Rapport de Projet - Développement d’un Moteur de Jeu 2D avec LibGDX

RunGame

Équipe et Contributions

- **Di Placido Bence** [développeur du moteur physique, développeur du gameplay, développeur des interfaces menu, développeur sur la gestion sonore du jeu, développeur des cartes Tiled]
- Lien github : https://github.com/Bence-dp/RunGame

Section 1. Introduction

Le projet **RunGame** est un jeu de plateforme 2D développé dans le cadre d’une exploration des capacités de la bibliothèque **libGDX** et de l’éditeur de cartes **Tiled**. L'objectif est de fournir un moteur de jeu simple mais extensible, permettre d’ajouter du contenu (monstres, platformes, collectibles) via Tiled .

Section 2. Présentation du projet

RunGame est un platformer 2D. Le joueur sera confronté a une chaîne de niveaux différents (extensible) avec des monstres a éviter.

Technologies et Outils Utilisés

- **LibGDX** : pour le développement du moteur de jeu.
- **Tiled** : pour la création et la configuration des cartes.
- LibGDX Box2D : pour la création d’une physique.

Fonctionnalités implémentées.

<u>Création de niveaux sur tiled :</u>

-Création automatisé des monstres,pièces, spawn du joueur et fin du niveau a l’aide des objets Tiled.

-Création automatisé des plateformes solides.

-Transitions entre les niveaux :

-Lorsqu’un joueur atteint un point spécifique, le jeu charge automatiquement le niveau suivant.

<u>Mouvement du joueur :</u>

-Contrôles de base : marche, course et saut.

- Gestion de la gravité pour un effet de chute réaliste.

-Les touches pour contrôler le joueur peuvent être redéfinies.

Entités :

-Les monstres se déplacent dans une zone définie sur Tiled.

-Faciliter a implémenter des nouveaux monstres

-Faciliter a implémenter des nouveaux déplacements pour les monstres

-Les pièces sont placer sur Tiled.

-Gestion des contacts entre les entités (player – monstres ; player – pièces)

Son :

-Effets sonores pour les différents événements (collecte un pièce,joueur éliminer, saut, niveaux terminé)

Fonctionnalités supplémentaires :

-Un fichier JSON est lu pour crée une chaîne de niveaux a compléter pour finir le jeu

cette chaîne de niveaux peux être donc modifier (ajouter des niveaux, supprimer des niveaux ,modifier des niveaux)

-Sauvegarde avec la sérialisation (dernier niveau,score)

Configuration et Ajout de Contenu avec Tiled

Création de niveaux valides via Tiled :

Chaques map tiled doit contenir **ABSOLUMENT** :

- un layer de tuiles nommé obstacles avec les plateformes

- un layer d'objets entity avec des rectangles (pour les entités voir plus bas pour leurs configurations)

-un layer d'objets teleporter pour placer un rectangle pour la fin du niveau.

Des layers peuvent etres ajouté pour la décoration.

Ajout d'entités :

Dans le dossier assets/maps/entityExemples , vous trouverez les configuration pour les différents objets implémenter. Pour les ajouter il vous suffira des les glisser sur votre map tiled sur le layer prescrit en haut, clique droit -> détacher (pour que les propriétés personnalisés soient actives).

Pour les ennemies, la largeur et la hauteur des objets définissent la zone dans laquelle les monstres se déplacent. Leur vitesse pourra être modifié également dans les propriétés des objets. Leurs textures sont définies par leur nom.

Les pièces n'ont besoin d'aucune modifications

Pour les sorties de niveaux, la zone de contact est également définie par la largeur et hauteur de l'objet Tiled.

Compilation et exécution

Présentez comment **compiler et executer votre projet**.

1. **Prérequis** :
   Java JDK version 11 ou supérieure installé.
   Gradle installé (ou utilisez le wrapper Gradle inclus).
   IDE recommandé (IntelliJ IDEA, Eclipse) ou terminal de commande.

2. **Lancement du Jeu** :

   Accédez au répertoire du projet où le fichier `gradlew` (ou `gradlew.bat` sur Windows) est situé.

   Ouvrir un terminal ou une ligne de commande dans ce répertoire.

   Exécuter le jeu avec la commande suivante :

   Sur **Windows** : gradlew.bat run

   Sur **macOS/Linux** : ./gradlew run

Lien github : https://github.com/Bence-dp/RunGame

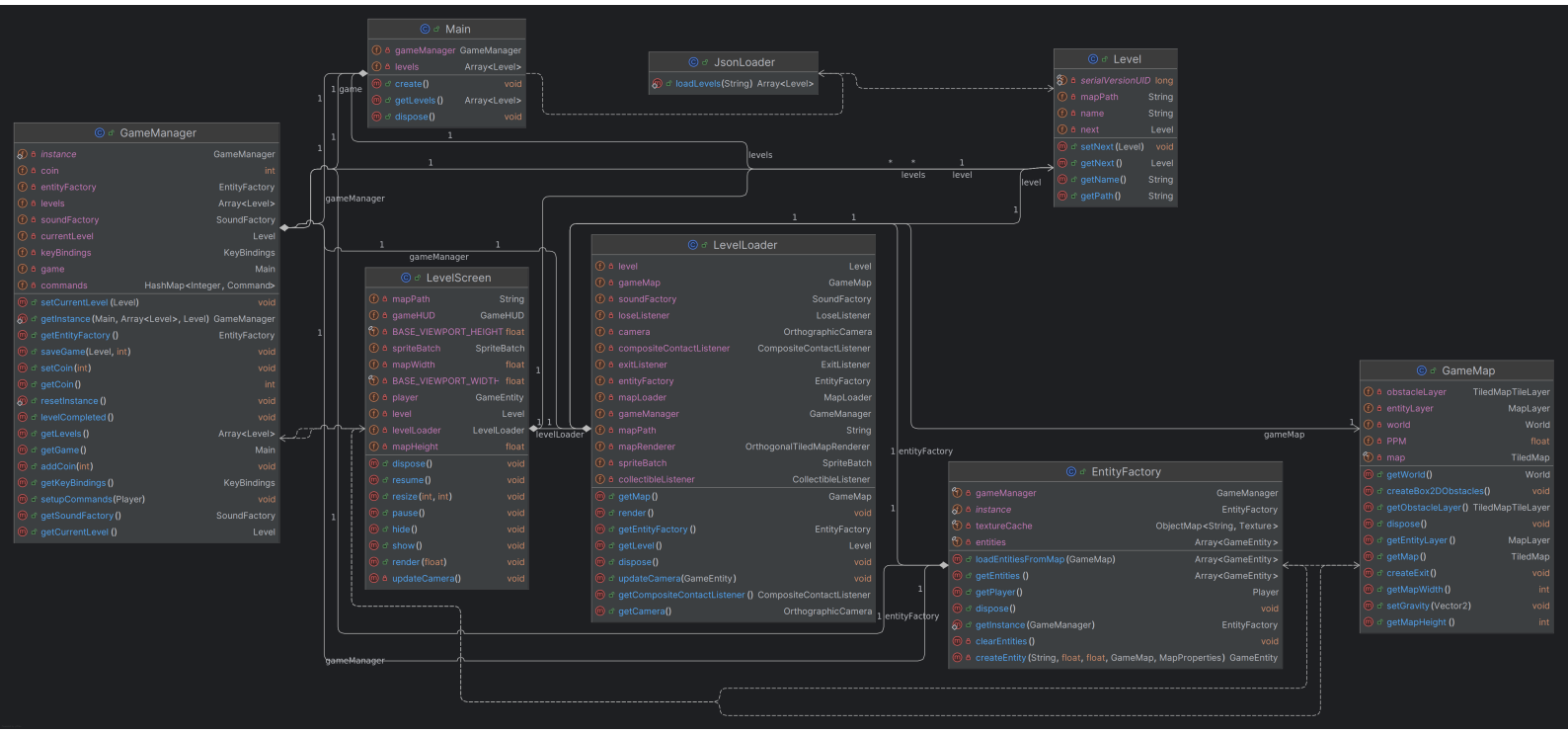
Section 3. Présentation technique du projet et contributions

Vous devrez avoir une section qui présente le code et les contributions de chacun.

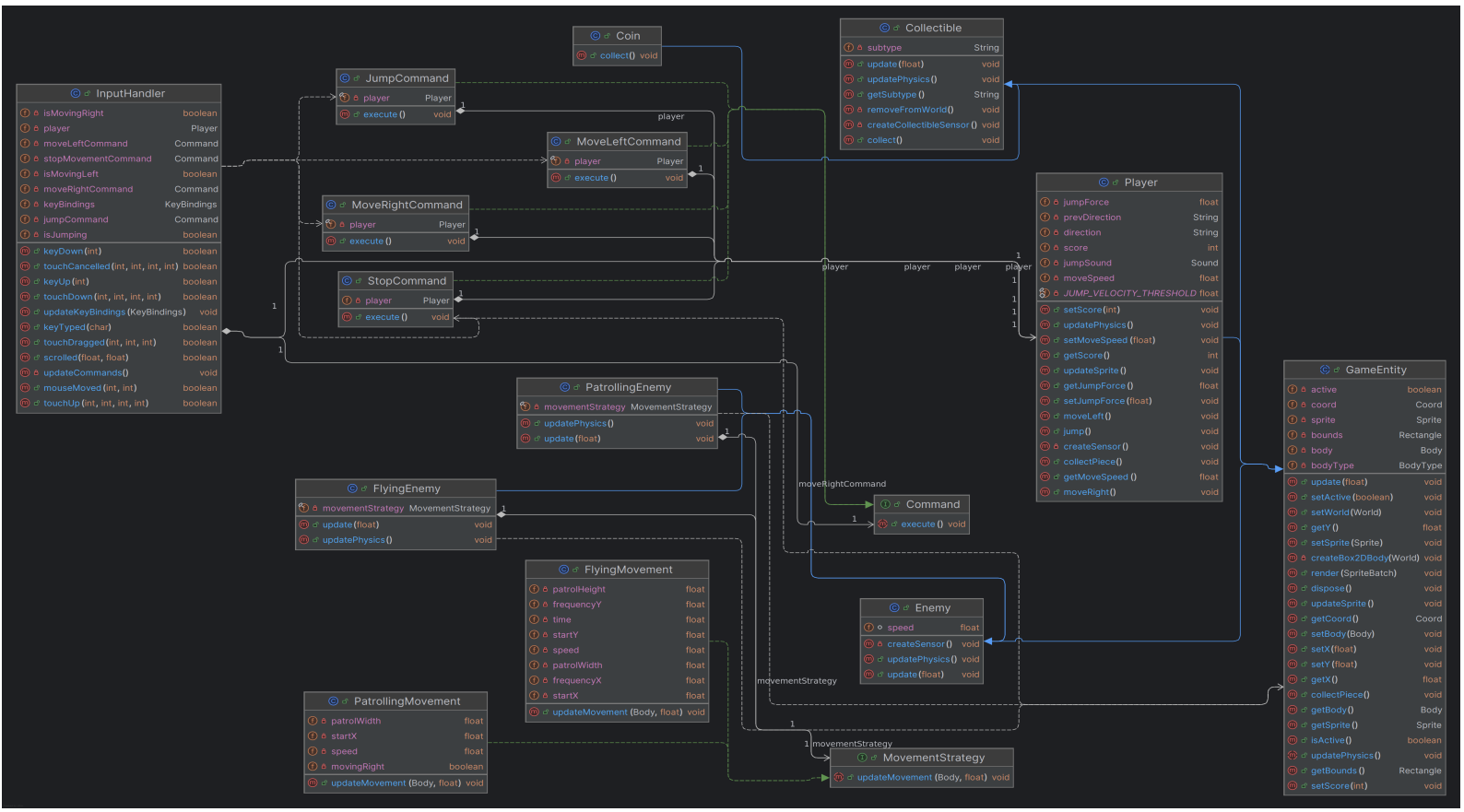
Architecture Générale du moteur de jeu

Au lancement du jeu on créé le singleton GameManager et récupere les niveaux (tiled) du json pour faire une chaîne de niveaux.

GameManager va créé les singletons utiles pour le jeu (EntityFactory,SoundFactory), on précharges alors les textures des entités et les sons. GameManager récupère la chaîne des niveaux et s'occupe de mettre a jour le niveau actuel. Une fois sur l'eccran de jeu (LevelScreen) la classe LevelLoader va charger le bon niveau fourni par GameManager. LevelLoader appelle GameMap qui gère les cartes Tiled . EntityFactory va alors créé les objet présent dans la GameMap.



La classe centrale, **GameEntity**, sert de base pour toutes les entités du jeu, telles que le joueur (**Player**), les ennemis (**Enemy**) et les objets collectables (**Collectible**). Le joueur interagit avec le système via un gestionnaire d'entrées (**InputHandler**) qui mappe les actions utilisateur sur des commandes concrètes (comme **MoveLeftCommand** ou **JumpCommand**) grâce au **Pattern Command**. Les ennemis, quant à eux, adoptent des comportements variés définis par des stratégies de mouvement (**MovementStrategy**), comme la patrouille ou le vol. Les objets collectables, tels que les pièces (**Coin**), interagissent avec le joueur pour modifier son score. Cette architecture modulaire permet une grande extensibilité, où de nouvelles entités, stratégies ou commandes peuvent être facilement ajoutées au système.



Utiliser et étendre la librairie du moteur de jeu

La librairie offre une grande flexibilité pour étendre et personnaliser divers aspects du jeu, notamment les ennemis, les stratégies de mouvement, les commandes du personnage, les objets collectables, et les cartes.

Ennemis et stratégies de mouvement

Les ennemis sont basés sur la classe **Enemy**, qui encapsule les comportements généraux tels que l'interaction avec le joueur ou les obstacles. Chaque ennemi peut être associé à une **MovementStrategy** (stratégie de mouvement) pour déterminer son comportement. Par exemple, un ennemi volant peut utiliser la classe **FlyingMovement**, tandis qu'un ennemi patrouillant peut s'appuyer sur **PatrollingMovement**. Pour étendre le système, il suffit de créer une nouvelle classe implémentant **MovementStrategy**, comme un comportement aléatoire ou de poursuite, puis de l'associer à un ennemi existant ou personnalisé.

Collectibles

Les objets à collecter, tels que les pièces, sont gérés par la classe **Collectible**, qui définit les bases pour tout objet interactif. Chaque objet peut surcharger les méthodes comme **collect()** ou **updatePhysics()** pour définir des comportements spécifiques lorsqu'ils sont ramassés par le joueur. Ajouter un nouvel objet collectable est aussi simple que d'étendre **Collectible** et de définir ses propriétés uniques, comme son apparence ou ses effets (par exemple, ajouter des points, donner un pouvoir temporaire, etc.).

Commandes pour contrôler le personnage

Le contrôle du personnage repose sur le **Command Pattern**, ce qui rend les commandes très modulaires. Les actions comme se déplacer à gauche, à droite ou sauter sont gérées par des classes spécifiques implémentant l'interface **Command**, telles que **MoveLeftCommand**, **MoveRightCommand** ou **JumpCommand**. Pour ajouter de nouvelles commandes, par exemple une attaque ou un dash, il suffit de créer une nouvelle classe implémentant **Command**, puis de l'associer à une touche ou une combinaison de touches via l'**InputHandler**.

Cartes

Les cartes sont représentées par la classe **Level**, qui sert de conteneur pour les éléments du jeu comme les ennemis, les collectibles, et les obstacles. Chaque carte peut être configurée pour inclure différents objets ou événements en fonction du gameplay. Les niveaux sont chaînés entre eux grâce à l'attribut **next**, permettant une progression fluide. Pour créer une nouvelle carte, il suffit d'instancier un objet **Level**, d'y ajouter les éléments souhaités, puis de l'intégrer au système via le **GameManager**.

Intégration et extensibilité

Grâce à cette architecture modulaire, les développeurs peuvent facilement ajouter de nouveaux ennemis avec des comportements uniques, enrichir les cartes avec des éléments interactifs, et proposer des commandes originales pour le joueur. La flexibilité offerte par les stratégies de mouvement, les commandes et les objets rend la librairie idéale pour construire rapidement des jeux personnalisés.

Répartition des Tâches

Pas de répartitions, travail seul.

Section 4. Conclusion et Perspectives

Bilan du projet

Ce projet de moteur de jeu a atteint de nombreux objectifs clés, notamment la création d’une architecture modulaire, extensible et facile à utiliser. Les fonctionnalités principales, comme le contrôle du personnage via un système de commandes, la gestion des ennemis avec des stratégies de mouvement, l'intégration de collectibles interactifs et la structuration des niveaux en cartes progressives, sont toutes opérationnelles. L’utilisation de patrons de conception comme le **Command Pattern** et le **Strategy Pattern** a renforcé la flexibilité et la réutilisabilité du code, rendant le moteur adaptable à divers types de jeux.

Objectifs atteints :

- **Modularité et extensibilité :** Les développeurs peuvent facilement ajouter de nouveaux ennemis, objets ou commandes sans modifier le code existant.
- **Gestion des niveaux :** Un système de cartes structuré avec une progression fluide entre les niveaux.
- **Interaction joueur-éléments :** Une intégration intuitive des collectibles et des interactions avec les ennemis.
- **Contrôle joueur optimisé :** Un système de commandes basé sur les actions du joueur, offrant une prise en main simplifiée.
- **Réutilisabilité :** Une base de code claire et bien organisée, facilitant les modifications et les extensions.

Défis rencontrés :

- **Gestion de la complexité :** L’intégration de plusieurs systèmes (ennemis, commandes, collectibles, cartes) a nécessité une coordination minutieuse pour éviter les dépendances circulaires.
- **Optimisation des performances :** La gestion des mouvements et des interactions physiques a nécessité des ajustements pour maintenir une fluidité optimale, en particulier avec plusieurs ennemis et objets actifs.
- **Extensibilité initiale :** Trouver un équilibre entre simplicité et flexibilité lors de la conception des bases a demandé une réflexion approfondie.

Pistes d'amélioration et nouvelles fonctionnalités

1. **IA avancée pour les ennemis :** Introduire une intelligence artificielle plus élaborée, permettant aux ennemis de réagir dynamiquement au comportement du joueur, comme éviter les obstacles, former des embuscades, ou changer de stratégie en cours de jeu.
2. **Créateur de niveaux visuel (intégré):** Développer un éditeur de niveaux graphique pour permettre aux concepteurs de glisser-déposer des éléments (ennemis, collectibles, obstacles) sur une carte et générer automatiquement les fichiers de configuration correspondants.

3. **Système de physique amélioré :** Étendre la gestion des collisions et des interactions physiques pour inclure des comportements avancés, comme des projectiles, des zones d’effet ou des interactions complexes entre objets.
4. **Amélioration des collectibles :** Ajouter des collectibles ayant des effets variés sur le joueur, comme des bonus de vitesse, des augmentations de saut, une invincibilité temporaire, ou encore des malus qui compliquent temporairement le gameplay. Ces effets pourraient être gérés via un système de "power-ups" centralisé pour garantir leur extensibilité.
5. **Support multijoueur :** Classements des joueur suivant leurs scores/temps.
6. **Personnalisation du joueur :** Introduire un système de personnalisation du personnage, permettant au joueur de choisir des apparences, des compétences ou des équipements avant de commencer une partie.
7. **Système de quêtes et d’objectifs :** Ajouter des quêtes ou des objectifs secondaires dans les niveaux pour enrichir l'expérience du joueur et diversifier le gameplay.

Bilan du projet

Ce projet de moteur de jeu a atteint de nombreux objectifs clés, notamment la création d’une architecture modulaire, extensible et facile à utiliser. Les fonctionnalités principales, comme le contrôle du personnage via un système de commandes, la gestion des ennemis avec des stratégies de mouvement, l'intégration de collectibles interactifs et la structuration des niveaux en cartes progressives, sont toutes opérationnelles. L’utilisation de patrons de conception comme le **Command Pattern** et le **Strategy Pattern** a renforcé la flexibilité et la réutilisabilité du code, rendant le moteur adaptable à divers types de jeux.

Objectifs atteints :

- **Modularité et extensibilité :** Les développeurs peuvent facilement ajouter de nouveaux ennemis, objets ou commandes sans modifier le code existant.
- **Gestion des niveaux :** Un système de cartes structuré avec une progression fluide entre les niveaux.
- **Interaction joueur-éléments :** Une intégration intuitive des collectibles et des interactions avec les ennemis.
- **Contrôle joueur optimisé :** Un système de commandes basé sur les actions du joueur, offrant une prise en main simplifiée.
- **Réutilisabilité :** Une base de code claire et bien organisée, facilitant les modifications et les extensions.

Défis rencontrés :

- **Gestion de la complexité :** L’intégration de plusieurs systèmes (ennemis, commandes, collectibles, cartes) a nécessité une coordination minutieuse pour éviter les dépendances circulaires.
- **Optimisation des performances :** La gestion des mouvements et des interactions physiques a nécessité des ajustements pour maintenir une fluidité optimale, en particulier avec plusieurs ennemis et objets actifs.

- **Extensibilité initiale :** Trouver un équilibre entre simplicité et flexibilité lors de la conception des bases a demandé une réflexion approfondie.

Pistes d'amélioration et nouvelles fonctionnalités

1. **IA avancée pour les ennemis :** Introduire une intelligence artificielle plus élaborée, permettant aux ennemis de réagir dynamiquement au comportement du joueur, comme éviter les obstacles, former des embuscades, ou changer de stratégie en cours de jeu.
2. **Créateur de niveaux visuel :** Développer un éditeur de niveaux graphique pour permettre aux concepteurs de glisser-déposer des éléments (ennemis, collectibles, obstacles) sur une carte et générer automatiquement les fichiers de configuration correspondants.
3. **Système de physique amélioré :** Étendre la gestion des collisions et des interactions physiques pour inclure des comportements avancés, comme des projectiles, des zones d’effet ou des interactions complexes entre objets.
4. **Support multijoueur :** Ajouter un mode multijoueur coopératif ou compétitif avec des mécaniques adaptées, comme des objectifs partagés, des scores individuels ou des interactions directes entre joueurs.
5. **Personnalisation du joueur :** Introduire un système de personnalisation du personnage, permettant au joueur de choisir des apparences, des compétences ou des équipements avant de commencer une partie.
6. **Système de quêtes et d’objectifs :** Ajouter des quêtes ou des objectifs secondaires dans les niveaux pour enrichir l'expérience du joueur et diversifier le gameplay.
7. **Système de sous niveaux :** Ajouter des passages de maps en maps dans le meme niveau
8. **Optimisation des performances :** Mettre en œuvre des techniques de **culling** (gestion intelligente des objets visibles) ou de mise en cache pour améliorer les performances sur des appareils moins puissants.
9. **Compatibilité multiplateforme :** Étendre le support du moteur pour faciliter le déploiement sur des plateformes variées, telles que les consoles, les navigateurs ou les appareils mobiles.

En conclusion, ce projet de moteur de jeu a posé des bases solides pour la création de jeux personnalisés tout en laissant un potentiel important d’évolution. Les améliorations futures pourraient se concentrer sur l'ajout de fonctionnalités immersives et l'optimisation de l'expérience utilisateur.

Annexes

https://libgdx.com/wiki/

https://www.youtube.com/watch?v=85A1w1iD2oA&list=PL-2t7SM0vDfdYJ5Pq9vxeivblbZuFvGJK

Bonus :

Model MVC

Design patterns utiliser :

Factory (entités,sons)

Command (commandes du joueur)

Strategy (mouvement des enemies)

Singletons (les factory,GameManager)