ESME SUDRIA

Super JUMP

Projet android studio

Thibault Masse

Nicolas Félis

Bastien Fernandez

Table des matières

1. Introduction

Le jeu Super Jump est une application Android développée en Kotlin contenant également un peu de Java, inspirée du jeu « Doodle Jump ». L'objectif principal est de grimper le plus haut possible en sautant sur des plateformes tout en évitant les monstres et en utilisant des bonus comme des jet packs et des trampolines pour monter plus haut rapidement.

2. Interface Utilisateur et Navigation dans l'application

2.1. Écran d'Accueil

- Sur ce premier écran nous pouvons retrouver un design en lien avec notre jeu avec également un bouton jouer qui sert à passer à l'écran suivant ou là nous pourrons commencer à jouer au jeu.

2.2. Écran de Jeu

- **Objectif :** Monter le plus haut possible en sautant sur des plateformes et en évitant les monstres.
- **Actions principales :**
    - Le joueur peut déplacer son personnage latéralement en bougeant le téléphone pour atteindre les plateformes et ne pas tomber et également éviter les monstres.
    - Les plateformes sont générées aléatoirement, certaines se déplacent d'autre sont fixes et encore d'autre si elles sont touchées disparaissent au bout d'un certain temps.
    - Des bonus comme des jet-packs ou des trampolines peuvent apparaître pour aider le joueur à monter plus vite.
    - Des monstres peuvent apparaitre aléatoirement pour compliquer le jeu.

- **Score :**
    - Le score augmente plus on monte haut.

2.3. Écran de Fin de Partie

- **Conditions de fin :**
    - Le joueur ne rebondis pas sur une platform et tombe.
    - Le joueur touche un monstre et la partie s'arrête.
- **Contenu de l'écran :**
    - Affichage du score final de la partie.
    - Un bouton "Rejouer" permet de relancer une partie rapidement pour rejouer.

3. Code et Fonctionnalités des Class

3.1. Class GameManager

La classe GameManager est essentielle au fonctionnement du jeu et joue le rôle de gestionnaire global des éléments.

3.1.1. Importations

- **android.graphics.** : Permet de manipuler des objets graphiques comme les couleurs, les pinceaux (Paint), ou les images (Canvas).
- **android.view.SurfaceView et SurfaceHolder** : Utilisés pour créer une surface graphique sur laquelle le jeu est dessiné en temps réel.
- **kotlin.random.Random** : Génère des valeurs aléatoires, utilisées ici pour créer des plateformes, des monstres, et des bonus de manière aléatoire.
- **java.lang.Thread.sleep** : Permet de ralentir l'exécution d'une boucle pour gérer le rythme du jeu (contrôle des FPS).

- **Imports spécifiques au jeu (boosts, monsters, plateforms)** : Ces class définissent les bonus, monstres, et plateformes du jeu, qui sont gérés par GameManager.

3.1.2. Variables et Constantes

- **objects et addStack :**
  - objects contient tous les éléments actifs du jeu (plateformes, monstres, etc.).
  - addStack stocke temporairement de nouveaux objets à ajouter dans objects pour éviter les conflits pendant la boucle de mise à jour.
- **score :** Suit la progression du joueur en fonction de la hauteur atteinte.
- **Constantes comme WIDTH et HEIGHT :** Fixent les dimensions de la surface de jeu et servent de base pour positionner les objets.

3.1.3. Méthodes Principales

1. **init (Initialisation)**
   - Configure la peinture pour l'affichage du score et le fond blanc.
   - Génère les premières plateformes pour remplir l'écran.
2. **gameLoop (Boucle du jeu)**
   - Gère les étapes principales :
     - Efface l'écran (drawColor).
     - Dessine chaque objet (plateformes, monstres, joueur).
     - Met à jour les positions et vérifie les collisions.
     - Ajoute ou supprime des objets si nécessaire.
   - En cas de mort du joueur (!player.alive), passe à l'écran de fin via activity.endScreen.

3. **run**
    - Lance la boucle principale du jeu en suivant un rythme constant défini par MS_PER_TICK (contrôle du FPS).
    - Mesure le temps écoulé entre chaque image pour maintenir une fluidité.
4. **generatePlatform**
    - Crée des plateformes ou autres objets selon des probabilités aléatoires.
    - Exemple : une probabilité de 5 % pour créer une MovingPlatform.
5. **moveObjects**
    - Déplace tous les objets vers le bas pour simuler la montée du joueur.
    - Gère la suppression des objets hors écran et le score du joueur.
6. **Gestion du cycle de vie**
    - **onPause et onResume :** Gèrent l'état de la boucle lorsque l'application est mise en pause ou reprise.

3.1.4. Utilité Globale

La classe GameManager orchestre l'ensemble des éléments dynamiques du jeu. Elle maintient la logique de jeu, gère le rendu graphique, et assure une interaction fluide entre le joueur et les objets. C'est le cœur du moteur du jeu, garantissant que tout fonctionne en harmonie.

3.2 Class GameObject

La classe GameObject constitue la base pour tous les éléments du jeu (plateformes, bonus, monstres).

3.2.1. Importations

- **android.graphics.*** : Permet de manipuler des graphiques pour les objets, notamment :
    - RectF : Définit une zone rectangulaire utilisée ici pour la gestion des collisions (hitbox).
    - Paint : Permet de configurer les couleurs et les styles pour le rendu des objets.
    - Bitmap : Utilisé pour gérer les images représentant les objets.

3.2.2. Variables

- **size et pos :** Dimensions et position de l'objet àl'écran.
- **sprite :** Référence à une ressource graphique représentant l'objet.
- **hitbox :** Zone de collision de l'objet, mise à jour lors des déplacements.
- **paint et hitboxPaint :**
    - paint : Utilisé pour dessiner l'objet.
    - hitboxPaint : Débuggage visuel pour afficher les zones de collision.

3.2.3. Méthodes Principales

1. **init**
    - Initialise les propriétés de l'objet.
    - Configure la couleur et la transparence de la hitbox pour un affichage visuel.
2. **isHit(box: RectF)**
    - Vérifie si la hitbox du personnage touche celle d'un autre objet ce qui détermine une collision.

3. **draw(game: GameManager)**
    - Dessine l'objet sur surface graphique.
    - Ajuste la taille et la position de l'image (Bitmap) en fonction des dimensions de l'écran.
4. **checkRemove()**
    - Détermine si l'objet est hors de l'écran (position négative) et le marque pour suppression.
5. **move(inc: Vector)**
    - Met à jour la position de l'objet en un déplacement de gauche à droite basée.
    - Recalcule la hitbox après chaque mouvement.

3.2.4. Utilité Globale

La classe GameObject fournit une base commune pour tous les objets du jeu. Elle simplifie la gestion des collisions, des mouvements, et du rendu graphique. Les classes spécifiques (plateformes, monstres, bonus) héritent de cette structure pour ajouter des comportements uniques tout en réutilisant la logique de base.

3.3. Class MainActivity

3.3.1. Importations

- **android.content.** :
    - Context : Permet d'accéder aux services système nécessaires, comme les capteurs.
    - Intent : Utilisé pour lancer des activités comme l'écran de replay.
- **android.hardware.** : Permet de gérer les capteurs matériels comme le mouvement du téléphone.
- **android.os.Bundle** : Contient l'état de l'activité lors de sa création ou restauration.

- **androidx.appcompat.app.AppCompatActivity** : Fournit les fonctionnalités d'une activité avec prise en charge des versions récentes d'Android.

3.3.2. Variables

- **game :** Instance de GameManager, utilisée pour gérer le moteur de jeu.
- **Sensor :**
    - sensor : Représente le capteur d'orientation utilisé pour contrôler les mouvements du Doodler.

3.3.3. Méthodes Principales

1. **onCreate**
    - Initialise l'interface utilisateur en liant le layout XML à l'activité.
    - Configure le GameManager et les propriétés du capteur d'orientation.
    - Assure que la surface graphique est transparente pour permettre un rendu fluide.
2. **onResume et onPause**
    - Gèrent le cycle de vie de l'application :
        - onResume : Active le capteur et reprend la boucle de jeu.
        - onPause : Désactive le capteur et met en pause la boucle de jeu.
3. **onSensorChanged**
    - Appelée à chaque mise à jour du capteur.
    - Modifie l'orientation horizontale du Doodler en fonction des mouvements détectés du téléphone.
4. **endScreen**
    - Lance l'écran de replay (activité ReplayActivity) avec le score final.

3.3.4. Utilité Globale

La classe MainActivity agit comme un intermédiaire entre le moteur de jeu (GameManager) et les fonctionnalités d'Android. Elle configure le jeu, gère l'interaction avec le capteur, et permet la navigation entre les écrans. C'est le point central qui initialise et supervise l'expérience utilisateur.

3.4. Class MenuActivity

La classe MenuActivity gère l'écran de menu principal du jeu.

3.4.1. Importations

- **android.content.Intent** : Utilisé pour lancer une nouvelle activité, comme MainActivity pour démarrer une partie.
- **android.graphics.RectF** : Définit une zone rectangulaire, ici utilisée pour détecter les clics sur le bouton "Jouer".
- **android.view.*** : Fournit les outils pour gérer les cliques sur l'écran, comme MotionEvent et OnTouchListener.
- **android.widget.ImageView** : Utilisé pour afficher l'arrière-plan ou les éléments graphiques du menu.

3.4.2. Variables

- **playButton :** Définit un rectangle correspondant à la zone cliquable du bouton "Jouer".
- **menuView :** Représente l'arrière-plan ou le layout du menu, qui suis les interactions tactiles.

3.4.3. Méthodes Principales

1. **onCreate**
    - Configure l'interface utilisateur du menu à partir du fichier XML associé (activity_menu).
    - Initialise menuView pour suivre les interactions tactiles avec setOnTouchListener.
2. **onTouch**
    - Vérifie si une interaction tactile correspond à un clic sur le bouton "Jouer" (défini par playButton).
    - Si un clic est détecté ça lance MainActivity pour démarrer une partie.

3.4.4. Utilité Globale

La classe MenuActivity offre un point d'entrée simple et intuitif pour l'utilisateur. Elle permet de naviguer vers l'écran principal du jeu tout en assurant une interaction fluide grâce à la détection tactile.

3.5. Class Platform

La classe **Platform** représente une plateforme dans le jeu Super Jump. Elle sert de base pour les différentes plateformes qui interagissent avec le joueur.

3.5.1. Importations

- android.graphics.RectF : Définition d'une zone rectangulaire, utilisée pour détecter les interactions avec le joueur (cette fonctionnalité est présente mais actuellement commentée dans le code).
- android.util.Log : Permet de journaliser des messages pour le débogage.
- com.doodlejump.GameObject : Classe parent définissant les caractéristiques générales des objets du jeu.

- com.doodlejump.IJumpable : Interface indiquant que l'objet peut interagir avec le joueur lorsqu'il saute.
- com.doodlejump.Player : Représente le joueur du jeu, nécessaire pour gérer les interactions avec les plateformes.
- com.doodlejump.Vector : Classe utilisée pour gérer les positions et dimensions sous forme vectorielle.

3.5.2. Variables

- pos0 : Instance de Vector représentant la position initiale de la plateforme sur l'axe des coordonnées.
- type : Un entier indiquant le type de plateforme (par exemple, les plateformes classiques, mobiles, destructibles, etc.).
- size : Constante définie dans le companion object, de type Vector, représentant les dimensions par défaut d'une plateforme (150 px de largeur, 40 px de hauteur).

3.5.3. Méthodes Principales

1. **isHit** (commentée dans le code actuel)
    - Vérifie si une collision entre la plateforme et le joueur a eu lieu.
    - Les conditions évaluent la position du joueur (définie par un rectangle box) par rapport à la position et aux dimensions de la plateforme.
2. **whenHit**
    - Implémentation de la méthode définie par l'interface IJumpable.
    - Action effectuée lorsque le joueur entre en contact avec la plateforme : le joueur effectue un rebond en appelant la méthode rebound() de la classe Player.

3.5.4. Utilité Globale

La classe Platform est une base essentielle pour modéliser les plateformes du jeu. Elle fournit les dimensions par défaut, une gestion des collisions (via la méthode isHit, actuellement inactive), et définit le comportement lorsque le joueur saute dessus (rebond). Cette abstraction permet de créer facilement des variantes spécifiques de plateformes en héritant de cette classe.

3.6. Class Monster

La classe **Monster** représente les ennemis dans le jeu Super Jump, introduisant un défi supplémentaire pour le joueur. Ces monstres réagissent aux interactions avec le joueur en fonction de la situation.

3.6.1. Importations

- android.graphics.BitmapFactory : Permet la manipulation et le chargement des images pour représenter visuellement les monstres dans le jeu.
- android.graphics.Paint : Utilisé pour gérer les propriétés graphiques des objets (non utilisé directement dans ce code, mais potentiellement pour des extensions).
- com.doodlejump.GameObject : Classe parent définissant les propriétés générales des objets du jeu.
- com.doodlejump.Player : Classe représentant le joueur, utilisée pour gérer les interactions avec les monstres.
- com.doodlejump.Vector : Classe pour manipuler les positions et dimensions des objets du jeu.

3.6.2. Variables

- size : Constante définie dans le companion object, de type Vector, représentant les dimensions par défaut d'un monstre (180 px x 180 px).

3.6.3. Constructeurs

1. **Principal**
    - Paramètres :
        - pos0 : Position initiale du monstre sous forme de vecteur.
        - iSize : Dimensions du monstre.
        - type : Identifiant permettant de charger un monstre spécifique.
    - Ce constructeur initialise un objet Monster avec des dimensions et un type personnalisé.
2. **Secondaire**
    - Paramètre :
        - pos0 : Position initiale.
    - Appelle le constructeur principal avec des dimensions par défaut (217F, 144F) et un type associé à une ressource graphique (R.drawable.esme_monster).

3.6.4. Méthodes Principales

1. **whenHit**
    - Définie pour gérer les interactions entre le joueur et le monstre lorsque le joueur entre en collision avec lui.
    - Fonctionnement :
        - Si la vitesse verticale du joueur (player.speed.y) est **négative** (le joueur tombe sur le monstre) :
            - Le monstre est retiré du jeu (removed = true).
            - Le joueur rebondit grâce à la méthode rebound().
        - Sinon (le joueur est frappé par le monstre) :
            - Le joueur meurt en appelant player.die().
            - Sa vitesse verticale est inversée, et il est projeté vers le haut à -50 px/s.

3.6.5. Utilité Globale

La classe Monster ajoute une dynamique essentielle au gameplay. Elle permet de définir des ennemis interactifs avec des comportements adaptés aux actions du joueur

3.7. Class MovingMonster

La classe **MovingMonster** représente un monstre mobile dans le jeu Super Jump, ajoutant une complexité supplémentaire en se déplaçant horizontalement.

3.7.1. Importations

• com.doodlejump.GameManager : Fournit les informations sur l'état du jeu, comme la largeur de l'écran, utilisées pour limiter le déplacement horizontal du monstre.
• com.doodlejump.IUpdate : Interface indiquant que l'objet doit être mis à jour à chaque cycle du jeu.
• com.doodlejump.R : Référence aux ressources graphiques, ici pour associer une image au monstre mobile.
• com.doodlejump.Vector : Classe permettant de manipuler les positions et mouvements sous forme vectorielle.

3.7.2. Variables

• direction : Variable privée de type Byte, initialisée à 1, représentant la direction du déplacement horizontal du monstre.

- **1** : déplacement vers la droite.
- **-1** : déplacement vers la gauche.

3.7.3. Constantes

• MONSTER_SPEED : Constante définissant la vitesse de déplacement horizontale du monstre, fixée à 10 px/cycle.

3.7.4. Constructeur

Le constructeur initialise un monstre mobile avec :

- iPos : La position initiale du monstre sur l'écran.
- Vector(187f, 121f) : Les dimensions du monstre mobile.
- R.drawable.movingmonster : La ressource graphique représentant le monstre.

3.7.5. Méthodes Principales

1. **update**
    - Implémentation de l'interface IUpdate.
    - Méthode appelée à chaque cycle du jeu pour mettre à jour la position du monstre.
    - Fonctionnement :
        - Si le monstre atteint le bord droit de l'écran (pos.x > game.width - size.x), sa direction est inversée (direction = -1).
        - Si le monstre atteint le bord gauche (pos.x < 0), la direction est remise à droite (direction = 1).
        - Le monstre est ensuite déplacé horizontalement en fonction de sa direction et de la constante MONSTER_SPEED grâce à la méthode move().

3.7.6. Utilité Globale

La classe MovingMonster enrichit l'expérience du jeu en introduisant des ennemis mobiles. Elle offre une dynamique supplémentaire en forçant le joueur à anticiper les déplacements du monstre. Grâce à son comportement géré dans la méthode update, ce type de monstre peut interagir de manière plus complexe avec le joueur, ajoutant une couche stratégique au gameplay. La gestion flexible des limites d'écran garantit que le monstre reste dans les bornes du jeu, tout en maintenant son mouvement fluide.

3.5. Class Player

La classe Player représente le Doodler dans le jeu.

3.5.1. Importations

- **android.graphics.*** : Permet la manipulation d'images et de graphismes, notamment :
    - Bitmap : Gère les images associées au doodler.
    - Matrix : Utilisée pour retourner l'image du joueur selon la direction de déplacement.

3.5.2. Variables

- **accélération et speed :**
    - acceleration : Définit la gravité agissant sur le doodler.
    - speed : Représente la vitesse actuelle du doodler.
- **alive et hitable :**
    - alive : Indique si le joueur est encore en vie.
    - hitable : Indique si le joueur peut subir des collisions (désactivé avec un jetpack et le trampoline).
- **jumpBox :** Zone utilisée pour vérifier les collisions pendant un saut.

- **Constantes :**
    - JUMP_SPEED, GRAVITY, et JUMP_HEIGHT contrôlent les paramètres physiques du saut.

3.5.3. Méthodes Principales

1. **update(game: GameManager)**
    - Met à jour la position et la vitesse du doodler en fonction de la gravité et des déplacements.
    - Gère les rebonds, les limites de l'écran, et le défilement des objets à mesure que le doodler monte.
2. **draw(game: GameManager)**
    - Dessine le doodler à l'écran, en retournant son image selon sa direction de déplacement.
3. **rebound()**
    - Permet au doodler de rebondir lorsqu'il touche une plateforme.
4. **die()**
    - Arrête les déplacements et marque le doodler comme mort.
5. **checkCollisions(objects)**
    - Vérifie si le doodler entre en collision avec d'autres objets.
6. **changeJetpack(enabled, game)**
    - Active ou désactive le mode jetpack, modifiant la taille et l'image du joueur.

3.5.4. Utilité Globale

La class Player gère tous les aspects du comportement et de l'apparence du Doodler. Elle intègre les interactions avec les objets du jeu, les rebonds sur les plateformes, et les effets des bonus, comme le jetpack.

3.6. Class ReplayActivity

La classe ReplayActivity gère l'écran de fin de partie, où le score final est affiché et où le joueur peut choisir de rejouer en cliquant sur le bouton « Rejouer ».

3.6.1. Importations

- **android.content.Intent** : Permet de naviguer entre les class, ici utilisé pour relancer MainActivity.
- **android.graphics.RectF** : Délimite la zone cliquable pour interagir avec les boutons du jeu.
- **android.view.** : Fournit les outils pour gérer les événements tactiles, comme MotionEvent et OnTouchListener.
- **android.widget.** : Inclut les widgets comme TextView (affichage du score) et ImageView (arrière-plan ou bouton rejouer).

3.6.2. Variables

- **score :** Récupère le score final depuis l'écran précédent via un Intent.
- **scoreView :** Un TextView qui affiche le score à l'écran.
- **replayView :** Un ImageView qui représente l'arrière-plan et un bouton pour redémarrer la partie.
- **playButton :** Une zone rectangulaire qui définit la zone tactile pour relancer une nouvelle partie.

3.6.3. Méthodes Principales

1. **onCreate**
    - Configure l'interface utilisateur de l'écran de replay à partir du fichier XML associé (activity_replay).
    - Initialise les variables :
        - score : Récupère la valeur du score final transmis par MainActivity.
        - scoreView : Définit l'affichage du score avec une taille et un formatage prédéfinis.
        - replayView : Active suivi des interactions tactiles avec setOnTouchListener.
2. **onTouch**
    - Vérifie si un clic sur l'écran correspond à la zone définie par playButton.
    - Si un clic est détecté, relance MainActivity pour redémarrer une nouvelle partie.

3.6.4. Utilité Globale

La classe ReplayActivity offre une interface simple pour conclure une partie en affichant le score final. Elle permet au joueur de relancer une nouvelle partie rapidement.

3.7. Class TimeObservable

La classe TimeObservable permet de gérer les événements limités dans le temps liés à un objet spécifique. Voici une analyse détaillée de son contenu.

3.7.1. Variables

- **duration :** Temps restant avant que l'objet ne soit marqué comme supprimé.
- **obj :** L'objet (GameObject) associé à cette logique temporelle.
- **started :** Indique si le compteur de temps a démarré.
- **maxDuration :** La durée initiale maximale pour cet objet.

3.7.2. Méthodes Principales

1. **start()**
    - Active le chronomètre en définissant started à true.
2. **update()**
    - Réduit la durée restante si le chronomètre est activé.
    - Marque l'objet associé comme supprimé (removed = true) lorsque la durée atteint zéro.
    - Enregistre des informations dans les journaux système (Log.d) pour déboguer.

3.7.3. Utilité Globale

La class TimeObservable est utilisée pour gérer des objets temporaires dans le jeu, comme des bonus ou des plateformes spéciales qui disparaissent après un certain temps. Elle garantit que ces éléments fonctionnent de manière autonome tout en restant synchronisés avec la logique principale du jeu.

3.8. Class Vector

La classe Vector représente des vecteurs mathématiques à deux dimensions, utilisés pour gérer les positions, mouvements et autres calculs vectoriels dans le jeu.

3.8.1. Méthodes et Opérations

1. Opérations arithmétiques
    - plus (+) : Additionne deux vecteurs et retourne le résultat.
    - minus (-): Soustrait un vecteur d'un autre.
    - times (*): Multiplie les composantes d'un vecteur par un scalaire.
    - div (/) : Divise les composantes d'un vecteur par un scalaire.
2. Accès aux composantes
    - get ([i]) : Retourne la composante x (si i == 0) ou y (si i == 1).
    - set ([i] = value) : Modifie la composante x ou y selon l'indice spécifié.
3. Méthodes utilitaires
    - toString() : Retourne une représentation textuelle du vecteur sous la forme (x, y).

3.8.2. Utilité Globale

La classe Vector est d’une grande importance pour gérer les calculs liés aux positions et aux mouvements dans le jeu. Elle simplifie les opérations mathématiques complexes en encapsulant la logique des vecteurs, rendant le code plus lisible et modulaire.

4. Utilité du Dossier drawable

Dans une application Android, le dossier drawable est utilisé pour stocker toutes les ressources graphiques nécessaires au fonctionnement de l'application. Cela inclut :

- Images et icônes : Les design représentant les personnages, plateformes, bonus ou arrière-plans dans le jeu Super Jump.
- Fichiers XML : Pour définir des formes, des dégradés ou des sélecteurs graphiques.

Dans notre jeu Super Jump, le dossier drawable joue un rôle central dans l'affichage : il contient les ressources visuelles appelées directement dans le code pour représenter des objets tels que le Doodler dans (Player), les plateformes dans (Platform), ou les bonus dans (Jetpack) ect. Ce dossier permet de centraliser toutes les ressources graphiques, facilitant leur gestion et leur réutilisation tout en maintenant une structure claire pour l'application.

5. Utilité des Fichiers layout

Dans une application Android, le dossier layout contient les fichiers XML qui définissent le visuelle et les composants des interfaces utilisateur. Ces fichiers décrivent la disposition des éléments graphiques à l'écran, comme les boutons, les textes, ou les images.

Dans notre jeu Super Jump, les fichiers layout jouent un rôle important pour structurer chaque écran :

- activity_menu.xml : Définit la structure de l'écran de menu principal, comprenant le bouton "Jouer".
- activity_main.xml : Structure l'interface principale où le jeu se déroule. Bien que la majeure partie du rendu graphique soit gérée par le GameManager, ce fichier inclue des éléments fixes comme un score.
- activity_replay.xml : Configure l'écran de fin de partie, affichant le score final et le bouton pour rejouer.

Les fichiers layout permettent de séparer la logique (code Java/Kotlin) de la présentation (interface utilisateur), facilitant ainsi la maintenance, la lisibilité, et la modification des interfaces de l'application. Ils offrent également la possibilité d'adapter facilement l'interface aux différentes tailles et orientations d'écran.

6. Conclusion et Bilan

La création du jeu Super Jump a été une expérience enrichissante et captivante. Développer pour la première fois un jeu vidéo nous a permis de combiner logique de programmation, gestion des ressources graphiques et conception d'interfaces interactives.

Ce projet nous a également donné l'opportunité de découvrir des concepts fondamentaux comme :

- La manipulation de graphiques et d'animations dans une application mobile.
- L'intégration de capteurs matériels, tels que l'orientation du téléphone pour contrôler les déplacements.
- La gestion des cycles de vie des activités dans une application Android.

Enfin, ce projet nous a permis de renforcer nos compétences en Kotlin et en développement Android, tout en travaillant sur un projet pratique et concret.

Nous souhaitons également remercier notre professeur pour son suivi constant, ses conseils avisés, et son aide précieuse tout au long de ce projet.