Projet Personel jeu : MINI Game : Don't Touch

ZHANG Boyuan

1. Introduction

Ce projet est un jeu simple basé sur Java Swing intitulé "MINI Game : Don't Touch". L'objectif est de contrôler un personnage elliptique avec la souris pour sauter sur une ligne brisée en mouvement sans la toucher. Le jeu utilise la bibliothèque Swing de Java pour créer l'interface utilisateur et applique la programmation multithread pour gérer la logique du jeu, telles que le mouvement du personnage, le calcul des scores et la détection des collisions. Nous voulons utiliser l'ellipse pour parcourir la plus grande distance possible, en contrôlant le saut de l'ellipse avec un clic de souris, un clic pour un saut, afin d'éviter la collision avec la ligne brisée en mouvement. Le score et le temps de survie sont proportionnels. Le mouvement de l'ellipse simule la gravité ; si on ne clique pas pendant un certain temps, la vitesse de chute de l'ellipse augmente. Après avoir réussi environ quatre secondes, la difficulté augmente, l'amplitude de la ligne augmente. Une fois que l'ellipse dépasse la ligne, le jeu se termine et le calcul du score s'arrête.

2. Analyse Générale

Fonctionnalités Complétées :

- Contrôle du saut de l'ellipse : Le joueur peut contrôler le saut de l'ellipse avec un clic de souris.
- Accélération après le saut : Simule l'accélération due à la gravité, faisant naturellement tomber le personnage après le saut.
- Génération automatique de la ligne brisée : Le jeu génère aléatoirement des lignes brisées comme obstacles pour le joueur.
- Mouvement et génération automatique de la ligne brisée : La ligne d'obstacle se déplace vers le joueur et génère continuellement de nouveaux segments.
- Détection des collisions : Détecte en temps réel les collisions entre le personnage et les obstacles.
- Visualisation des points de collision : Visualise les points de collision en phase de développement pour faciliter le débogage.

- Calcul des scores : Le score est calculé en fonction du temps de survie du joueur.
- Arrêt du calcul des scores après collision : Le score cesse d'être calculé dès qu'une collision se produit.

Fonctionnalités Non Complétées :

- Ajout de boutons pause et début : Permet au joueur de mettre le jeu en pause et de redémarrer.
- Ajout d'objets collectables autour de la ligne brisée : Augmente l'interaction et le défi du jeu.
- Calcul des scores des objets collectables et effet d'animation : Attribue des points pour les objets collectés et ajoute des effets d'animation pour augmenter l'attrait du jeu.

3. Plan de Développement (Ajusté)

Répartition du Temps pour les Fonctionnalités Complétées :

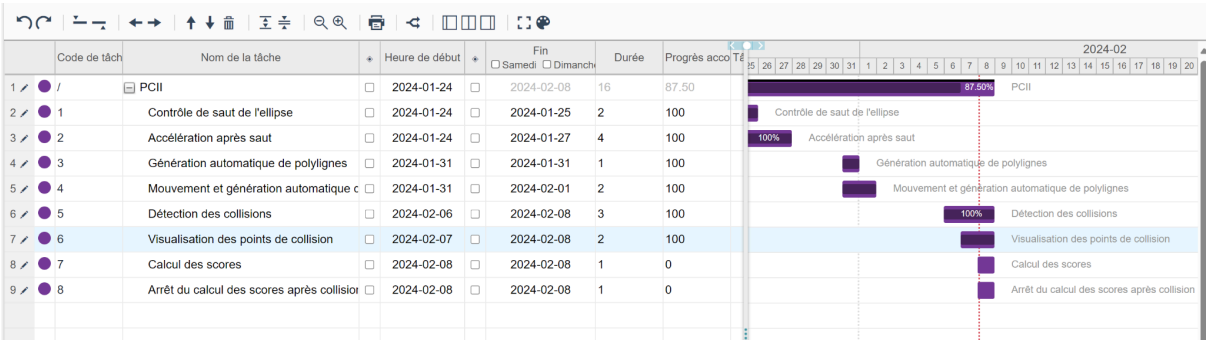
Fonctionnalité	Analyse (min)	Conception (min)	Développement (min)	Test (min)	Total (heures)
Contrôle du saut de l'ellipse	30	30	42	18	2
Accélération après le saut	12	18	24	6	1
Génération automatique de la ligne	30	30	42	18	2
Mouvement et génération automatique	36	42	84	18	3
Détection des collisions	96	120	192	72	8

Visualisation des points de collision	36	42	84	18	3
Calcul des scores	24	24	48	24	2
Arrêt du calcul après collision	24	24	48	24	2

Répartition du Temps pour les Fonctionnalités Non Complétées :

Fonctionnalité	Analyse (min)	Conception (min)	Développement (min)	Test (min)	Total (heures)
Ajout de boutons pause et début	48	60	96	36	4
Ajout d'objets collectables	60	90	120	30	5
Scores des objets et effets d'animation	72	108	144	36	6

Gannt chart



4. Conception Générale

Dans cette section, nous explorons en détail l'architecture MVC du jeu et comment ses différents composants travaillent ensemble pour réaliser les fonctionnalités du jeu. En analysant les rôles et responsabilités des modèles (Model), vues (View) et contrôleurs (Controller), nous pouvons mieux comprendre le processus de conception et d'implémentation du jeu.

Modèle (Model) :

Le modèle est au cœur de la logique du jeu, responsable de la gestion des données et de l'état du jeu. Dans ce jeu, le modèle comprend principalement les classes suivantes :

- `Position` : Gère les informations de position du personnage, y compris sa hauteur dans la fenêtre de jeu et sa vitesse lors du saut. Il fournit des méthodes telles que `move()` pour mettre à jour la position du personnage en simulant l'effet de la gravité, et `jump()`, permettant au personnage de sauter.
- `Parcours` : Responsable de la génération et de la mise à jour du chemin d'obstacles. En maintenant une collection de points de la ligne brisée, la classe `Parcours` génère dynamiquement de nouveaux obstacles en fonction du mouvement du personnage et supprime les obstacles hors de vue pour assurer une expérience de jeu continue. Cette classe calcule également la position relative des obstacles par rapport au personnage pour la détection des collisions.
- `Score` : Suit et met à jour le score du joueur. Cette classe fonctionne dans un thread séparé, augmentant le score à intervalles réguliers jusqu'à la fin du jeu. L'augmentation du score reflète la durée de survie du joueur dans le jeu.

Vue (View) :

La couche de vue gère l'affichage et la mise à jour de l'interface utilisateur du jeu, assurant que le joueur peut voir l'état actuel du jeu, y compris la position du personnage, les obstacles et le score.

- `Affichage` : Au cœur de la couche de vue, la classe `Affichage` hérite de `JPanel` et redéfinit la méthode `paintComponent(Graphics g)` pour dessiner

l'interface du jeu. Elle dépend des données fournies par la couche modèle pour dessiner le personnage et les obstacles, ainsi que pour afficher le score actuel. En outre, lorsque le jeu se termine, la classe `Affichage` est également responsable de l'affichage des informations de fin de jeu.

Contrôleur (Controller) :

Le contrôleur agit comme un intermédiaire entre le modèle et la vue, gérant les entrées de l'utilisateur et mettant à jour l'état du jeu en conséquence.

- `ReactionClic` : Écoute les événements de clic de souris, constituant le principal moyen d'interaction de l'utilisateur avec le jeu. Lorsque le joueur clique sur la souris, la classe `ReactionClic` appelle la méthode `jump()` de la classe `Position` pour faire sauter le personnage. Cette implémentation garantit que les actions de l'utilisateur sont immédiatement reflétées dans la logique du jeu et l'interface utilisateur.

En utilisant l'architecture MVC pour la conception du jeu, nous pouvons séparer efficacement le traitement logique, la gestion des données et l'affichage de l'interface utilisateur, facilitant ainsi le développement et la maintenance. De plus, cette architecture offre une flexibilité pour les extensions futures, telles que l'ajout de nouveaux éléments de jeu ou l'amélioration de l'interface utilisateur, en analysant spécifiquement les responsabilités et les interactions de chaque composant, nous pouvons acquérir une compréhension approfondie de la conception globale et de l'implémentation du jeu.

5. Conception Détaillée

Classe `Position` :

- Fonctionnalité : Gère la position et la vitesse du personnage, simulant son mouvement et son saut dans le jeu.
- Propriétés clés :
  - `hauteur` (Hauteur) : Position verticale du personnage dans la fenêtre de jeu.
  - `vitesse` (Vitesse) : Vitesse du personnage lors du saut, ainsi que les changements de vitesse pendant le saut et la chute.

- Méthodes principales :
    - `move()` : Met à jour la position du personnage en fonction de la vitesse actuelle, simulant l'effet de la gravité sur la chute. Aucun paramètre d'entrée, la sortie est la nouvelle position du personnage.
    - `jump()` : Définit la vitesse initiale du personnage à une valeur ascendante pour réaliser un saut. Aucun paramètre d'entrée, pas de retour, mais met à jour la propriété `vitesse`.

Pseudocode `move()` :

```
move():
  IF hauteur < -160 THEN
    hauteur = -160
  ELSE
    hauteur += vitesse
    vitesse -= 1
```

Pseudocode `jump()` :

```
jump():
  vitesse = ACCEL
```

Classe `Parcours` :

- Fonctionnalité : Génère et met à jour le chemin des obstacles dans le jeu.
- Propriétés clés :
    - `points` : Collection de points formant la ligne brisée des obstacles.
    - `position` : Liée à la position du personnage pour calculer la mise à jour dynamique du chemin.
- Méthodes principales :
    - `initializePoints()` : Initialise les points de départ de la ligne brisée. Aucun paramètre d'entrée, pas de retour, mais remplit la collection `points`.

- `updatePoints()` : Met à jour dynamiquement les points de la ligne brisée en fonction de l'avancement du personnage. Aucun paramètre d'entrée, pas de retour, mais modifie la collection `points`.

Pseudocode `initializePoints()` :

```
initializePoints():
  points = [new Point(20, 130), new Point(80, 100), ...]

updatePoints():
  WHILE points.last().x - position.getAvancement() < 400
    newX = lastPoint.x + X_MIN + Random(X_MAX - X_MIN + 1)
    newY = Random range within game bounds
    points.add(new Point(newX, newY))
```

```
points = [Point(20, 130), Point(80, 100), ...]
```

Pseudocode `updatePoints()` :

```
TANT QUE dernier point de points.x - avancement du personnage < 400 FAIRE
  nouveauX = dernier point.x + X_MIN + Aléatoire(X_MAX - X_MIN + 1)
  nouveauY = Aléatoire dans la plage de jeu
  points.ajouter(Point(nouveauX, nouveauY))
```

Classe `Score` :

- Fonctionnalité : Calcule et met à jour le score du jeu dans un thread séparé.
- Propriétés clés :
  - `score` : Score du jeu, augmentant avec le temps jusqu'à la fin du jeu.
- Méthodes principales :
  - `run()` : Augmente périodiquement le score. Comme corps de thread, aucun paramètre d'entrée, pas de retour, mais met à jour périodiquement la propriété `score`.

Pseudocode `run()` :

```
TANT QUE vrai FAIRE
  PAUSE 1000 millisecondes
  SI jeu non terminé ALORS
```

incrémenterScore(10)

Classe `Collision` :

- Fonctionnalité : Détecte les collisions entre le personnage et les obstacles.
- Propriétés clés :
  - `affichage` : Accède à l'interface du jeu pour déterminer si le jeu est terminé.
  - `position`, `parcours` : Stockent respectivement la position du personnage et le chemin des obstacles pour la détection des collisions.
- Méthodes principales :
  - `checkCollision()` : Calcule la position du personnage et des obstacles pour déterminer s'il y a collision. Aucun paramètre d'entrée, retourne un booléen indiquant la présence d'une collision.

Pseudocode `checkCollision()` :

```
POUR chaque segment dans Parcours FAIRE
  SI position du personnage intersecte le segment ALORS
    RETOURNER vrai
RETOURNER faux
```

6. Résultats

À travers le développement de ce projet, nous avons réussi à créer un jeu interactif de base où le joueur contrôle un personnage elliptique pour sauter entre les obstacles. Le jeu intègre des fonctionnalités essentielles telles que le contrôle des sauts, la génération et le mouvement automatiques des obstacles, la détection des collisions et le calcul des scores, offrant ainsi une expérience de jeu simple mais divertissante.

7. Documentation Utilisateur

Comment Commencer : Exécutez la classe Main pour démarrer le jeu. Un personnage elliptique apparaîtra au centre de l'interface. Le jeu commence automatiquement pour l'instant, avec des projets futurs incluant une fonction de démarrage choisi par l'utilisateur et potentiellement une période d'invincibilité initiale.
Instructions de Jeu : L'objectif est de faire passer l'ellipse aussi loin que possible, en cliquant avec la souris pour la faire sauter. Chaque clic provoque un saut. Évitez les collisions avec les obstacles en mouvement. Le score est proportionnel au temps de survie. La gravité est simulée, donc si vous ne

cliquez pas pendant un moment, la vitesse de chute de l'ellipse augmentera. Après environ quatre secondes, la difficulté augmente, avec une plus grande amplitude dans les mouvements de la ligne.
Condition de Fin : Si l'ellipse dépasse la ligne, le jeu se termine et le calcul du score s'arrête.

8. Documentation pour les développeurs

Besoins environnementaux

- Kit de développement Java (JDK) : L'environnement de développement nécessite l'installation du JDK version 1.8 ou ultérieure.
- Environnement de développement intégré (IDE) : Nous recommandons l'utilisation d'IDE tels qu'IntelliJ IDEA ou Eclipse pour le développement.

Structure et extension des classes

- Classe `Main` : Point d'entrée du programme, initialisant l'interface utilisateur et la logique du jeu. Si vous cherchez à comprendre le flux principal du jeu, c'est ici que tout commence.

Points d'extension :

- Gestion des interactions utilisateur : En ajoutant de nouveaux écouteurs de souris (`MouseListener`), il est possible d'intégrer davantage de fonctionnalités interactives, telles que des boutons de pause et de reprise du jeu.
- Classe `Parcours` : Pour complexifier les défis rencontrés par le joueur, cette classe peut être étendue pour générer des obstacles plus variés et dynamiques.

Fonctionnalités non implémentées et suggestions

- Boutons de pause et de démarrage : Une fonctionnalité essentielle pour améliorer l'expérience utilisateur serait l'ajout de boutons permettant de mettre le jeu en pause et de le redémarrer. Cette fonctionnalité pourrait être intégrée dans la classe `Affichage`, en manipulant l'état du jeu pour contrôler son exécution.
- Objets collectables : L'introduction d'éléments collectables augmenterait l'engagement des joueurs. L'utilisation de timers Swing (`javax.swing.Timer`) pour animer ces objets et les rendre interactifs offrirait une couche supplémentaire de dynamisme et d'attrait. La logique derrière leur collecte et les effets sur le score ou le gameplay pourraient être gérés en étendant la fonctionnalité de la classe `Parcours`.

Pour une version améliorée du projet...

- Premières classes à examiner : Le développeur devrait commencer par la classe `Main`, qui héberge la méthode `main`, le point d'entrée de l'application. Cela lui donnera une vue d'ensemble du flux d'exécution du jeu.
- Principales constantes à modifier : Des constantes telles que les paramètres de vitesse, de gravité, ou les intervalles de génération des obstacles peuvent être ajustés pour modifier la difficulté et le rythme du jeu, se trouvant majoritairement dans les classes `Position` et `Parcours`.
- Fonctionnalités prioritaires non implémentées :
    - L'ajout de boutons de pause et de démarrage serait un premier pas significatif pour améliorer l'interface utilisateur.
    - La génération d'objets collectables avec des animations et leur intégration dans le système de score offrirait une nouvelle dimension au jeu.

Pour développer ces fonctionnalités, j'aurais commencé par structurer l'interface utilisateur avec des boutons intégrés dans la classe `Affichage`, en gérant les états de jeu. Ensuite, pour les objets collectables, j'aurais créé une nouvelle classe pour gérer leurs propriétés et comportements, en les rendant interactifs grâce à des écouteurs d'événements et en intégrant leur logique de scoring au sein du moteur du jeu existant.
9. Conclusion et perspectives

Ce projet a permis de réaliser un jeu simple utilisant Java Swing, approfondissant ainsi notre compréhension de l'interface graphique Java, du traitement des événements et de la programmation multi-thread. Les directions futures pour le développement incluent l'augmentation de la difficulté du jeu, l'amélioration de l'interface utilisateur, l'ajout de nouvelles fonctionnalités (telles que des boutons de pause/démarrage et des objets à collecter) et l'optimisation de la structure du code pour faciliter l'extension et la maintenance.

Au cours de ce projet, les principaux défis rencontrés ont été la gestion de l'accélération de la chute, la génération dynamique des lignes brisées et l'implémentation précise de la détection des collisions.

Défis majeurs

1. **Gestion de l'accélération de la chute** : Initialement, j'avais représenté la vitesse de saut et la vitesse de chute par deux variables distinctes, calculant ainsi la vitesse finale de l'ellipse. Sur les conseils de mon professeur, j'ai simplifié le programme en représentant la vitesse de l'ellipse par une seule variable. À chaque clic pour sauter, une vitesse initiale est attribuée à l'ellipse, qui diminue progressivement, agissant ainsi comme une accélération dans la direction opposée. Cette simplification a non

seulement réduit la complexité du programme mais a également évité de nombreux bugs.

2. **Génération dynamique des lignes** : La mise à jour des points sur la ligne et leur déplacement uniforme vers la gauche étaient initialement codés dans la classe `Parcours`. Cela a rendu la classe très désordonnée et difficile à étendre avec de nouvelles fonctionnalités. Mon professeur a souligné que j'avais confondu les fonctionnalités de contrôle et de vue. Après avoir consulté davantage de documentation sur la classe `Thread`, j'ai finalement résolu le problème efficacement.

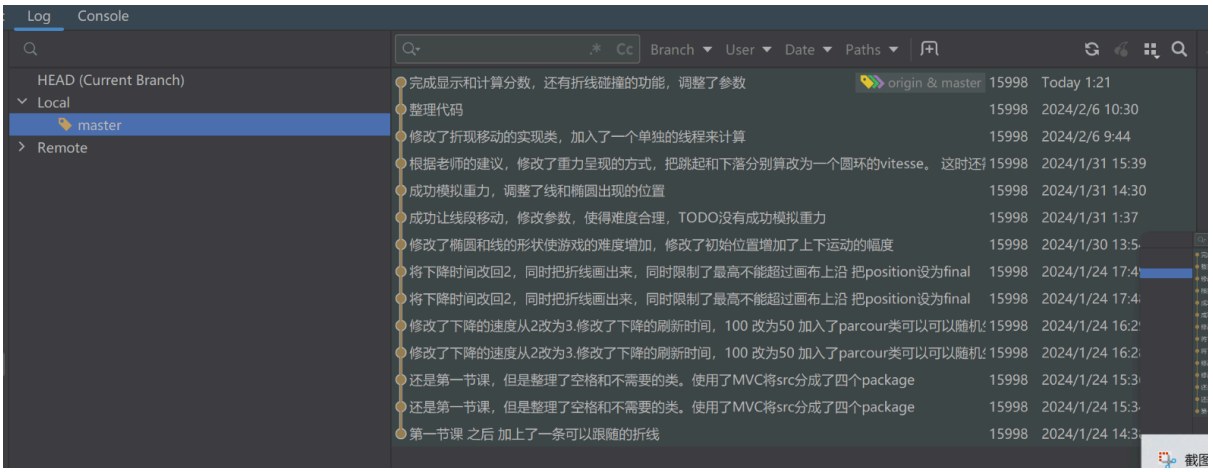
3. **Détection des collisions** : J'avais supposé que les collisions se produiraient toujours entre le deuxième et le troisième point, ce qui rendait la détection des collisions très imprécise en raison de la fluctuation constante du point de collision. En visualisant séparément les trajectoires de mouvement de l'ellipse et de la ligne brisée, j'ai pu clarifier leur interaction. J'ai ensuite opté pour une approche consistant à parcourir la ligne brisée à la recherche du premier point dépassant x=80, utilisé comme p2, le point précédent devenant p1. Ainsi, le point de collision se trouve sur le segment de ligne généré par p1 et p2.

D'autres problèmes mineurs ont été résolus en utilisant des méthodes de visualisation similaires pour le débogage.

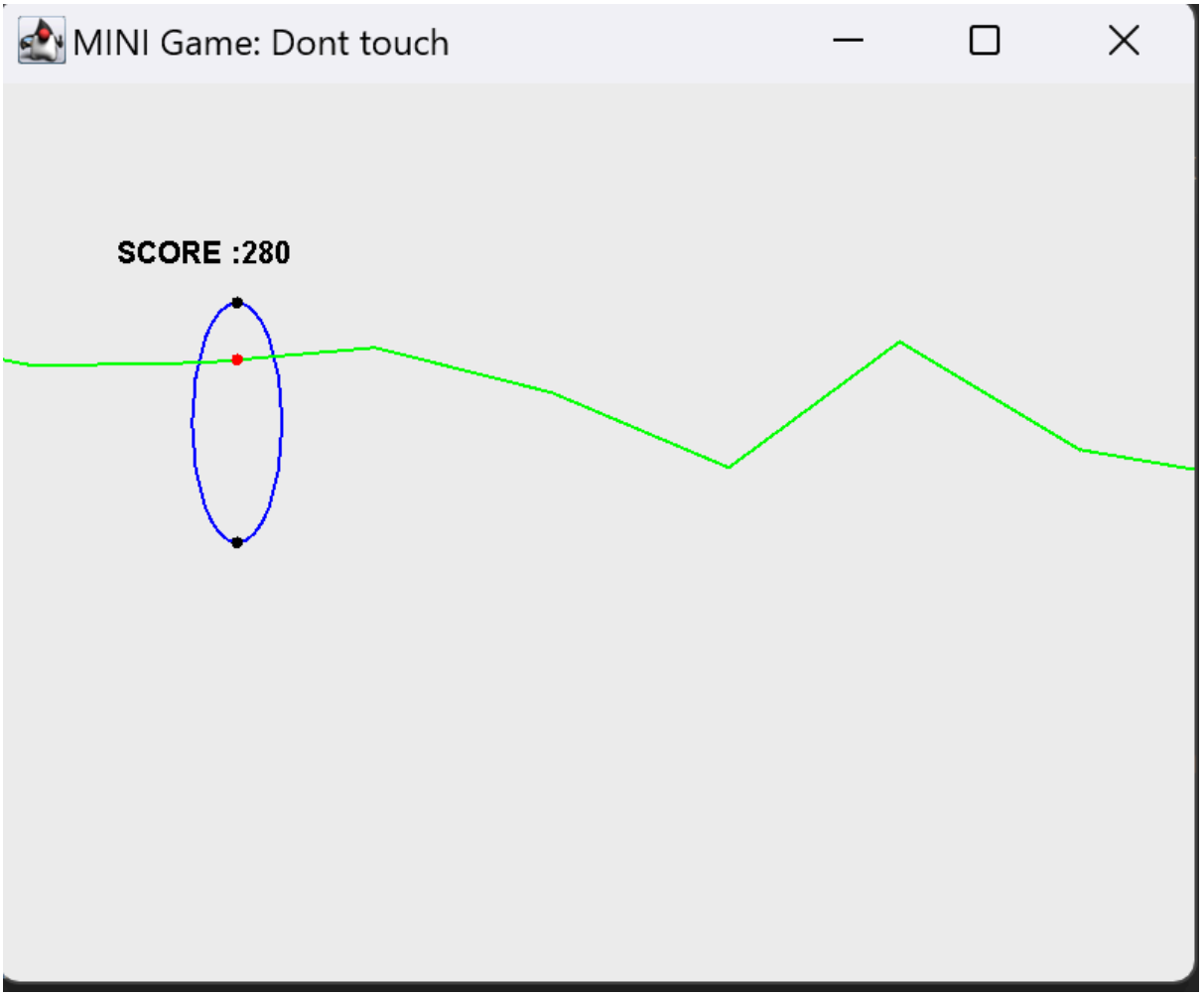
Les connaissances clés acquises comprennent l'utilisation avancée de Java Swing, les techniques de programmation multi-thread et la gestion des états du jeu dans un environnement dynamique. Les travaux futurs se concentreront sur l'amélioration de l'expérience de jeu, l'introduction de nouveaux éléments, l'embellissement de l'interface et l'augmentation de l'interactivité pour attirer davantage de joueurs.

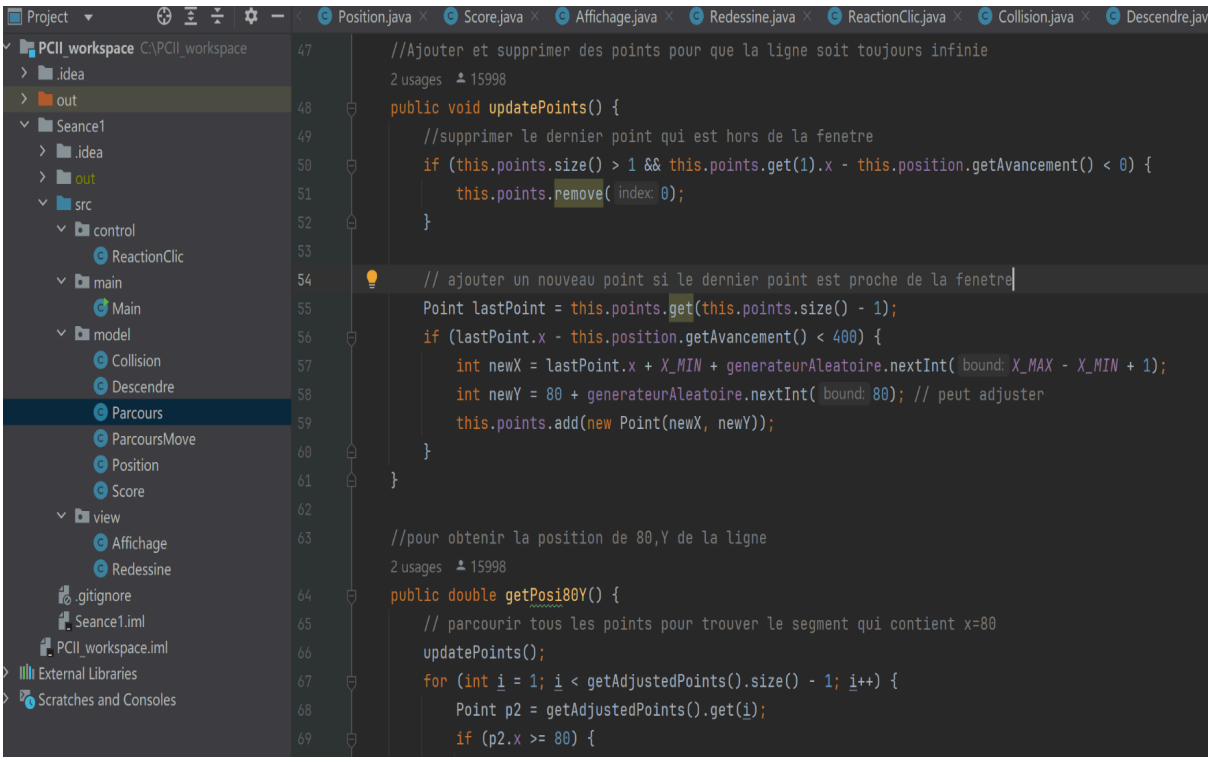
10 ANNEXE Dessines

git history



interface affichage




Project
PCII_workspace C:\PCII_workspace
.idea
out
Seance1
.idea
out
src
control
ReactionClic
main
Main
model
Collision
Descendre
Parcours
ParcoursMove
Position
Score
view
Affichage
Redessine
.gitignore
Seance1.iml
PCII_workspace.iml
External Libraries
Scratches and Consoles
Position.java
Score.java
Affichage.java
Redessine.java
ReactionClic.java
Collision.java
Descendre.jav
//Ajouter et supprimer des points pour que la ligne soit toujours infinie
2 usages 15998
public void updatePoints() {
//supprimer le dernier point qui est hors de la fenetre
if (this.points.size() > 1 && this.points.get(1).x - this.position.getAvancement() < 0) {
this.points.remove(index: 0);
}
// ajouter un nouveau point si le dernier point est proche de la fenetre
Point lastPoint = this.points.get(this.points.size() - 1);
if (lastPoint.x - this.position.getAvancement() < 400) {
int newX = lastPoint.x + X_MIN + generateurAleatoire.nextInt(bound: X_MAX - X_MIN + 1);
int newY = 80 + generateurAleatoire.nextInt(bound: 80); // peut adjuster
this.points.add(new Point(newX, newY));
}
}
//pour obtenir la position de 80,Y de la ligne
2 usages 15998
public double getPosi80Y() {
// parcourir tous les points pour trouver le segment qui contient x=80
updatePoints();
for (int i = 1; i < getAdjustedPoints().size() - 1; i++) {
Point p2 = getAdjustedPoints().get(i);
if (p2.x >= 80) {