Le motif MVC (*source : Wikimedia*)

Ce motif est composé de quatre classes principales :

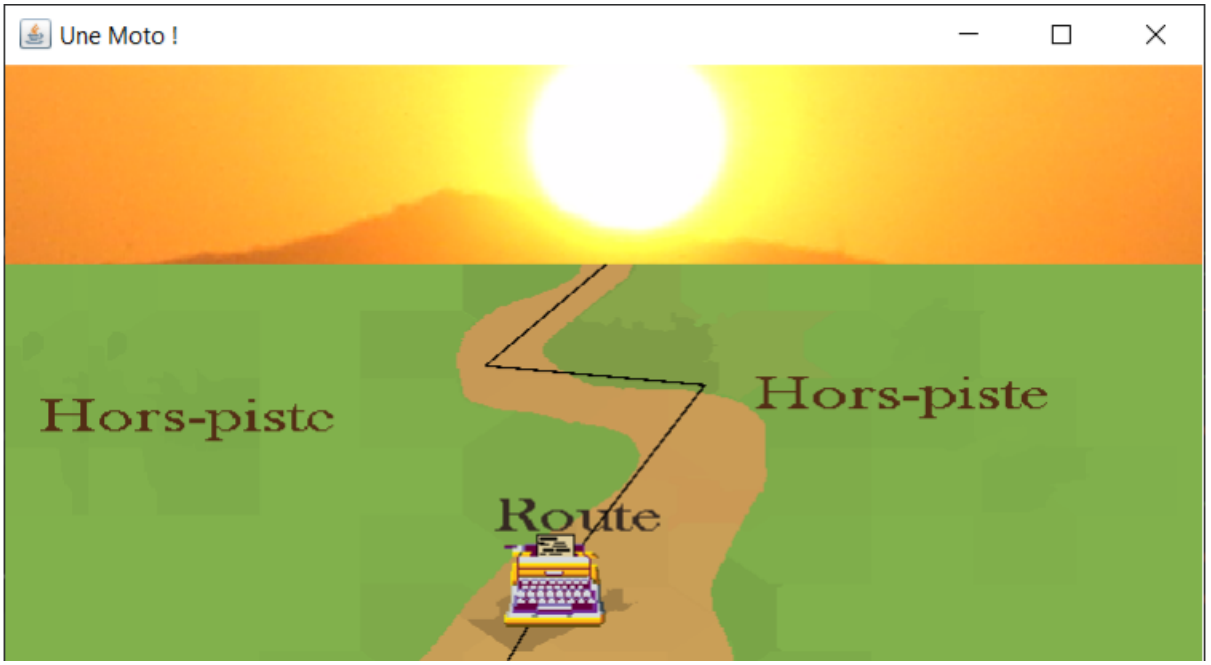
- Affichage pour la vue
- Etat pour le modèle
- Control pour le contrôleur
- Main pour lancer le programme

Conception détaillée

Tout d'abord au lancement du jeu, une fenêtre avec le menu principal s'ouvre. Cette fenêtre est créée dans la classe Main qui hérite de la classe JPanel. La fenêtre est composée d'une JFrame dans laquelle sont ajoutés un JLabel contenant l'image de fond et deux JButton. Les deux boutons sont le bouton start pour lancer le jeu et quit pour fermer l'application. Le bouton start permet au clic de fermer la fenêtre actuelle et de lancer la fenêtre de jeu en appelant la classe Game chargée de lancer le jeu.

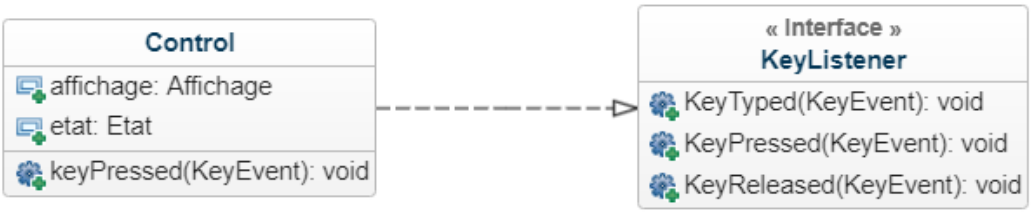
Le développement d'une fenêtre de test consiste principalement en la création et l'implémentation d'une classe Affichage. Cette classe hérite de la classe JPanel, ce qui nous permet de l'ajouter à une fenêtre (on réalise cette opération dans le main avec l'instruction "fenêtre.add(affichage)". Tout ce qui est "peint" par l'affichage y figurera donc. La classe affichage est construite autour d'une méthode principale, héritée de JPanel, la méthode paint(Graphic c). C'est au sein de cette méthode qu'on dessine sur la fenêtre les différents éléments du programme.

Dans un premier temps, nous avons utilisé des “placeholder” pour concevoir les premiers prototypes du jeu (le résultat est la capture d’écran ci-dessous). Le placeholder est un élément graphique simple (une forme géométrique, une image temporaire…) utilisé à la place d’un élément graphique plus complexe (un dessin de moto, une route générée par génération procédurale...), afin de créer et visualiser une première esquisse de l’interface en cours de développement. L’objectif était de faire le minimum pour pouvoir interagir avec le modèle et les commandes. Une fois que les fonctionnalités de base du modèle ont été implémentées, nous avons travaillé l’ambiance du jeu en remplaçant ces placeholders par des “assets” (ou ressources graphiques) créés sur un logiciel de pixel art externe.

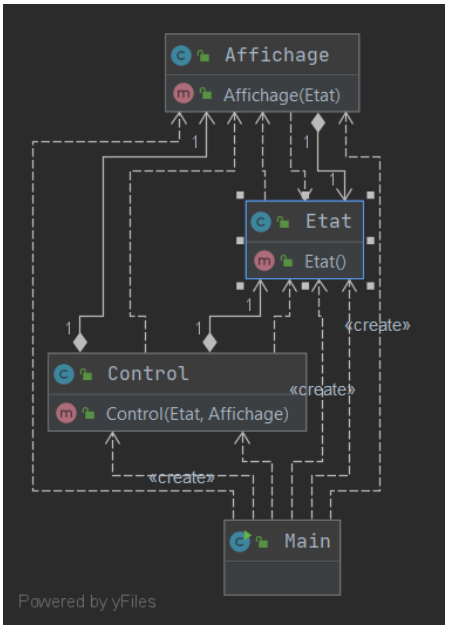


Pour le déplacement de la moto dans la fenêtre de jeu, nous utilisons la programmation événementielle avec la classe KeyListener. La position de la moto est définie dans la classe Etat avec la constante “pos”, qui va subir des modifications de coordonnées selon la touche du clavier appuyée. En effet, la classe Etat contient quatre méthodes qui vont gérer le déplacement : moveUp(), moveDown(), et moveLeft() et moveRight(). Ces méthodes vont être utilisées dans la méthode KeyPressed de la classe Control qui va appeler l’une des quatres méthodes selon la touche pressée.
La classe Control implémente l’interface KeyListener pour la gestion du clavier comme on peut le voir dans le diagramme suivant :

Diagramme de classe présentant l’implémentation de l’interface KeyListener dans la classe Control :



<u>On peut également observer l'ensemble des interractions entre les classes sur un diagramme plus général :</u>

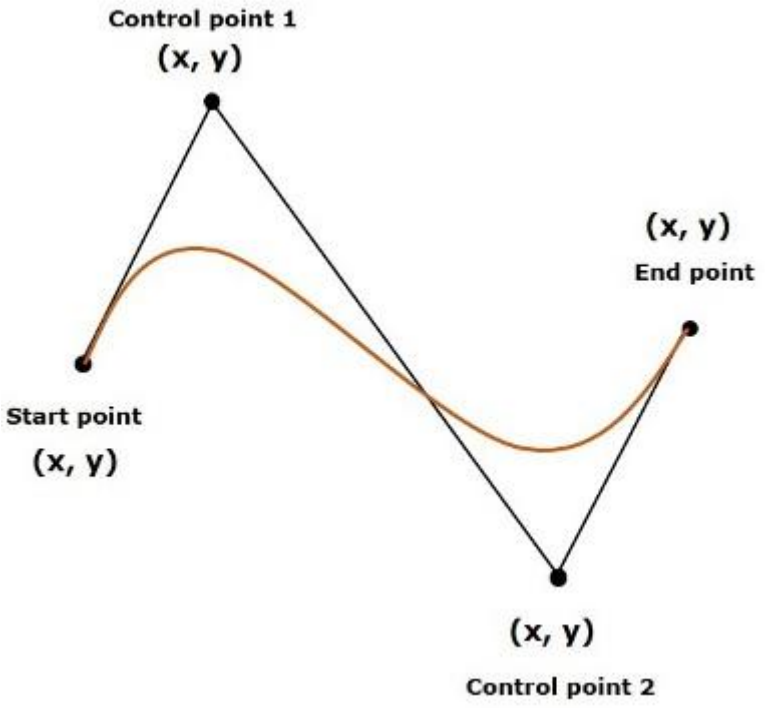


L'animation de la route infinie se fait par le biais de la classe Avancer. En effet, la classe Avancer hérite de Thread qui permet de réaliser de la programmation concurrente. Avancer hérite de la méthode run() qui va permettre d'actualiser la route en un intervalle de temps fixe. La méthode run() appelle updateRoute() dans la classe Route pour mettre à jour les points qui composent la route.
Cette méthode met à jour les coordonnées des points qui composent la route et s'assure de la création du dernier point ce qui permet de donner l'impression que la route est infinie.

Diagramme de classe présentant la classe Avancer qui hérite de Thread:

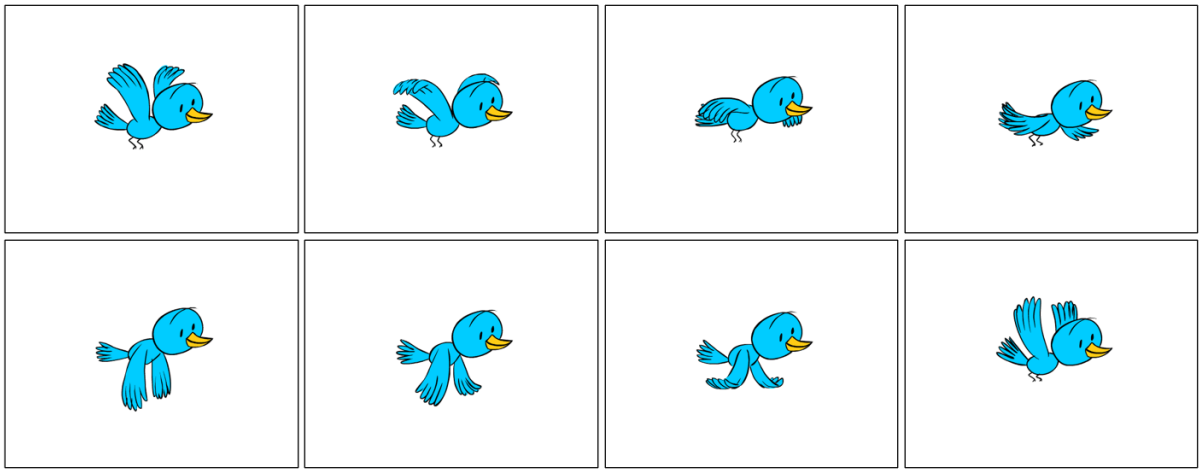
Nous avons stylisée la route à l'aide de courbes de Bézier en utilisant la bibliothèque AWT qui définit les classes `QuadCurve2D` et `CubicCurve2D`.
Un QuadCurve2D nécessite 3 points : le point de début, le point de contrôle et le point de fin; le point de contrôle permet d'ajuster la courbe entre le point de début et de fin ce qui permet de donner la forme arrondie. Ainsi, nous avons utilisé ce principe sur les points qui constituent la courbe. Pour qu'il y ait un effet de continuité entre deux courbes du parcours, on prend les 3 points successifs du parcours puis on relie le point de la première moitié à celui de la deuxième moitié avec comme point de contrôle le point qui se situe entre les deux. On complète ensuite le parcours, en traçant le segment entre le point de la dernière moitié au dernier point du parcours.

Schéma présentant le principe des courbes de Bézier :



Pour implémenter un décor dynamique, nous avons affiché un oiseau qui se déplace le long de l'écran. l'image de l'oiseau est une image gif qui a été découpée en 8 images chacune référencée par la variable "frame". Ces frames vont être regroupées dans le tableau "frames" de la classe VueOiseau chargée de l'affichage de celui-ci. Un oiseau est animé par le biais d'un Thread, la frame de départ est initialisé par la frame d'indice 0 et pendant l'exécution du thread cet indice va augmenter pour parcourir tous les indices du tableau frames de la classe VueOiseau avec à la fin un modulo 7 pour revenir sur le premier indice.

Image présentant les différentes frames de l'animation de l'oiseau :



Ce thread va permettre d'afficher successivement les images pour animer cet oiseau. La classe VueOiseau qui permet d'afficher l'oiseau est créée dans le constructeur de la classe Affichage. Affichage va appeler la méthode dessiner de VueOiseau ce qui va permettre de lancer le thread de l'oiseau.

Ce décor dynamique est accompagné de points de contrôle qui vont permettre de créditer le temps restant au véhicule. Les points de contrôle sont créés dans la classe Route par le biais de la méthode createCheckpoint(). Dans notre application, ces points de contrôles sont représentés par deux points dans une ArrayList appelées Checkpoints. Ainsi, la liste de points n'est composée que de deux points qui vont ensuite être reliés dans la classe Affichage pour former un point de contrôle représenté par une ligne horizontale sur la route.
La position du point de contrôle est mise à jour avec la méthode updateCheckpoint(), appelé dans le Thread de la classe Avancer, qui supprime les points lorsqu'elles dépassent de l'affichage.

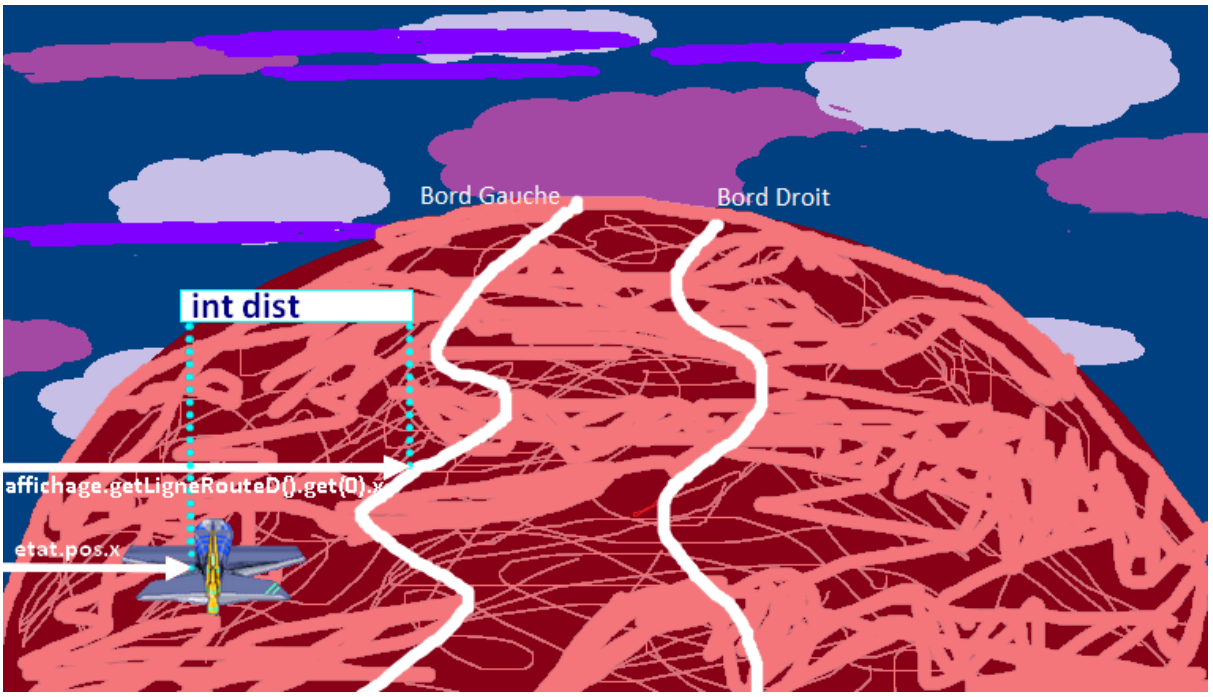
```
/** Mise a jour des points de controle*/
updateCheckpoint()
    //Verifie que la liste n'est pas vide
    if(checkpoints not empty)
        //L'ordonnée des points se déplacent en meme temps que la route
        Checkpoints[0].y += Affichage.getMove()
        Checkpoints[1].y += Affichage.getMove()
        //La liste est vidée quand les points dépassent l'horizon
        if (Checkpoints[0].y > HEIGHT())
            Checkpoints.clear();
```

Après une certaine distance parcourue, deux nouveaux points sont ajoutés à la liste pour former un nouveau point de contrôle. Cet appel à createCheckpoint() se fait dans le thread de la classe Avancer et qui selon la distance parcourue par la voiture, va appeler cette méthode.

Pour implémenter le temps restant à la voiture pour atteindre le prochain point de contrôle, nous avons créé la classe Chrono qui implémente un Thread. Ce thread va permettre d'initialiser un temps dans la variable chrono et de le diminuer chaque seconde en définissant le délai du thread à 1000 ms, soit une seconde. Une fois que la voiture passe un point de contrôle, le temps est crédité. Cela est vérifié dans la classe Avancer qui compare l'ordonnée de la voiture à celle du point de contrôle et crédite le temps si l'ordonnée de la voiture est inférieure. Cependant plus la voiture passe de points de contrôle, plus le temps crédité diminue.

La vitesse de la voiture est calculée en fonction de sa proximité à la route. En effet, ce calcul fait en sorte que la voiture soit ralentie progressivement si elle s'éloigne de la route jusqu'à descendre à zéro indiquant la fin de partie. Pour faire cela, on calcule dans la variable dist la distance entre la moto et l'extrémité de la piste la plus proche(voir schéma ci-dessous).

Schéma présentant l'implémentation de la vitesse de la moto en fonction de sa proximité à la route



La classe Obstacle permet la création des obstacles de l'environnement. Un obstacle possède comme attribut : une position représentée par un point, une image avec sa largeur et sa hauteur, et un booléen visible qui indique si l'obstacle est affiché à l'écran. Les obstacles sont réunis dans la classe Route dans une ArrayList. Ils sont créés par le biais de la méthode createObstacle() qui crée une instance d'Obstacle et l'ajoute à l'arraylist.

```
/** Création d'un obstacle*/
createObstacles() :
    //on choisit un nombre aléatoire entre 0 et 100
    entier rand = Random(100)
    Si rand < 20 :  //la probabilité de créer des obstacles
        entier X = Random(LARGEUR Fenêtre)
        entier Y = Random(HAUTEUR Horizon)
        //Création de l'obstacle
        Obstacle o = Nouvel Obstacle(X, Y)
        Ajouter o à obstacles
```

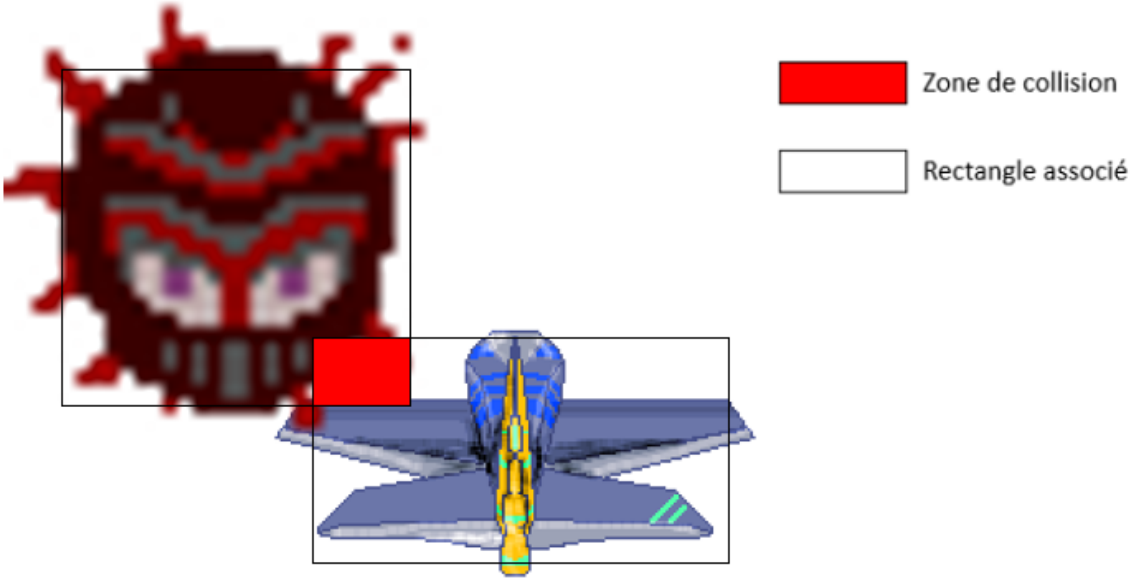
La méthode updateObstacle() permet ensuite d'actualiser leur position pendant l'avancée de la moto en incrémentant leur ordonnée de la variable move de la classe Affichage représentant de combien avance la moto. Cette méthode est appelée dans la méthode du thread Avancer et modifie les coordonnées des obstacles.

La collision entre un obstacle et la moto est gérée dans la classe Etat par la méthode CheckCollision(). Une image du jeu est définie par un rectangle et en récupérant pour chaque image le rectangle associé, on peut vérifier la collision de

deux rectangles par le biais de la méthode intersect(). Ici, et pour respecter les normes de conception de ce type de jeu, on associe à chaque objet un rectangle plus petit que l'image qui le représente.

Le rectangle associé à la moto est récupéré puis on teste si elle entre en collision avec l'un des rectangles associés aux obstacles. S'ils entrent en collision, la variable booléenne visible de l'obstacle devient false et n'est plus affichée à l'écran.

Schéma présentant le principe de collision en utilisant les rectangles associés aux images :



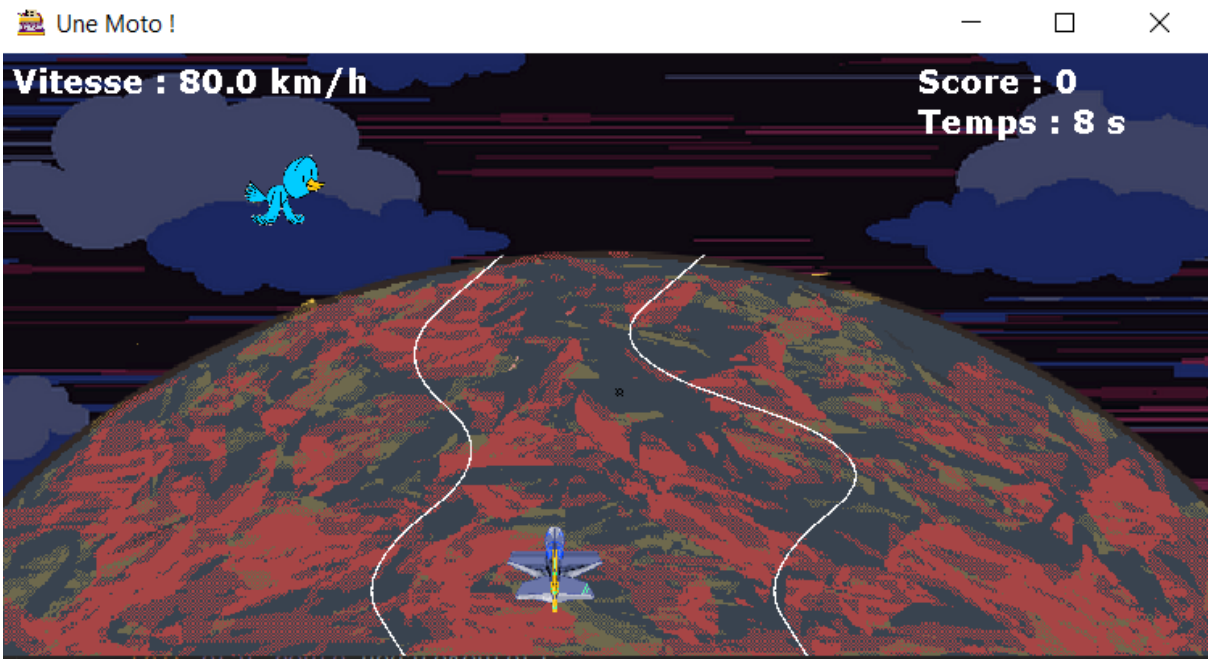
Entrer en collision avec un obstacle ajoute une pénalité à la vitesse de la voiture, ce qui la ralentit. Cela s'opère en ajoutant la valeur de la pénalité au délai de la classe Avancer pour créer un effet de lenteur.

Enfin la fin du jeu est vérifiée par la méthode testPerdu() de la classe Etat renvoyant un booléen qui vérifie si le temps restant ou la vitesse de la voiture sont tombés à zéro. Ce booléen est la condition d'arrêt du thread de la classe Avancer et est évalué dans la méthode paint d'Affichage avant d'afficher l'écran de fin

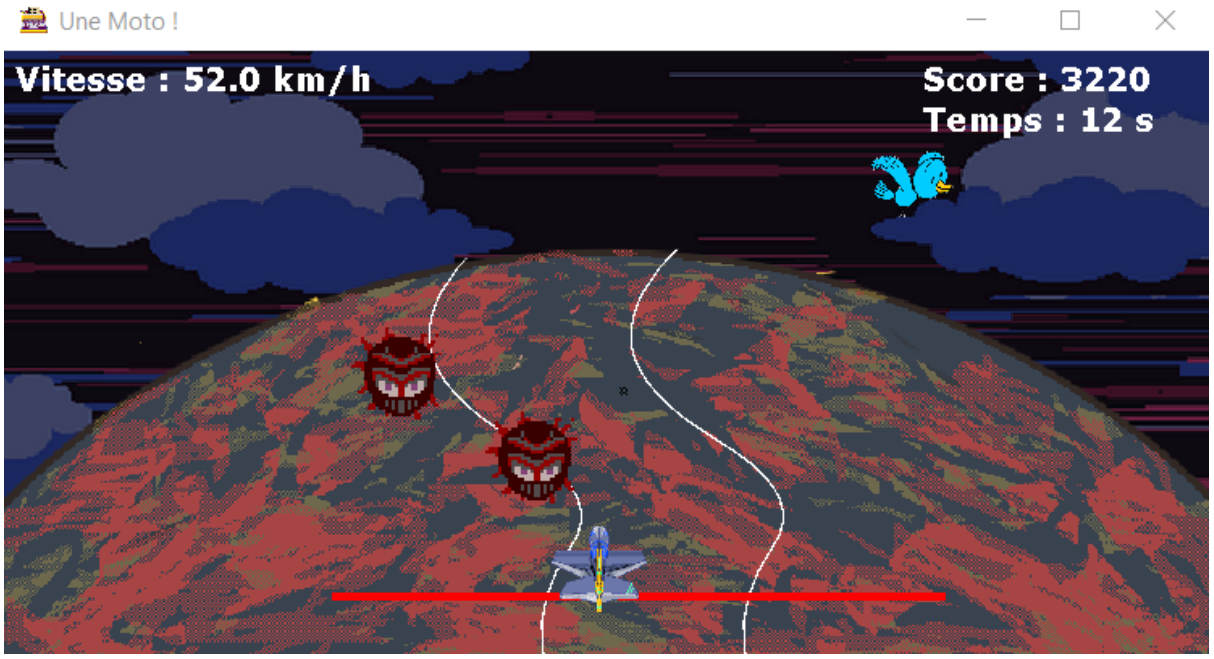
Résultat

Au lancement, le menu principal du jeu apparaît avec les deux boutons : start pour lancer le jeu et quit pour fermer la fenêtre. La capture suivante montre l'affichage du menu à l'écran

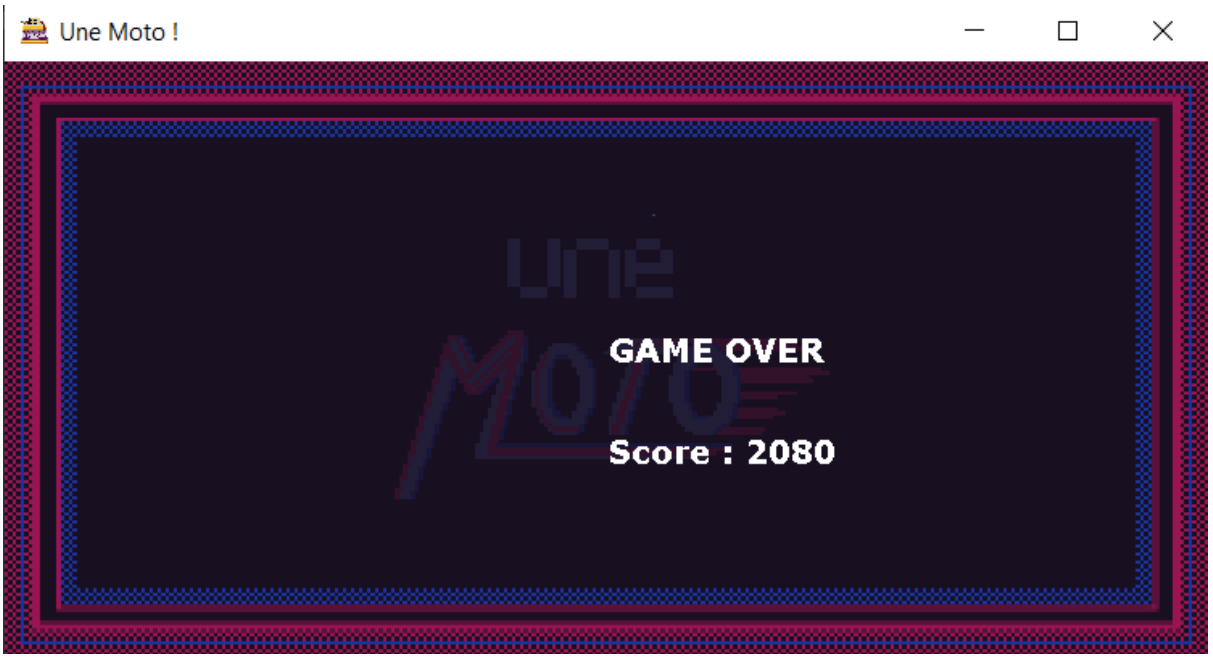
Une fois le bouton start appuyé, une nouvelle fenêtre avec le jeu s'ouvre. On peut voir sur la capture d'écran suivante l'image que nous avons créé pour la moto avec son environnement. On peut y voir également l'oiseau dynamique et la stylisation de la route par le biais de courbe de Bézier.



La capture d'écran suivante montre l'affichage du temps restant à la voiture pour atteindre le prochain point de contrôle et l'affichage d'un point de contrôle matérialisé par une ligne horizontale sur la route.



On peut voir aussi l'image créée pour représenter les obstacles que la voiture doit éviter



Lorsque la méthode testPerdu() est vérifiée dans la classe Affichage (soit le temps est écoulé, soit le véhicule a perdu trop de vitesse), un écran signale la fin du jeu et affiche le score obtenu.

Documentation utilisateur

- Prérequis : Java version 1.8 ou plus, avec un IDE
- Mode d'emploi : importer le projet dans l'IDE, sélectionner la classe Main puis "Run as Java Application"

  Une fenêtre s'ouvre avec l'écran titre, appuyez sur le bouton start pour lancer le jeu. Une autre fenêtre s'ouvre avec la moto au centre, utiliser la flèche du haut pour lancer le jeu et les flèches directionnelles pour déplacer la moto dans son environnement.

Documentation développeur

Les classes principales à regarder sont celles qui implémentent le schéma MVC : Affichage, Etat, Control. La classe Main permet de lancer le programme.

Les principales constantes que l'on peut modifier pour changer le fonctionnement du code sont, tout d'abord dans la classe Affichage, la taille de la fenêtre de jeu défini par les constantes "WIDTH" et "HEIGHT". Les constantes liées à la moto peuvent aussi être modifiées, notamment sa largeur et hauteur définis respectivement par les constantes "largeurMoto" et "hauteurMoto" et de combien de pixels elle peut se déplacer définie par la constante "move".
Dans la classe Etat, la position initiale de la moto peut être modifiée en changeant les coordonnées de la constante "pos".
Dans la classe Oiseau, le déplacement de l'oiseau peut être changé en modifiant la variable délai: plus le délai est faible plus l'oiseau se déplace rapidement dans la fenêtre. Sa hauteur et sa position peuvent aussi être modifiées avec respectivement les variables hauteur et pos.
Dans la classe Chrono, le temps initial donné à la voiture pour atteindre le prochain point de contrôle est défini par la variable start qui peut être augmentée ou diminuée pour ajuster la difficultée du jeu.
L'image d'un obstacle peut être modifiée dans la classe Obstacle en changeant le fichier stocké par la variable image.

Les fonctionnalités que nous n'avons pas encore pu implémenter mais qui mériteraient d'être développées en premier sont la présence d'adversaires et la création d'une sensation de profondeur.
La création d'adversaire devrait être possible à implémenter en définissant une classe Adversaire qui hérite de Thread, ainsi chaque concurrent serait animé par un

thread. L'affichage serait géré par une classe VueAdversaire qui disposerait d'un tableau d'adversaire à afficher. Une méthode addAdversaire() se chargera d'ajouter un adversaire à l'écran selon une probabilité fixée et une méthode updateAdversaire() aura pour rôle d'enlever du tableau les adversaires qui sortent de la fenêtre d'affichage.
Pour l'effet de profondeur, il faudrait définir un point de fuite vers lequel les points de la route convergeront.

Conclusion et perspectives

Nous avons donc réalisé un jeu vidéo des années 80 de type "course de voiture" en vue à la première personne.
Les principales difficultés étaient de créer une route en forme de courbe car il fallait calculer chaque point de contrôle de la courbe de Bézier entre les coordonnées des points composants la route. Une autre difficulté est la création d'une sensation de profondeur.

Le projet nous a permis, d'une part, de se concentrer sur la planification et l'organisation du travail en équipe (notamment en étant pour nous l'occasion de nous familiariser avec les outils de gestions de version, tel que ceux que propose GitHub, ou plus simplement s'appuyant sur une liste de jalons tout le long du développement). D'autre part, nous avons été amenés à nous initier à la programmation de plusieurs Threads au sein d'un même programme. Nous avons également pu revoir le motif Modèle-Vue-Contrôle qui nous a été enseigné précédemment.
Ce jeu pourra dans le futur évoluer en améliorant son aspect graphique notamment dans le décor et l'ajout d'une sensation de profondeur. On peut également réfléchir à des améliorations du gameplay en expérimentant différentes valeurs pour les variables du jeu (les variables de vitesse ou de pénalité par exemple), afin de jouer sur la sensation de jeu qui est souvent au centre de l'expérience vidéoludique des jeux de course. Dans le même sens, il est souvent plus intéressant pour le joueur d'implémenter des motifs dans l'apparition des obstacles, plutôt que de le générer aléatoirement. Il pourrait également être intéressant d'améliorer l'interaction déclenchée par la collision entre la moto et un obstacle, en animant par exemple la moto ou l'obstacle (avec un clignotement par exemple) afin de signifier au joueur plus explicitement que l'événement ne lui est pas bénéfique.