université
PARIS-SACLAY

FACULTÉ
DES SCIENCES
D'ORSAY

RAPPORT DE PROJET

PROGRAMMATION CONCURRENTE ET INTERFACE INTERACTIVES

BERTHIER Martin
&
BERTRAND Quentin

Janvier 2021 à Avril 2021

Référent de matière : Monsieur SABOURET <nicolas.sabouret@ université-paris-saclay.fr>
Chargée de TD : Madame NGUYEN <thi-thuong-huyen.nguyen@université-paris-saclay.fr>

Table des matières

Table des figures

Introduction

Le but de ce projet est de réaliser un jeu de course de moto en pseudo 3D. Le but du jeu étant d'incarner une moto et de se mouvoir dans un environnement grâce aux touches directionnelles. On devra passer des portes avant un certain lapse de temps si on ne veut pas perdre. Il y aura aussi d'autres motos dans cette course qui seront vos adversaires et que vous devrez dépasser. La vitesse sera un facteur à prendre en compte, en effet plus vous roulez plus vous gagnerez en vitesse mais *a contrario* si vous roulez autre part que sur la route ou si vous heurtez un ennemi et/ou obstacle vous perdrez de la vitesse. La difficulté de ce jeu étant de réussir à appréhender la vitesse tout en esquivant les autres joueurs et les différents obstacles tout en passant les portes dans le temps imparti. Le score du joueur se base sur la distance parcourue.

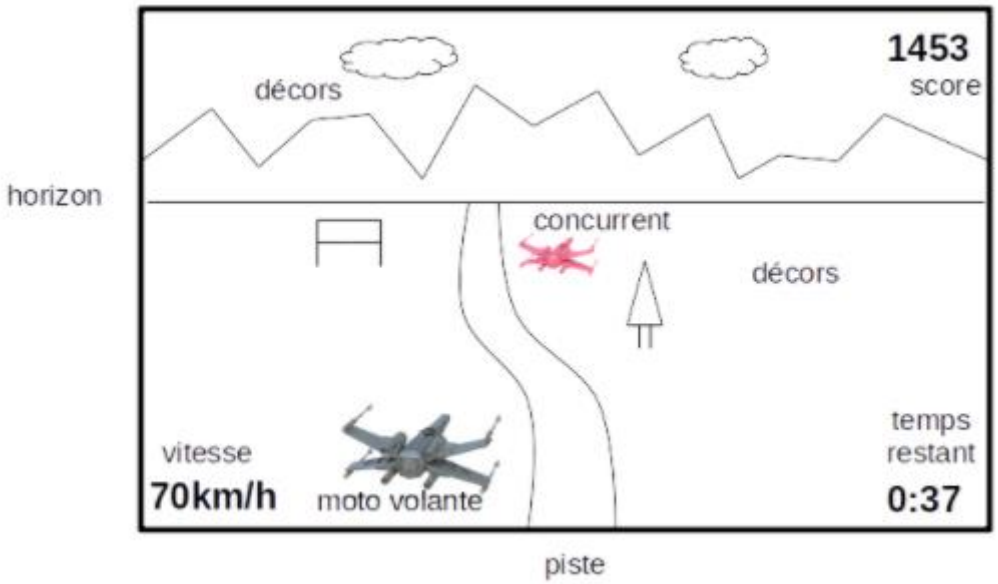

Figure 1 : Aperçu de base que le jeu devrait avoir

Analyse Globale

Nous allons réaliser un projet en *Java*, ce projet devra respecter le *model MVC*, nous allons donc répartir nos classes de façon à ce que chacune d'entre elles se situe soit dans le contrôleur, la vue ou le modèle. Ce projet devra être réalisé sur une durée de sept semaines. Pour implémenter notre jeu, nous allons devoir implémenter de nombreuses fonctionnalités. Une interface graphique afin d'afficher notre jeu (route, moto, obstacles, etc.) qui nous servira aussi d'interface avec l'utilisateur. Il faudra également mettre en place une interface faisant le lien entre les touches pressées par l'utilisateur et notre programme, à l'aide d'un *KeyListener* ainsi qu'une multitude de fonctionnalités plus ou moins nécessaires que nous allons énoncer dans la partie Plan de développement.

Plan de Développement

Nous allons énoncer toutes les fonctionnalités qui vont être mises en place dans ce projet, la répartition de ces fonctionnalités se fait en deux catégories distinctes, d'un coté les fonctionnalités dites « nécessaires » au bon fonctionnement du projet et de l'autre coté les fonctionnalités dites « complémentaires » ou « Optionnelles » qui peuvent être vues comme des *features* améliorant le projet. Il sera également mentionné la difficulté générale liée à l'analyse, la conception, le développement et les tests de chacune de ces fonctionnalités.

Fonctions nécessaires :

- Création d'une interface graphique, facile
- Affichage de la route avec une notion de profondeur, moyenne
- Affichage de la moto, facile
- Affichage d'un environnement autour de la route, moyenne
- Implémentation du mouvement latéral de la moto, facile
- Implémentation de *KeyListener*/*MouseListener*/*MouseMotionListener* pour les interactions jeu / utilisateur
- Mouvement de la route et route infinie, moyenne
- Utilisation de *sprites* pour la moto et le *background*, difficile
- Implémentation de courbes dans la route, difficile
- Mise à jour de l'affichage, moyenne
- Mise en place d'un *timer* et du score, moyenne
- Implémentation des *gates* servant de *checkpoints*, moyenne
- Implémentation de la vitesse (moto + route), moyenne
- Mise en place d'obstacles, difficile
- Création d'une partie, moyenne
- Implémentation d'une fin de partie, moyenne

Fonctions complémentaires :

- Implémenter une notion de vitesse contrôlable avec les touches directionnelles, moyenne
- Aucun déplacement possible quand la vitesse est nulle, facile
- Implémentation d'animation pour la moto lors de l'accélération et du freinage, facile
- Implémentation d'un *sprite* de compteur de vitesse évoluant en fonction de la vitesse, moyenne
- Mise en place d'un écran d'accueil, difficile
- Mise en place d'un *highscore*, moyenne
- Animation de la moto quand elle tourne, difficile
- Mise en place des collisions avec les obstacles, difficile
- Mise en place d'un système de vie, moyenne
- Affichage d'éléments de décor en dehors de la route, moyenne

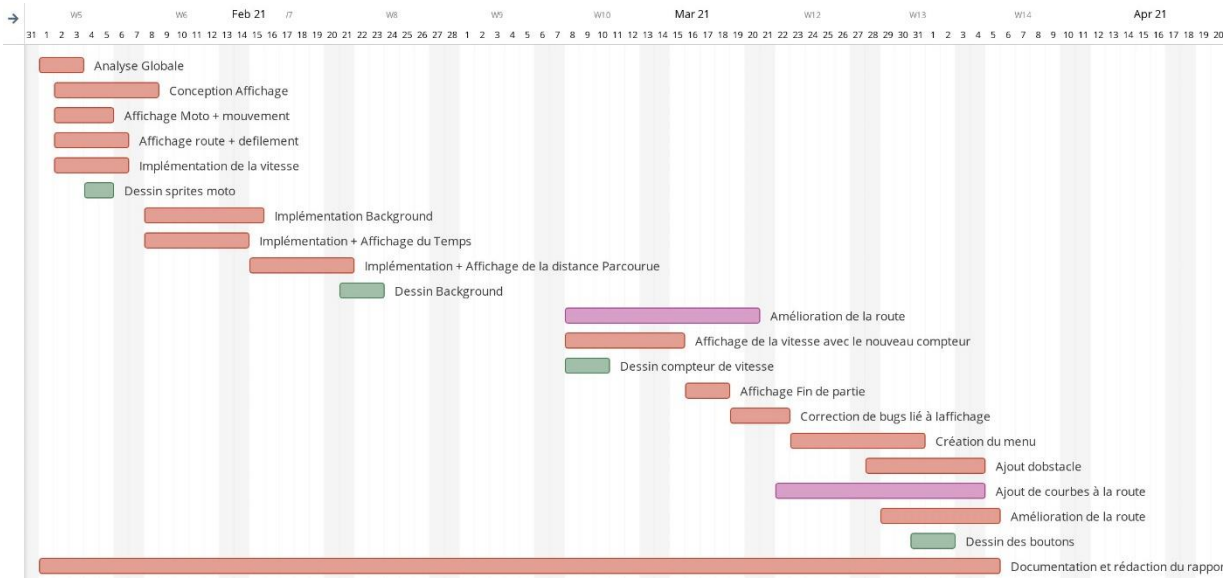

Figure 2 : Diagramme de Gantt du projet

Ci-dessus le diagramme de Gantt de notre projet, en rouge ce que nous avons réalisé ensemble, en mauve ce que Martin BERTHIER a réalisé et en vert ce que Quentin BERTRAND a réalisé.

Conception Générale

Ce projet a été réalisé en utilisant le modèle *MVC*. Ce modèle consiste a séparer dans différents packages les classes gérants respectivement la *View* (donc la partie affichage), le *model* (toutes les mécaniques de jeu) et enfin le *Controller* (tous les inputs du joueurs). Ce modèle nous permet d'avoir une meilleure lisibilité du projet et de mieux comprendre l'impact de chaque classe sur le projet.

Le package *Model* contient toutes les classes gérants la partie de contrôle du projet, composé des packages suivants :

- *Road* contient les classes : *Curbs, Obstacles*, *Road* et *Segment* qui gèrent tout ce qui concerne la route, composée de *curbs* et de segments et les obstacles qui se trouvent dessus.
- *Threads* contenant les classes suivantes : *TH_Game, TH_Handler, TH_KeyManager* et *TH_Scrolling* qui contiennent toutes des *threads* qui gèrent respectivement le déroulement de la partie, l'*update* du *Handler* qui permet d'accéder aux divers éléments de façon plus simple, *update* le *keyManager* et gère la vitesse de déroulement de la route.
- La classe *Player* qui gère tout ce qui concerne le joueur (accélération, freinage, mouvements latéraux, etc.)

Le package *View* contient toutes le classes gérants la partie affichage du projet, composé des packages suivants :

- *Gfx, contenant les* classes *: Assets, Display* et *SpriteSheet* qui gèrent respectivement les *assets* (*load* les *sprites* et les polices), l'affichage de la fenêtre et enfin la gestion de tous les *sprites.*
- *Scenes*, contenant les classes : *GameScene*, *CreditScene, HighscoreScene, MenuScene*, *Scene* et *SceneManager*, toutes ces classes gèrent ensemble le système de *scenes* nous permettant de changer de « menu », nous offrant la possibilité de faire plusieurs choses sur des menus différents
- *UiObjects*, contenant les classes : *Button*, *ClickListener*, *UiObjects* et *UiObjectsManager* gérant toute la partie des *UiObjects*, c'est-à-dire tout ce qui concerne la souris et l'action de la souris avec le système de bouton et le système d'action de souris (*onClick(), onMouseMove(), onMouseRelease(),* etc.)
- *Utils contenant les classes : FontLoader, ImageLoader, Utils* et *Text,* gérant respectivement les polices d'écriture du projet, l'importation d'image dans le projet, la manipulation de fichier et toutes les insertions de textes.

Le package *Controller* contient toutes les classes gérant la partie de traitement des *inputs* du joueur, composé des classes suivantes :

- *KeyManager* gère tous les *inputs* du joueur sur le clavier
- *MouseManager* gère tous les *inputs* du joueur avec la souris

La classe *Main,* elle, se trouve dans le dossier *src* et sert à elle seule à lancer tout le projet. Il existe aussi un dossier *ressources* contenant à son tour trois sous-dossiers : *Buttons, Others* et *Textures* contenant les *sprites* des boutons, les polices utilisées dans le projet et enfin tous les *sprites* utilisés pour faire les textures du jeu (personnage, *background,* obstacles, etc.)

Conception Détaillée

Pour l'affichage de la fenêtre de notre projet, nous avons utilisé *JFrame* qui appartient à *Java Swing* et pour l'affichage du contenu de la fenêtre nous utilisons *JPanel*, sous la forme d'un *SceneManager*. La classe *Display* contient donc l'affichage de la fenêtre du jeu et la classe *SceneManager, extends JPanel* affiche le contenu de cette fenêtre comme dis précédemment. Le *SceneManager* permet de changer de scene très aisément, nous nous en servons afin d'afficher ce que nous avons besoins au moment voulu, comme par exemple l'affichage du menu ou bien du jeu. Chaque *scene* découle de la classe abstraite *Scene*, ce qui permet de facilement créer des *scenes* et ajouter de nouvelles *scenes* différentes rapidement. Chaque *scene* contient son propre affichage, c'est-à-dire que tout ce qu'une *scene* doit afficher est contenu dans cette même dite *scene.* Nous avons donc les classes *GameScene*, *CreditScene*, *HighscoreScene* et *MenuScene* qui s'occupent respectivement de l'affichage du jeu, des crédits, des *highscore* et du menu. Pour mieux comprendre, le diagramme de classe ci-dessous représente tout le système de *scene* que nous avons mis en place.

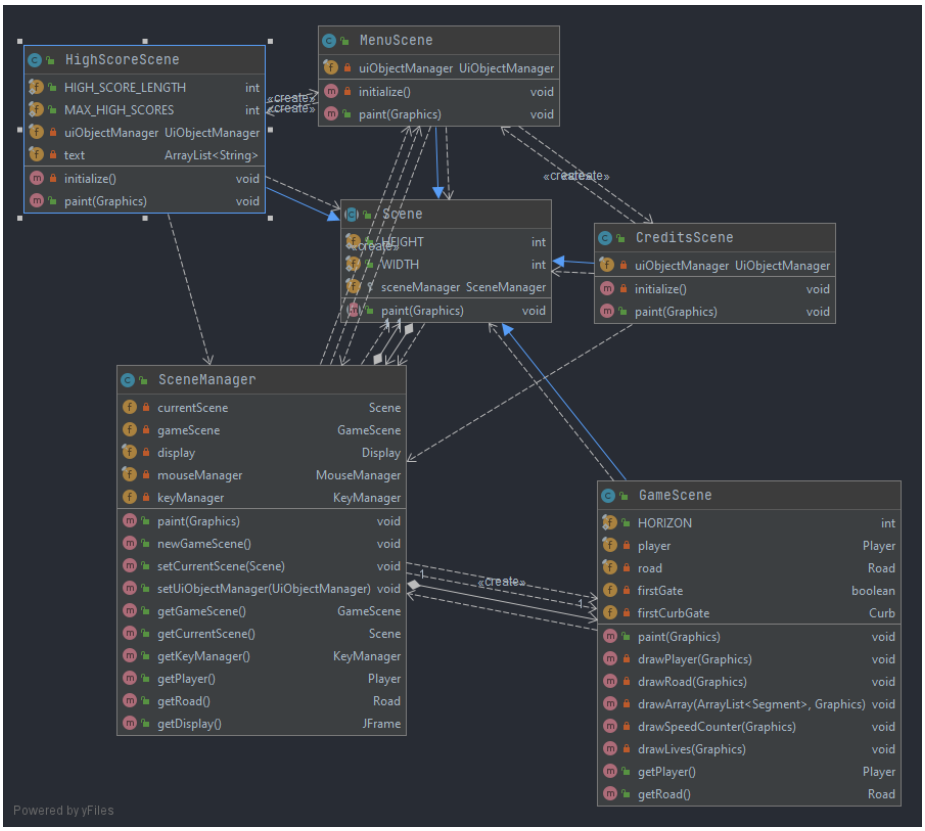

Figure 3 : Diagramme de classe du package Scenes

L'interface du menu de notre jeu est une interface complètement cliquable. Un système de bouton a été mis en place géré par le *package UiObject*. Chacun de ces boutons est composé de trois *sprites*, chacun de ces *sprites* correspond à un état du bouton : le bouton normal, sans aucune action de l'utilisateur sur ce dernier, le bouton lorsque la souris se situe sur le bouton sans pour autant cliquer dessus et enfin lorsque l'utilisateur clique sur le bouton. Chacun des boutons est mis à jour grâce à la classe *UiObjectManager*, elle met à jour l'ensemble des *UiObjects*, ce system rend le code très permissif quant à l'ajout de nouveaux types d'*UiObjects*. Tout comme le *package Scenes*, nous vous donnons ici aussi un diagramme de classe afin de mieux comprendre les relations entre toutes les classes *UiObject*, le diagramme correspondant se situe ci-dessous.

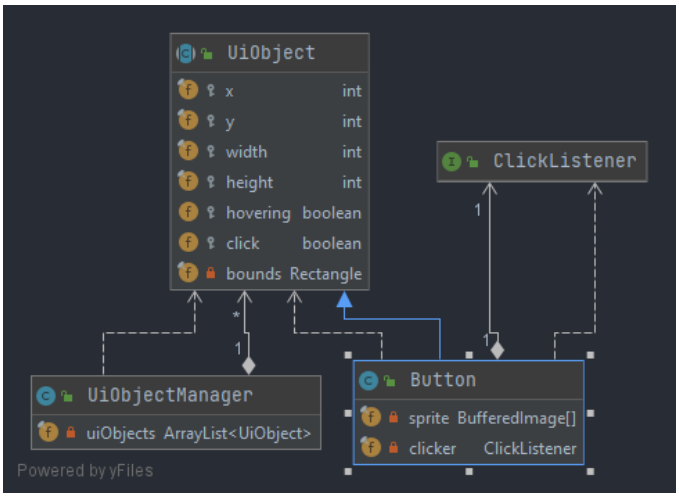

Figure 4: Diagramme de classe du package UiObjects

Nous allons maintenant nous intéresser brièvement aux classes *Assets*, *SpriteSheet* et au *package Utils*. Toutes ces classes ont un point commun, ils ne servent pas à proprement parler au fonctionnement du jeu, mais sans eux le jeu ne serait pas ce qu'il est. En effet, toutes ces classes permettent d'importer dans le projet de nombreuses choses comme par exemple les polices d'écriture, les images, l'écriture de texte et l'importation de *sprites*. *Assets* et *SpriteSheet*, comme leurs noms indiquent, gèrent tout ce qui concerne l'importation de *sprites* dans le projet. Dans le *package Utils*, nous avons les classes *FontLoader, ImageLoader*, *Utils* et *Text* qui comme dit plus haut gèrent l'importation de police, d'images, la modification du fichier *highscore* et de texte.

Nous allons maintenant nous intéresser à la partie qui gère les *inputs* de l'utilisateur. Nous avons utilisé un *KeyListener*, un *MouseListener* et un *MouseMotionListener* afin de servir d'interface entre le joueur et le jeu.

Le *MouseManager* appelle l'*UiObjectListener* afin d'appeler l'ensemble des méthodes correspondantes à celle du *MouseListener* et du *MouseMotionManager*. Cela a pour but de mettre à jour les *UiObjects* susceptibles d'être concernés par les actions de ces derniers. Le *MouseManager* est l'interface entre l'utilisateur et le jeu via la souris.

Le *KeyManager* sert d'interface entre l'utilisateur et le jeu via le clavier. Il est composé d'un *array* de booléen, qui sont tous initialisés à *FALSE*, dont la taille est 256, correspondant à la table ASCII étendue (8 bits). A chaque fois qu'une touche est pressée, le booléen correspondant à l'index du *keyCode* de la touche pressée est passé à *TRUE* puis est remise à *FALSE* lorsque la touche est relâchée. Grâce à ce système, on peut faire appel à une méthode de mise à jour sur le *KeyManager*, qui va permettre de faire des actions en fonction de la valeur du booléen correspondant à la touche voulue.

Par exemple, pour augmenter la vitesse de notre joueur nous devons appuyer sur la touche *up* des flèches directionnelles, également appelée *VK_UP*, ci-dessous le pseudo code de la fonction d'accélération en fonction de la dite touche d'accélération :

```
Début de la méthode accélération :
        Si Array[KeyEvent.VK_UP] == VRAI
        Alors accélérer()
        Fin Si
Fin de la méthode accélération
```

Nous allons maintenant regarder plus en détails le fonctionnement de notre jeu. Notre projet est basé sur un système de *threads*. Lorsque la fenêtre est créée, le *thread TH_Handler* est créé, ce *thread* est celui qui gère l'affichage de la scène en appelant *sceneManager.repaint()*. Il permet aussi de lancer les autres *threads* lorsque la *GameScene* est la *scene* actuelle.

TH_KeyManager est, comme son nom l'indique, un *thread* qui s'occupe d'appeler la méthode *update* du *KeyManager* qui permet d'effectuer les actions adéquates au moment où la touche est pressée.

TH_Scrolling est un *thread* très important, en effet il gère tout ce qui concerne le défilement de la route. Il met à jour la position de la route en fonction de la vitesse du joueur, cette dernière est calculée avec la fonction suivante *calculateSleep()*, qui est une fonction linéaire qui varie en fonction de la vitesse du joueur et renvoie la valeur de l'attente du *thread*, ce qui permet de réguler la vitesse de mise à jour de la route.

TH_Game est le dernier *thread* de notre projet mais probablement le plus important. Quand la *GameScene* est la *scene* active, *TH_Handler* s'arrête et lance *TH_Game*, ce *thread* met à jour l'affichage du jeu ainsi que la distance parcourue par le joueur. Son temps d'attente est de 16 millisecondes, ce qui nous permet d'avoir un affichage de 60 FPS (*frames per second*), ce temps d'attente est réglable grâce à la constante *GAME_SPEED*. C'est aussi ce *thread* qui lance les *threads TH_KeyManager* et *TH_Scrolling*. Il s'occupe également d'afficher la fenêtre de fin de partie.

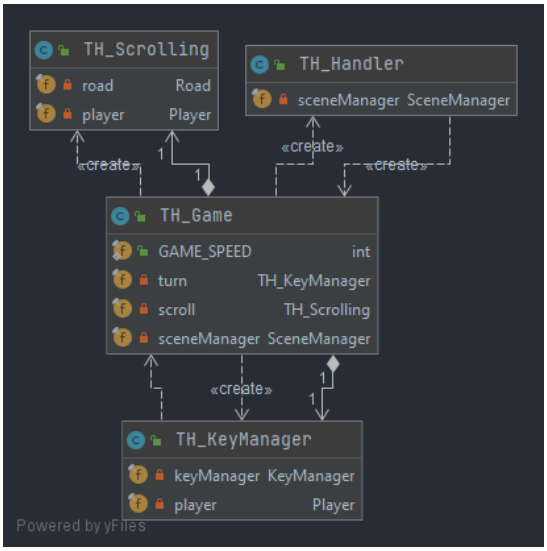

Figure 5: Diagramme de classe du package Threads

Nous allons maintenant nous intéresser au joueur. Le joueur ne bouge jamais, c'est l'environnement qui évolue de façon à donner l'illusion que le joueur bouge. Le joueur est défini dans la classe *Player*, dans cette classe nous gérons tout ce qui concerne notre joueur, comme par exemple sa vitesse, sa position, sa distance parcourue ou encore plusieurs constantes comme MAX_SPEED qui donne la valeur de la vitesse maximale que le joueur peut atteindre. L'incrémentation de la vitesse du joueur varie en fonction de la vitesse du joueur, pour donner cet effet, plus le joueur va vite plus la prise de vitesse est lente, comme avec les vrais véhicules. On réalise cette accélération de notre joueur avec la méthode *accelerate()* :

Début de la méthode *accelerate* :
si player_speed < MAX_SPEED
alors si player_speed < 75 alors player_speed <- player_speed + 0.7
alors si player_speed < 200 alors player_speed <- player_speed + 0.5
alors si player_speed < 275 alors player_speed <- player_speed + 0.3
sinon player_speed <- player_speed + 0.1
Fin Si
Fin Si
Fin de la méthode *accelerate*

L'importance de la constante MAX_SPEED est d'autant plus grande car la vitesse de défilement de la route étant en fonction de la vitesse du joueur, changer cette dernière revient à changer la vitesse maximale du jeu en lui-même. En ce qui concerne la perte de vitesse du joueur, il existe de méthodes une dite naturelle avec la méthode *decelerate()* et une dite manuelle avec la méthode *brake()*. La vitesse de freinage est constante peu importe la vitesse grâce à la constante BREAKING_SPEED, changer sa valeur revient à modifier la vitesse de freinage du joueur. La vitesse de décélération est aussi constante et correspond à 1/3 de la valeur de BREAKING_SPEED.

En ce qui concerne l'animation du joueur, tout dépend de son état, il existe plusieurs états pour le joueur : freinage, décélération et accélération. Chaque animation est composée de plusieurs frames qui changent en fonction de la vitesse du joueur. Par exemple, plus le joueur va vite plus la roue tourne vite etc. La vitesse de défilement des *sprites* varie également en fonction de la vitesse du joueur et donc de la route afin de créer une certaine cohérence dans le jeu. Pour l'animation on utilise une fonction *anim()* qui utilise un système de *clock* assez basique.

```
Début de la méthode anim :
    setup <- calculateSleep() / 1000000
    clock <- clock + 1
    si clock > setup
    alors animation <- animation + 1
        animation <- animation % Assets.player[0].length
        clock <- 0
    Fin Si
Fin de la méthode anim
```

Animation est une variable de la classe *player* qui correspond à la phase de l'animation afin de savoir le *sprite* à afficher.

La classe *Player* définie aussi la distance parcourue par le joueur, qui est mise à jour à chaque appel de *TH_Game*, qui rajoute à la distance parcourue la distance que le joueur a parcouru entre deux appels du *thread*. La ligne pour faire cet ajout est : *distanceTraveled += Math.floor(speed/3.6)*. On divise par 3.6 pour convertir la distance en mètre par seconde, on sait qu'un *tick* est égal à 16 millisecondes (*TH_Game.GAME_SPEED*, permet d'avoir 60 FPS) ainsi on fait une horloge de façon à mettre à jour à chaque seconde la distance parcourue, de plus dans la méthode *addDistanceTraveled()*, on décrémente le *timer*.

Cette classe contient aussi le système de vie du joueur, le nombre maximum de vie est défini par la constante MAX_LIVES. Le joueur perd une vie lorsqu'il heurte un obstacle, en plus de perdre la moitié de sa vitesse. Le nombre de vie a zéro signifie un *Game Over*.

Nous allons maintenant regarder ce qui concerne la route. La route est définie dans la classe *Road*, une route est composée de deux *arrayList* un contenant les *curbs* et un contenant les obstacles.

Les *curbs* correspondent aux différentes parties de la route, nous avons implémenté un changement de couleur afin de donner un effet de profondeur. Les *curbs* sont de gros polygones, chaque *curbs* a donc une couleur qui alterne avec le suivant, une hauteur et enfin une coordonnée y. Un *curb* peut aussi être appelé *special curbs* si celui-ci contient une *gate*, une *gate* sera alors affichée et lorsque le *curb* sera traversé, le temps de la *gate* sera rajouté, ce temps est une constante du nom ADDED_TIME qui peut aussi être modifiée.

Le temps de départ, celui avec lequel le joueur commence, correspond au temps de lorsque l'on passe à travers une *gate* peut aussi être modifié dans la classe *player*, l'attribut *timer.*

Lors d'un virage, le *curb* peut avoir un *xOffset* pour le déplacer en fonction de cet *Offset* et donc le faire se décaler et donner l'illusion de tourner.

Chaque *curb* est composé d'un *arrayList* de segments, chaque *curb* est composé de 10 segments en fonction de sa hauteur totale. La hauteur d'un segment est au minimum de 1 pixel. L'utilisation de segments est de lisser l'affichage de notre route, en effet, les courbes sont réalisées au niveau des segments, qui, grâce à l'*Offset* du *curb* dans lequel ils sont, se déplacent légèrement afin de limiter le décalage entre chaque *curb* lors de courbes. La hauteur d'un *curb* est calculée grâce à une fonction linéaire où : La hauteur est égale à *COEFFICIENT * y1 + ORIGIN*, l'origine est égale à *MAX_HEIGHT – COEFFICIENT * height* (de la fenêtre) et enfin le coefficient directeur est égal à *2 * (MAX_HEIGHT – 1) / Height* (de la fenêtre). La largeur des segments n'est pas égale des deux côtés de la route, en effet, suivant s'il y a un virage ou non, la largeur des segments sera calculée en fonction du coté avec une fonction linéaire qui varie tout le temps.

Toutes ces mécaniques sont nécessaires pour faire en sorte que la route évolue en fonction de ce que fait le joueur.

La route est composée d'un deuxième type d'éléments, les obstacles. Le nombre maximal d'obstacle sur la route est défini par une constante *MAX_OBSTACLE* (dans *Road*). Un obstacle est associé à un *curb* à la création de ce dernier, lorsqu'il est mis à jour des *curbs*, les obstacles sont aussi mis à jour de façon à ce qu'ils suivent bien le *curb* qui leur a été assigné. Chaque obstacle a un *sprite* aléatoire afin d'afficher différents obstacles. On calcule les dimensions d'un obstacle à partir de la largeur du *curb* qui lui est associé : *with = curb_with / 2* et *heigth = width * 0.6*.

Pour faire les virages de la route, nous avons utilisé un *iterator* sur un *arrayList* contenant des valeurs de 1 à *MAX_CURVE* pour les virages dans un sens et de -1 à – *MAX_CURVE* pour les virages dans l'autre sens, on a un pas pour l'*iterator* pour contrôler la courbure du virage, il s'agit de la constante *TURNING_SPEED*. En plus de notre premier *iterator*, on en a un second qui sert à rajouter un *Offset* pour chacun des segments en fonction du nombre de segment, ce dernier est calculé à l'aide du nombre de segments lors de l'*update* précédente du *curb*, cela évite de devoir faire plusieurs fois les mêmes calculs.

En ce qui concerne le *Main*, il crée le *Display* ce qui lance tout le processus de lancement du jeu.

Résultats

Figure 9 : Aperçu du Main menu

Figure 7 : Aperçu début de partie

Figure 8 : Aperçu d'un checkpoint

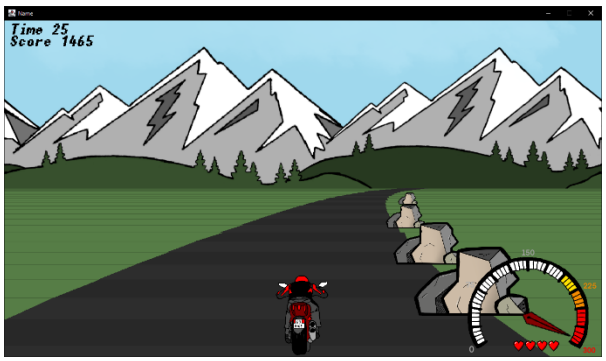

Figure 6 : Aperçu de la route sans gates

Documentation Utilisateur

Pour pouvoir jouer au jeu il y a quelques petites choses à savoir. Tout d'abord, un *IDE Java* est nécessaire (ou *Java* seul si jamais vous avez fait un *export .jar* en exécutable), avec au moins la version 11 de la *JDK* ou *java* version au moins 1.8. Si vous êtes dans le premier cas et que vous avez un *IDE Java*, il vous faut importer le projet dans votre *IDE*, sélectionner la classe *Main* et enfin lancer l'application en cliquant sur *Run as Java Application*. Utilisez la souris pour sélectionner ce que vous voulez faire dans le menu principal, une fois dans le jeu vous pouvez utiliser les flèches directionnelles afin de faire bouger la moto et de le faire accélérer et freiner. Si vous être dans le second cas, que vous possédez un exécutable *.jar*, il vous suffira de double cliquer sur le fichier exécutable et enfin de respecter les mêmes conditions énumérées ci-dessus.

Documentation Développeur

Le jeu est assez simple à comprendre mais il possède malgré tout quelques complexités. Les packages contenant les classes les plus importantes sont :

- *Road*, contient toutes les classes concernant la route (la partie *model*)
- *Scenes*, contient toutes les classes concernant le *management* du jeu
- *Threads*, contient tous les *threads* servant aux bons fonctionnements du jeu
- La classe *Player*
- La classe *Main*

Il y a pas mal de constantes qui peuvent influer sur le fonctionnement premier du jeu :

- DISTANCES_BW_GATES : distance entre deux gates
- MAX_LIVES : Nombre de vie maximum
- GATE_ADDING_TIME : Temps récupéré par passage de *gate*
- MAX_SPEED : Vitesse max du joueur
- MOVE_SPEED : Vitesse de mouvement latéral du joueur

Il y a de nombreuses fonctionnalités que nous n'avons pas pu implémenter par faute de temps mais que nous implémenterons sans doute lors de la prochaine mise à jour de notre projet, parmi ces fonctionnalités on retrouve une animation lorsque le joueur tourne ou encore l'implémentation d'adversaires.

Conclusion et Perspectives

Nous avons réussi à implémenter de nombreuses fonctionnalités, nécessaires et optionnelles, toutefois il nous reste encore beaucoup de travail si nous voulons faire un jeu qui nous semblera complet et fonctionnel. Cependant, nous avons rencontré quelques difficultés comme par exemple l'implémentation des courbes sur la route ou encore certains affichages pour la route en trois dimensions. Ce projet nous a permis d'apprendre à programmer tout en respectant une méthode de programmation, ici le modèle *MVC*. Le fait de devoir séparer la vue, du modèle et de l'affichage à quelque peu changé notre façon de programmer, nous en ressortons une expérience non négligeable qui nous aidera sans l'ombre d'un doute dans notre avenir. Il reste encore de nombreuses fonctionnalités à implémenter dans ce projet, comme par exemple un system de *settings*, des adversaires, des éléments de décor ou encore des animations pour le joueur et ses adversaires pour rendre le jeu plus vivant.