

Rapport Projet Individuel

I. Introduction

Le but de ce projet est de créer un jeu inspiré de Ketchapp Circle.

Un ovale se déplace le long d'une ligne brisée généralement aléatoirement. Le joueur cherche à éviter de sortir de la ligne pour aussi longtemps que possible

Le joueur clique sur l'écran pour faire monter l'ovale, qui va redescendre ensuite tout seul.

Ici on cherche un jeu codé en Java qui va utiliser des notions d'interfaces interactives et de programmation concurrente (utilisation de threads).



II. Analyse

A. Analyse globale

1. Interface graphique avec l'ovale, la ligne brisée et le fond d'écran

L'interface graphique combine plusieurs éléments visuels pour offrir une expérience immersive.

L'affichage repose sur un ovale, représentant l'élément central du jeu, une ligne brisée, qui structure le parcours, et un fond d'écran dynamique, composé de trois plans de montagnes superposés, pour donner de la profondeur à notre jeu.

2. Contrôle du mouvement de l'ovale

Le mouvement de l'ovale est un élément central de l'interactivité du projet, offrant à l'utilisateur un moyen direct d'influencer le déroulement du jeu. L'objectif est de garantir un contrôle intuitif, où un simple clic de souris permet d'agir sur son déplacement, tout en assurant une certaine fluidité et réactivité.

3. Génération et défilement automatique de la ligne brisée

Pour créer l'illusion de progression du cercle dans le jeu, la ligne brisée et les éléments du décor doivent défiler continuellement à l'écran. Pour garantir un mouvement fluide et ininterrompu, une génération automatique du décor est nécessaire, assurant ainsi un renouvellement constant des éléments visuels.

4. Système de score et fenêtre de fin

Enfin, le jeu doit avoir un objectif et une condition de fin. Pour cela, un système de score est mis en place, permettant d'évaluer la performance du joueur. De plus, une condition de défaite est intégrée, déclenchant un écran de fin affichant le score atteint, offrant ainsi une conclusion claire à chaque partie.

B. Analyse détaillée

1. Interface graphique avec l'ovale, la ligne brisée et le fond d'écran

a) Mise en relation entre le système de coordonnées du modèle à ceux de l'affichage

→ Difficulté moyenne, Priorité 1

b) Dessin du cercle

→ Difficulté basse, Priorité 1

c) Dessin d'une rangée de montagne

→ Difficulté moyenne, Priorité 2

d) Dessiner les trois rangées de montagnes pour donner un effet de profondeur

→ Difficulté basse, Priorité 3

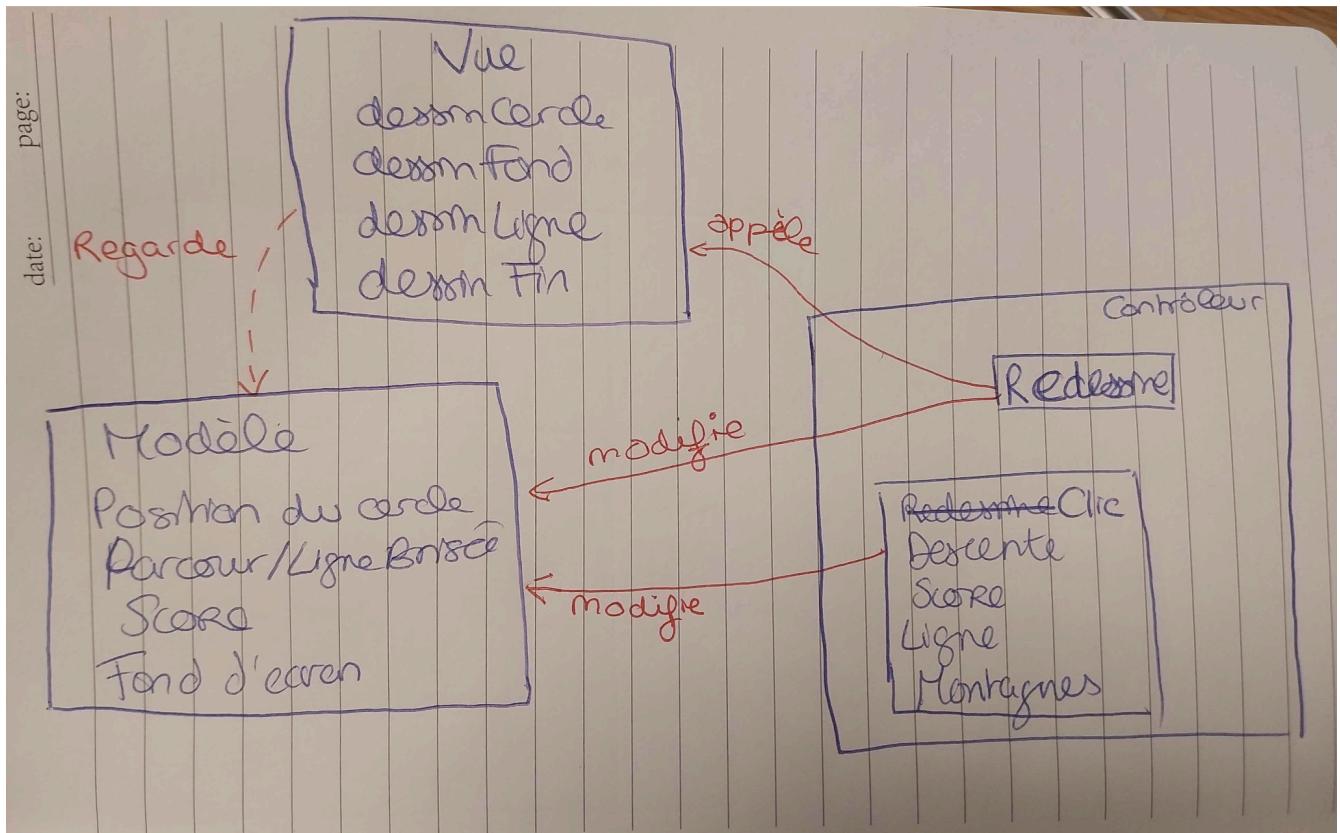
e) Dessin de la ligne brisée

→ Difficulté moyenne, Priorité 1

2. Contrôle du mouvement de l'ovale
 - a) Faire monter le cercle lors qu'on clic
→ Difficulté moyenne, Priorité 1
 - b) Faire descendre le cercle automatiquement sinon
→ Difficulté basse, Priorité 1
 - c) Ajout d'une vitesse pour créer un mouvement de saut
→ Difficulté moyenne, Priorité 2
3. Génération et défilement automatique de la ligne brisée
 - a) Génération automatique du prochain point de la ligne
→ Difficulté moyenne, Priorité 1
 - b) Défilement de la ligne
→ Difficulté basse, Priorité 1
 - c) Génération automatique de la prochaine montagne
→ Difficulté moyenne, Priorité 2
 - d) Défilement des montagnes
→ Difficulté basse, Priorité 2
 - e) Donner à chaque rangée des montagnes une vitesse, pour créer un effet de profondeur
→ Difficulté basse, Priorité 3
4. Système de score et fenêtre de fin
 - a) Creation du score
→ Difficulté basse, Priorité 2
 - b) Incrémentation du score toute les secondes
→ Difficulté basse, Priorité 2
 - c) Affichage du score
→ Difficulté basse, Priorité 3
 - d) Faire perdre le joueur si il touche la ligne
→ Difficulté haute, Priorité 1
 - e) Arrêter le jeu si la condition de perte est atteinte
→ Difficulté moyenne, Priorité 2
 - f) Dessin de la fenêtre de fin
→ Difficulté moyenne, Priorité 3

III. Plan de développement

IV. Conception générale

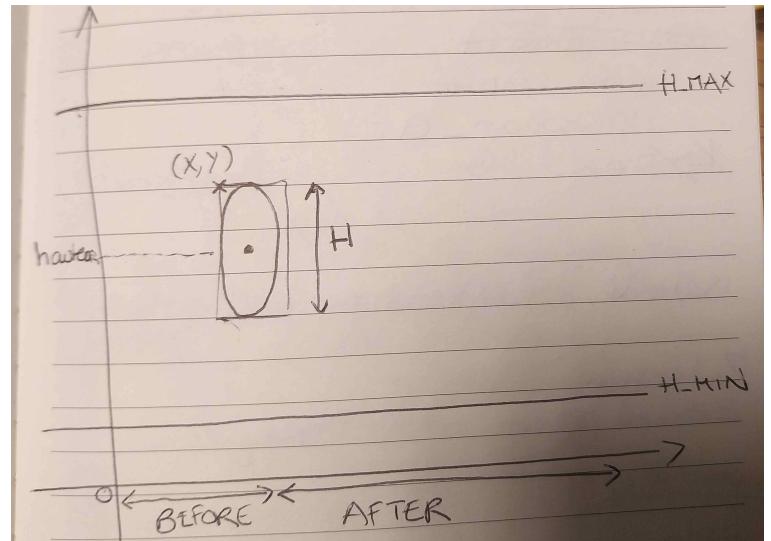
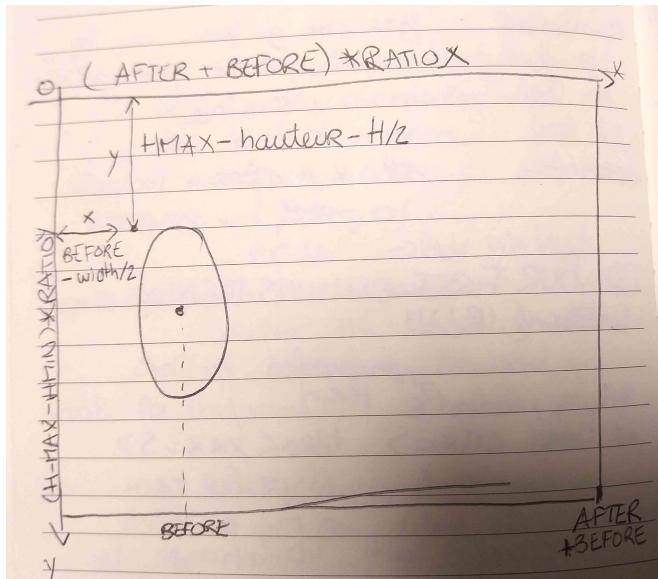


Dans ce schéma, le Modèle représente les données essentielles à la modélisation du jeu, telles que la position du cercle, le score et les éléments de l'environnement comme le fond de l'écran ou le parcours en ligne brisée. Il est directement modifié par le Contrôleur, qui gère les interactions utilisateur, notamment les clics, et applique les modifications nécessaires au modèle. Le Contrôleur intervient lorsqu'un événement est détecté, comme un clic ou l'expiration d'un délai, et met alors à jour le modèle en conséquence.

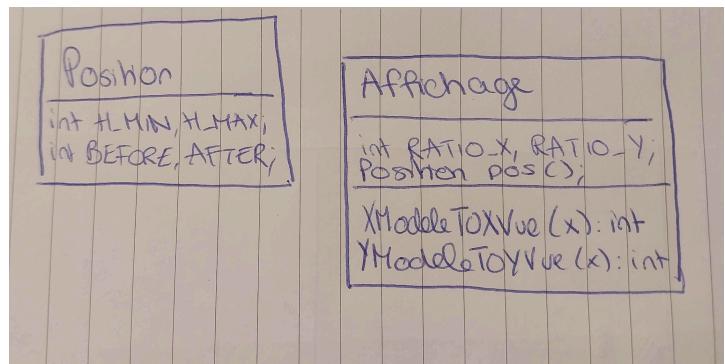
La Vue est responsable de l'affichage des informations à l'utilisateur. Elle récupère les données du modèle et les traduit sous forme visuelle en dessinant les éléments du jeu, comme le curseur, le fond et la ligne. La communication entre les composants suit le principe du MVC : la Vue regarde le Modèle, mais ne le modifie pas directement ; le Contrôleur modifie le Modèle en réponse aux actions de l'utilisateur ; et enfin, le Modèle notifie la Vue des changements pour mettre à jour l'affichage.

V. Conception détaillée

1. Interface graphique avec l'ovale, la ligne brisée et le fond d'écran
- a) Mise en relation entre le système de coordonnées du modèle à ceux de l'affichage



Ici, il était nécessaire de mettre en relation le système de coordonnées du modèle avec celui de la vue. Pour cela, j'ai créé deux fonctions dans la classe Affichage qui, en s'appuyant sur les constantes de la classe Position, permettent de convertir n'importe quelle valeur de x du modèle en sa valeur équivalente sur l'axe x ou y de la vue.



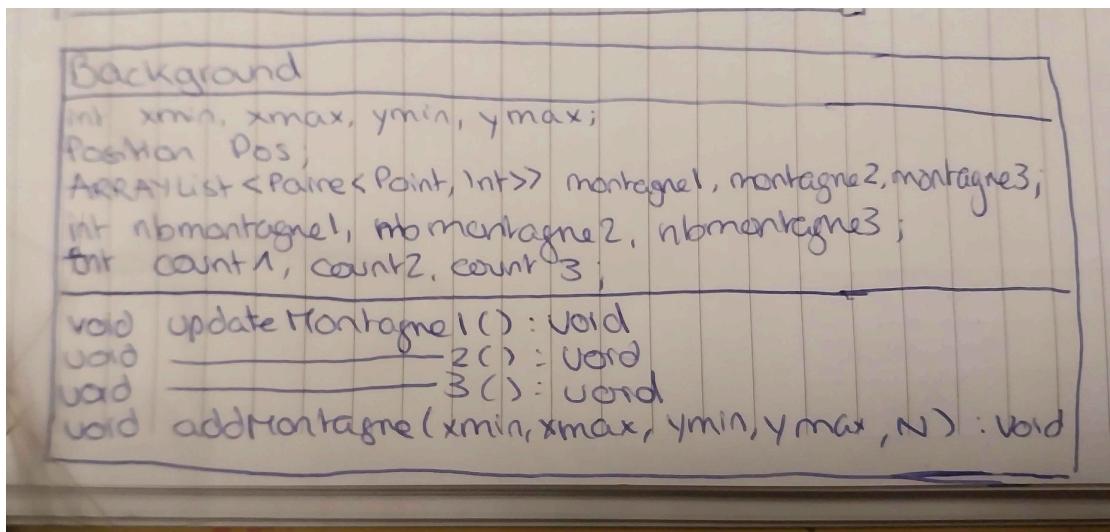
- b) Dessin du cercle, d'une rangée de montagne, de la ligne brisée

```

JANEL
Affichage
int ratioX, ratioY;
int largeur, hauteur;
int CircleX, CircleY, circleWidth, circleHeight;
Position pos;
Parcours ligne;
Background fond;
Score sc;
dessine() : void

```

Pour chaque élément du décor, j'ai créé une fonction auxiliaire appelée lors de paint(). La classe Affichage y accède via son attribut fond, qui fait référence à la classe Background.



c) Dessiner les trois rangées de montagnes pour donner un effet de profondeur
La fonction utilisée pour dessiner chaque rangée de montagnes est la même, seule la hauteur varie plus ou moins.

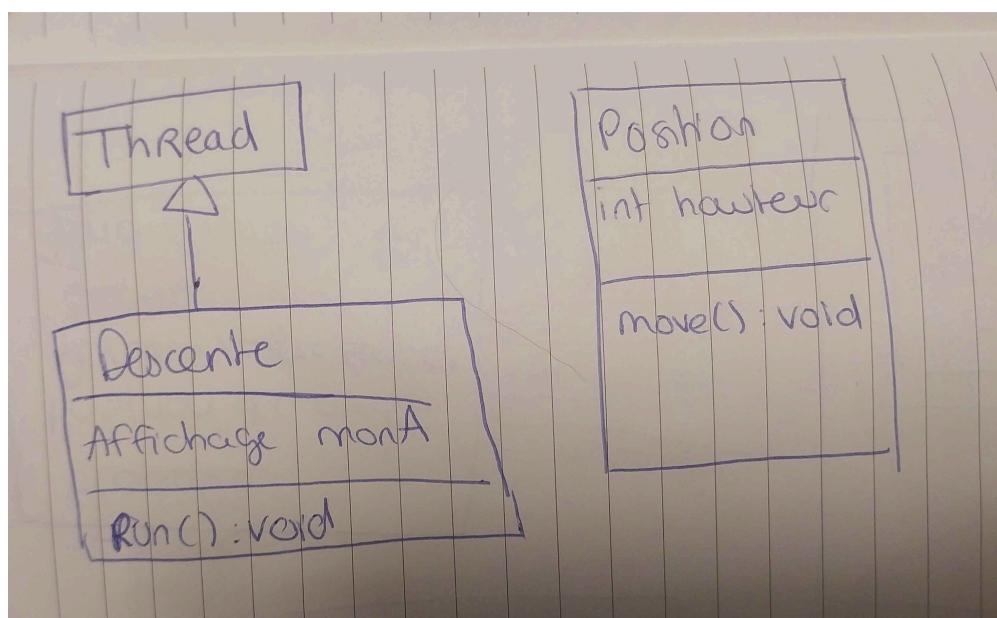
2. Contrôle du mouvement de l'ovale

a) Faire monter le cercle lors qu'on clic

Dans la classe Position, une fonction jump() est définie et appelée par la classe ReactionClic, qui implémente MouseListener. Les classes implémentant MouseListener possèdent des méthodes spécifiques qui s'exécutent en réponse aux actions de l'utilisateur sur la souris, comme le mouvement ou les clics. Ici, nous avons redéfini la méthode mouseClicked() pour qu'elle appelle jump().

La classe ReactionClic dispose d'un attribut de type Affichage, ce qui lui permet d'accéder à l'objet Position et ainsi de déclencher le saut au bon moment.

b) Faire descendre le cercle automatiquement sinon



J'ai une classe qui implémente Thread de Java et qui, dans sa méthode run(), appelle la fonction move() de Position. Cette dernière modifie alors l'attribut hauteur de Position, permettant ainsi de mettre à jour sa position dynamique.

c) Ajout d'une vitesse pour créer un mouvement de saut

Ici, seule la classe Position a été modifiée. J'y ai ajouté un nouvel attribut vitesse et ajusté les fonctions move() et jump() afin qu'elles utilisent cet attribut pour gérer les déplacements.

3. Génération et défilement automatique des éléments du fond

a) Génération et défilement automatique du prochain point de la ligne

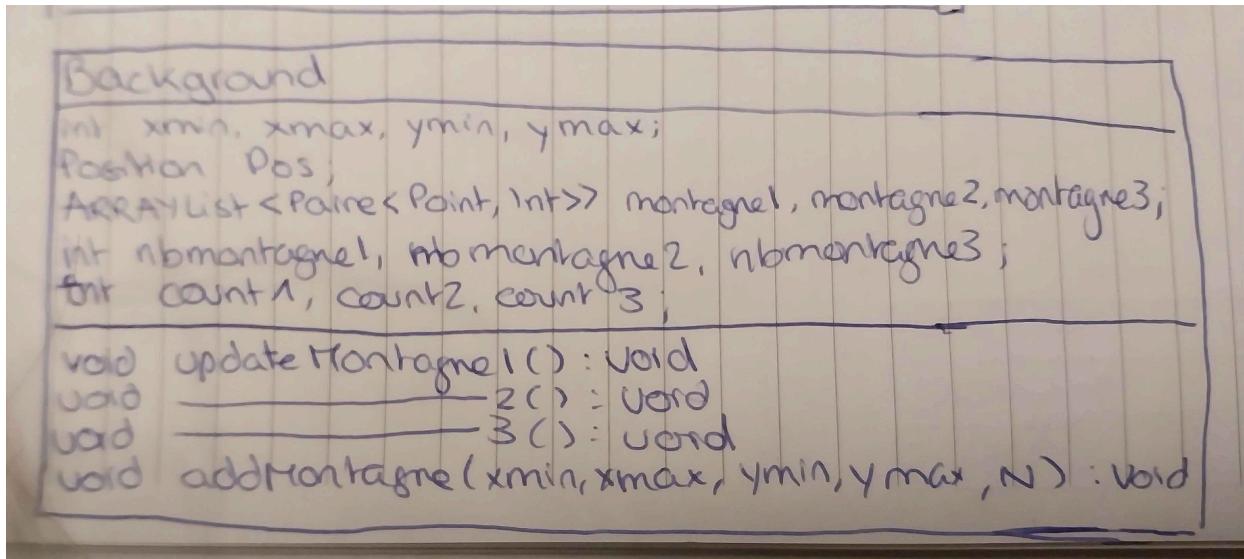
La ligne brisée est modélisée par une liste de points qui se succèdent. Chaque point est généré aléatoirement en respectant les limites définies par les attributs de la classe Parcours. Ainsi, l'ordonnée d'un point doit être comprise entre 'HAUTEUR_MIN' et 'HAUTEUR_MAX', tandis que son abscisse doit être située entre 'X_MIN' et 'X_MAX' au-delà de l'abscisse du point précédent.

Les fonctions move() et remove() de la classe Parcours assurent la génération et le défilement automatique de la ligne. Elles sont appelées par la classe Ligne, qui implémente Thread. La fonction move() fait avancer tous les points, tandis que remove() supprime ceux qui ne sont plus visibles et en ajoute de nouveaux pour que la ligne se prolonge indéfiniment. Pour garantir cette continuité, les nouveaux points sont ajoutés avant même d'être visibles, lorsque leur abscisse dépasse la limite de la fenêtre.

The image shows handwritten code for a class named 'Parcours'. The code is written in blue ink on lined paper. It includes declarations for variables (int Hauteur_min, Hauteur_max; int x_min, xmax; Position pos), a member variable (ArrayList<Point> points; Point pt_contact), and several methods (move():void, remove():void, add():void, point de contact():void). The code is enclosed in a rectangular border.

```
Parcours
int Hauteur_min, Hauteur_max;
int x_min, xmax;
Position pos
ArrayList<Point> points; Point pt_contact;
move():void
remove():void
add():void
point de contact():void
```

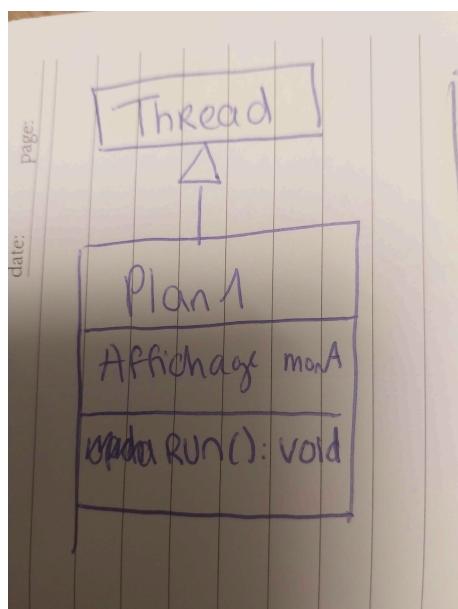
b) Génération automatique de la prochaine montagne



Chaque rangée de montagnes est modélisée par une liste de paires. Le premier élément de chaque paire est un objet de la classe Point, représentant le sommet de la montagne, et le second est un entier indiquant la moitié de la longueur de sa base. Pour chaque rangée, un nombre prédéfini N de montagnes est généré, ce qui permet de déterminer précisément dans quelle section de la fenêtre chaque sommet doit apparaître afin de former une rangée harmonieuse.

De la même manière, une nouvelle montagne est ajoutée dès que l'on a avancé de $1/N$ * largeur, où largeur correspond à la largeur de la fenêtre. Cependant, pour éviter une apparition brusque à l'écran, la montagne est générée juste au-delà de la limite de la fenêtre, garantissant ainsi une transition fluide dans le décor.

c) Défilement des montagnes



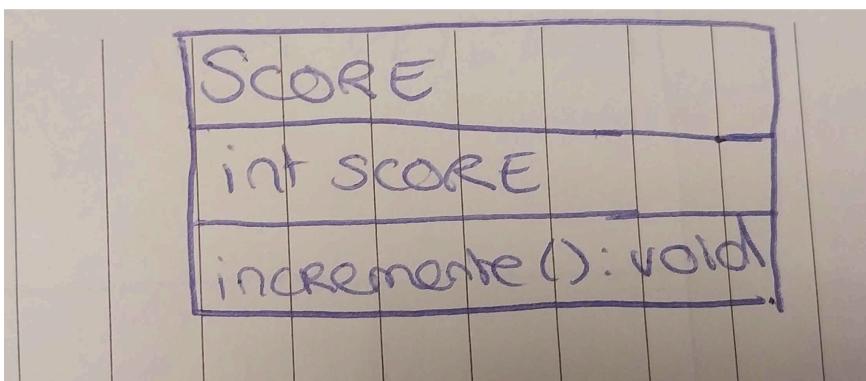
Pour chaque rangée de montagnes, modélisée dans la classe Background, j'ai créé une classe Thread associée, de la forme comme indiqué à gauche. Grâce à son attribut de type Affichage, ce thread appelle la fonction appropriée pour mettre à jour les montagnes. Ainsi, la classe PlanAvant exécute updateMontagneAvant() de Background, la classe PlanMilieu appelle updateMontagneMilieu(), et la classe PlanArriere exécute updateMontagneArriere(). Cela permet de gérer indépendamment le déplacement de chaque plan de montagnes.

- d) Donner à chaque rangée des montagnes une vitesse, pour créer un effet de profondeur

Pour cela, j'ai simplement décidé de ne pas ajouter d'attributs supplémentaires. Au lieu de cela, le changement de vitesse est directement perceptible grâce au fait que chaque montagne possède sa propre fonction de mise à jour, ce qui permet de gérer leur déplacement de manière indépendante.

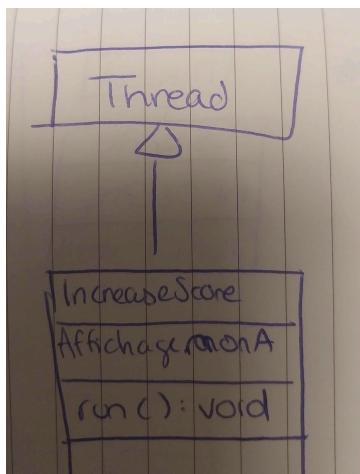
4. Système de score et fenêtre de fin

- a) Creation du score



J'ai dû créer une nouvelle classe Score dans mon modèle, avec un attribut servant à enregistrer le score et une fonction pour l'incrémenter.

- b) Incrémentation du score toute les secondes



J'ai également ajouté une autre classe qui étend Thread dans mon contrôleur, afin que le score augmente automatiquement chaque seconde. Comme toutes les autres threads, elle possède un attribut de la classe Affichage qui permet de déterminer quand elle doit s'arrêter et d'accéder au score qu'il faut augmenter.

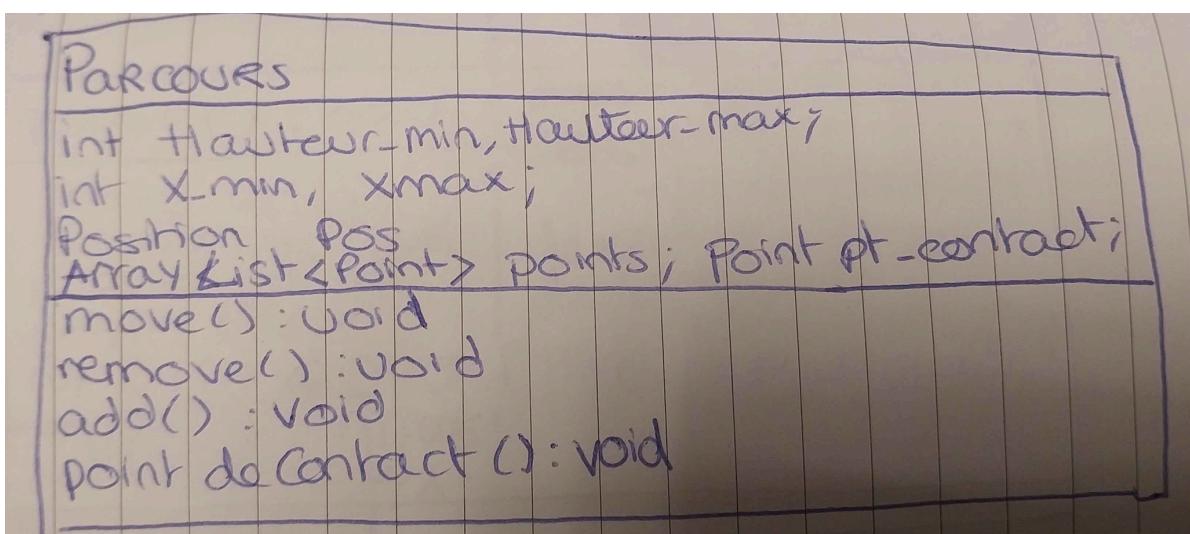
- c) Affichage du score

Comme mentionné précédemment, la classe Affichage possède un attribut de type Score, qui permet d'accéder au score actuel de notre modèle. Il m'a donc suffi d'ajouter une fonction dessineScore() à la classe Affichage et de l'appeler dans paint() pour afficher le score en haut à droite de la fenêtre.

- d) Faire perdre le joueur si il touche la ligne et arrêt de jeu si la condition de perte est atteinte

Dans la classe Parcours, qui modélise la ligne brisée, un attribut entier ‘point_contact’ est défini pour indiquer la hauteur de la ligne à l’abscisse Position.BEFORE, qui correspond à la position du cercle. Cet attribut permet de déterminer l’altitude exacte de la ligne brisée à l’endroit où se trouve le joueur.

Dans la classe Perdu, qui implémente Thread, la méthode calcul_contact() est appelée pour vérifier si ‘point_contact’ correspond aux extrémités du cercle. Cette fonction recalcule la position du point de contact en identifiant les deux points de la ligne brisée qui encadrent actuellement le cercle. Si une collision est détectée, l’attribut ‘jeuEstFini’ de la classe Affichage est mis à true, ce qui entraîne automatiquement l’arrêt des threads et la fin du jeu.



Parcours

```
int Hauteur_min, Hauteur_max;
int x_min, xmax;
Position pos;
ArrayList<Point> points; Point pt_contact;
move(): void
remove(): void
add(): void
Point deContact(): void
```

- e) Dessin de la fenêtre de fin

Dans ma classe Affichage, j’ai un attribut ‘jeuEstFini’ qui permet de suivre l’état du jeu. Le thread Perdu, qui possède un attribut de type Affichage, surveille les conditions de défaite. Lorsque l’une d’elles est atteinte, il met à jour ‘jeuEstFini’ en le passant à false. Cela permet d’interrompre les autres threads et d’empêcher toute action supplémentaire. Ensuite, j’appelle la fonction responsable d’afficher l’écran de fin de jeu, garantissant ainsi une transition fluide vers la conclusion de la partie.

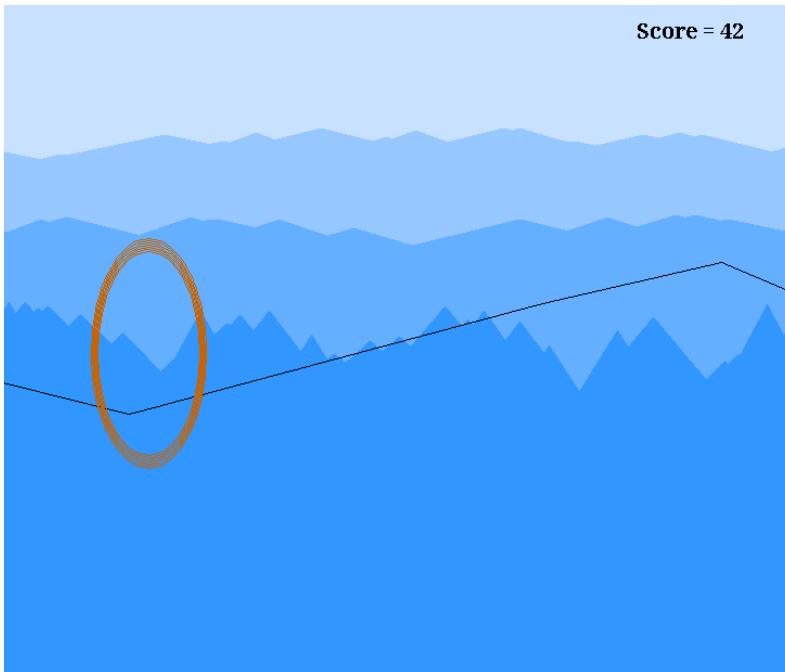
5. Rapport d’étonnement sur l’utilisation de Copilot

Après avoir testé Copilot, je trouve que c’est un outil vraiment utile pour le développement, mais j’ai remarqué quelques points à prendre en compte. Par exemple, quand j’ai voulu utiliser une bibliothèque précise, si elle n’était pas bien définie dans le fichier, Copilot avait tendance à suggérer une autre bibliothèque qui fait quelque chose de similaire mais qui n’est pas exactement ce que je voulais. Ça peut créer des erreurs ou des incohérences pas évidentes à repérer. Aussi, en l’ajoutant au milieu d’un projet

ou d'un fichier déjà bien avancé, j'ai vu qu'il pouvait avoir du mal à comprendre certaines fonctionnalités complexes, surtout si les commentaires n'étaient pas assez détaillés. Il propose parfois des complétions un peu absurdes. Mais dans l'ensemble, j'ai trouvé l'outil sympa et efficace : il m'a bien aidé à écrire plus vite et à automatiser certaines parties du code.

VI. Résultat

Finalement, le jeu se lance dès l'exécution du code. L'objectif est d'empêcher le cercle de toucher la ligne le plus longtemps possible en cliquant avec la souris. L'affichage graphique met en scène trois rangées de montagnes utilisant un dégradé de couleurs et des vitesses différentes, créant un effet de profondeur immersif.



Le score, affiché en haut à droite de la fenêtre, correspond au nombre de secondes écoulées depuis le début de la partie et augmente progressivement. Si le cercle touche la ligne, l'affichage se fige et une fenêtre blanche apparaît au centre de l'écran, indiquant la défaite ainsi que le score atteint



VII. Documentation utilisateur

Bienvenue dans notre jeu d'adresse et de réflexes ! L'objectif est de maintenir le cercle en mouvement et d'éviter qu'il ne touche la ligne le plus longtemps possible. Grâce à un affichage dynamique avec un effet de profondeur et un suivi du score en temps réel, ce jeu vous offrira un défi captivant.

Installation et Lancement

Assurez-vous d'avoir installé Java. Ensuite il suffit d'ouvrir le dossier et de lancer le main. Le jeu démarre automatiquement dès l'exécution.

Règles du Jeu

Un cercle apparaît à l'écran et descend vers une ligne fixe. Vous devez cliquer sur la souris pour maintenir le cercle en l'air et éviter qu'il ne touche la ligne. Un compteur en haut à droite indique votre score. Si le cercle touche la ligne, c'est perdu!

Clic gauche : Maintient le cercle en mouvement.

Si vous galérez à augmenter votre score, je vous conseille d'anticiper la descente du cercle et d'adapter vos clics pour maintenir un contrôle optimal. L'essentiel c'est de rester concentré et de vous amuser en défiant vos propres records !

VIII. Documentation développeur

Pour bien comprendre le fonctionnement général du jeu, il est essentiel d'examiner plusieurs classes clés.

La classe Main joue un rôle central puisqu'elle contient la méthode `main()` et assure l'initialisation du jeu en créant les différents objets et en lançant les threads responsables de l'affichage, de la gestion physique et du score.

La classe Perdu est également primordiale car elle contrôle l'arrêt du jeu et la gestion des threads, garantissant ainsi la bonne transition vers l'écran de fin.

Les classes Position et Parcours sont les deux éléments clés du modèle sur lesquels repose tout le jeu. Sans elles, le jeu ne pourrait pas fonctionner, car elles définissent les éléments essentiels du gameplay, le cercle est la ligne brisée. La classe Position gère la localisation et les déplacements du cercle, tandis que Parcours définit la ligne brisée qui représente le terrain sur lequel le joueur doit évoluer. Ces classes modélisent l'environnement dans lequel le jeu prend vie et permettent d'assurer une logique cohérente entre les actions du joueur et les mécaniques du jeu.

Le comportement du jeu peut être ajusté en modifiant certaines constantes définies principalement dans la classe Position.

Les constantes '`H_MIN`' et '`H_MAX`' déterminent respectivement la hauteur minimale et maximale que le cercle peut atteindre à l'écran.

La constante '`HAUTEUR`' définit la hauteur d'un saut du cercle, tandis que '`BEFORE`' et '`AFTER`' influencent les marges d'affichage avant et après le cercle.

D'autres paramètres clés incluent '`avancement`', qui règle la vitesse de déplacement de la ligne brisée sur l'axe des abscisses, et '`impulsion`' et '`vitesse`', qui détermine la force du saut du cercle et la vitesse à laquelle il redescend. En ajustant ces valeurs, il est possible de modifier la difficulté du jeu, par exemple en rendant le saut plus haut ou en accélérant la vitesse de descente du cercle. Ces constantes permettent de personnaliser l'expérience du joueur en ajustant le niveau de difficulté et la réactivité des contrôles, ce qui peut offrir une plus grande variété dans le gameplay.

À court terme, les fonctionnalités à développer en priorité incluent l'ajout d'un menu de début de jeu, la sauvegarde du meilleur score et l'ajout d'un bouton permettant de recommencer une partie sans devoir relancer l'application. Actuellement, le joueur est obligé de redémarrer entièrement le programme pour relancer une partie, ce qui nuit à l'expérience utilisateur. Pour implémenter ces améliorations, il serait nécessaire d'ajouter une interface de menu avant le début du jeu, accompagnée d'un attribut booléen indiquant si une partie est en cours. Une fonction dédiée permettrait de réinitialiser les éléments du jeu pour faciliter un re-démarrage rapide. Concernant la sauvegarde du meilleur score, il suffirait d'ajouter un attribut dans la classe '`Main`' et de le transmettre à la classe '`Score`', afin de conserver et comparer les performances du joueur au fil des parties.

Des fonctionnalités qui me paraissent intéressantes, mais qui pourraient être plus longues à implémenter, seraient d'ajouter au menu de début un choix de niveau de difficulté, ainsi qu'un choix du monde dans lequel on joue, ce qui modifierait l'arrière-plan du jeu.

IX. Conclusion et perspectives

En conclusion, j'ai réussi à développer un jeu fonctionnel qui se lance et se termine correctement, tout en intégrant de la programmation concurrente ainsi que des éléments d'interface interactive. Je suis assez satisfaite de mon travail, mais si je devais recommencer, je consacrerais plus de temps à la documentation. J'avais l'impression d'y avoir passé énormément de temps, et pourtant, je me rends compte qu'il y a encore de nombreux aspects où je pourrais approfondir mes explications.

L'une de mes plus grandes difficultés durant ce projet a été la gestion du temps. J'ai sous-estimé le temps nécessaire pour implémenter correctement les fonctionnalités et j'ai également trouvé complexe de bien structurer les classes en suivant l'architecture MVC.

Lors du projet de groupe, je pense qu'il sera essentiel de mieux commenter mon code, de tester plus régulièrement et de documenter au fur et à mesure. De plus, je veillerai à découper encore davantage les fonctionnalités en sous-fonctionnalités afin d'améliorer l'organisation et la clarté du projet.