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

Moteur de Jeu de Plateforme 2D - Style Super Mario Bros

Équipe et Contributions

Andrea Kocovic : Développeur Vue & Contrôleur

Responsable du rendu graphique, de l'interface utilisateur, de la gestion des états du jeu et de l'intégration avec Tiled.

Moussa CISSE : Développeur Modèle & Logique

Responsable de l'architecture des entités, de la logique métier, du système de collisions et des comportements ennemis.

Section 1- Introduction

Ce projet s'inscrit dans le cadre du cours de développement Java en Licence 3 MIAGE. Il vise à créer un moteur de jeu de plateforme 2D inspiré de Super Mario Bros, permettant d'approfondir les concepts de programmation orientée objet, d'architecture logicielle et d'intégration d'outils externes.

Le choix d'un jeu de plateforme permet d'explorer des problématiques techniques variées comme la physique du mouvement, la gestion des collisions, l'intelligence artificielle basique, le rendu graphique multi-couches et l'architecture extensible. L'utilisation du framework LibGDX offre une base solide pour le développement multiplateforme.

Objectifs du Projet

Développer un moteur de jeu 2D extensible : Le moteur permet d'ajouter du contenu (nouveaux niveaux, ennemis, power-ups) sans modifier le code Java existant grâce à l'utilisation de l'éditeur Tiled.

Implémenter une architecture MVC propre : Séparation claire entre la logique métier (Modèle), l'affichage (Vue) et la coordination (Contrôleur) pour faciliter la maintenance.

Créer un système de niveaux configurable via Tiled : Les concepteurs peuvent créer et modifier des cartes sans connaissances en programmation.

Intégrer des mécaniques de gameplay riches : Physique réaliste, système de collisions robuste, transformations de personnage, power-ups variés et séquence de fin élaborée.

Appliquer les principes SOLID et les design patterns : Garantir un code de qualité professionnelle, maintenable et évolutif.

Section 2. Présentation du Projet

Technologies et Outils Utilisés

LibGDX version 1.9.10 est le framework Java principal utilisé pour le développement. Il offre des APIs complètes pour le rendu graphique, la gestion des entrées et permet le déploiement multiplateforme.

****Java 17**** est le langage de programmation choisi pour sa robustesse en programmation orientée objet et sa portabilité.

****LWJGL 3.3.1**** sert de backend OpenGL pour le rendu graphique sur desktop, fournissant les liaisons Java vers les bibliothèques natives.

****Tiled Map Editor**** permet de créer des niveaux visuellement en générant des fichiers TMX que le moteur peut charger et interpréter.

****IntelliJ IDEA**** est l'environnement de développement utilisé pour l'édition, le débogage et la gestion du projet.

Fonctionnalités Implémentées

Système du Joueur

Le personnage principal dispose de déplacements fluides avec une physique réaliste incluant gravité, inertie et friction. Le système de transformation permet d'évoluer à travers trois états : PETIT, GRAND. Chaque transformation modifie la taille du personnage et ses capacités. Le joueur dispose de trois vies, avec possibilité d'en gagner supplémentaires. Une période d'invincibilité de deux secondes suit chaque dégât pour éviter les éliminations instantanées.

Ennemis

Les ennemis terrestres Goomba patrouillent au sol avec un comportement de va-et-vient, détectant automatiquement les bords et obstacles. L'architecture utilise le pattern Strategy pour permettre l'assignation dynamique de comportements différents.

Les ennemis peuvent être éliminés en sautant sur leur tête, tandis qu'un contact latéral inflige des dégâts au joueur.

Power-Ups

Quatre types de power-ups sont disponibles : le Champignon Super pour grandir, la Fleur de Feu pour obtenir des capacités spéciales, le Champignon 1UP pour gagner une vie, et l'Étoile pour dix secondes d'invincibilité totale. Tous apparaissent depuis des blocs question et se déplacent automatiquement.

Séquence de Fin de Niveau

La fin de niveau comporte une séquence automatique en quatre étapes : glissade le long du mât du drapeau, marche automatique vers le château, disparition progressive avec effet de fade out, puis affichage du menu avec les options Rejouer, Niveau Suivant ou Quitter.

Système de Collisions

Le moteur implémente un système de détection de collisions robuste basé sur l'algorithme AABB. Les collisions sont gérées séparément pour chaque type d'interaction : joueur contre obstacles, ennemis, power-ups et drapeau. Les ennemis et power-ups détectent également les obstacles pour adapter leur mouvement.

Interface et Rendu

L'interface affiche en temps réel le score, les vies, les pièces collectées et le monde actuel. Le système de rendu utilise sept couches avec gestion du Z-index pour un affichage correct de la profondeur. Les effets visuels incluent l'alpha blending, le clignotement et des animations fluides à 60 FPS. La caméra suit le joueur avec une interpolation douce.

Configuration et Ajout de Contenu avec Tiled

Structure d'un Niveau

Chaque niveau est un fichier TMX contenant plusieurs couches d'objets. La couche Joueur définit la position de départ. La couche Ennemis contient les ennemis avec leurs propriétés de type et comportement. La couche PowerUps place les différents power-ups avec leur type spécifique. La couche Obstacles définit les blocs et plateformes. La couche Drapeau marque la fin du niveau. La couche Collision définit les zones solides de la carte.

Processus de Création d'un Niveau

Pour créer un nouveau niveau, le concepteur ouvre Tiled et crée une nouvelle carte avec des tuiles de 16×16 pixels. Il importe le tileset fourni contenant tous les éléments visuels nécessaires. Il crée ensuite les différentes couches obligatoires et place les éléments de gameplay en définissant leurs propriétés. Le fichier TMX est sauvegardé dans le dossier assets/cartes du projet. Enfin, le chemin du niveau est ajouté au fichier de configuration JSON. Aucune modification du code Java n'est nécessaire.

Interprétation par le Moteur

Au démarrage, la classe ChargeurNiveau parse automatiquement le fichier TMX et crée dynamiquement toutes les entités Java correspondantes. Le pattern Factory est utilisé pour instancier les bonnes classes selon les types spécifiés dans Tiled. Les propriétés personnalisées définies dans l'éditeur sont lues et appliquées aux objets Java.

Compilation et Exécution

Prérequis

Le projet nécessite Java JDK 17 ou supérieur. Toutes les dépendances LibGDX sont incluses dans le dossier lib du projet. Le système fonctionne sur macOS, Linux et Windows.

Compilation

La compilation peut se faire automatiquement via le script lancer.sh qui compile et exécute le jeu en une commande. Pour un contrôle manuel, les scripts séparés **./compiler_javac.sh et ./executer_java.sh et ./lancer.sh** sont disponibles dans le dossier scripts. Les fichiers compilés sont générés dans le dossier out.

Lancement et Contrôles

Le jeu se lance dans une fenêtre de 800×600 pixels. Les flèches gauche et droite permettent de déplacer le personnage. La barre d'espace fait sauter, avec hauteur variable selon la durée de pression. La touche Échap met en pause ou retourne au menu. Les paramètres de configuration peuvent être modifiés dans le fichier configuration.json.

Dépôt GitHub

Le code source complet est disponible sur GitHub à l'adresse à compléter par l'équipe.

Section 3. Présentation Technique du Projet et Contributions

Architecture Générale du Moteur

Le moteur adopte une architecture MVC stricte garantissant une séparation claire des responsabilités.

Le Modèle

Le Modèle contient toute la logique métier du jeu. Le package modele.entites définit la hiérarchie des entités avec une classe abstraite Entite dont héritent Joueur, Ennemi, PowerUp, ObjetCollectable, Obstacle et Drapeau. Le package modele.comportements implémente le pattern Strategy pour les comportements des ennemis via l'interface ComportementEnnemi. Le package modele.gestionnaires contient GestionnaireCollisions pour les collisions et GestionnaireNiveaux pour la progression. Le package modele.niveau représente un niveau avec ses entités, objectifs et progression.

La Vue

La Vue est responsable uniquement de l'affichage graphique sans jamais modifier le Modèle. Le package vue contient l'interface RenduEntite définissant le contrat pour tous les renderers. Les classes RenduJoueur, RenduEnnemi, RenduPowerUp, RenduObjet et RenduDrapeau implémentent le rendu spécifique de chaque entité. RenduNiveau orchestre le rendu complet avec gestion du Z-index. RenduHUD affiche l'interface utilisateur fixe. MenuFinNiveau et RenduTransition gèrent les menus et effets.

Le Contrôleur

Le Contrôleur coordonne Modèle et Vue. ControleurJeu implémente une machine à états gérant sept états du jeu : menu principal, gameplay actif, pause, séquence de fin, menu de fin, game over et victoire. Il orchestre la boucle de jeu à 60 FPS. ControleurEntrees capture les entrées utilisateur et les traduit en actions.

Interactions entre Composants

À chaque frame, le Contrôleur capte les entrées, met à jour le Modèle, déclenche la détection des collisions, puis la Vue lit l'état du Modèle et dessine tous les éléments. Cette séparation stricte permet de modifier l'affichage sans toucher à la logique.

Utiliser et Étendre le Moteur

Ajouter un Nouveau Type d'Ennemi

Pour créer un ennemi avec un comportement différent, il faut créer une nouvelle classe héritant de Ennemi dans le package modele.entites, implémenter la méthode de mise à jour avec le comportement souhaité, modifier ChargeurNiveau pour reconnaître le nouveau type, ajouter les textures dans le dossier assets, puis utiliser le nouveau type dans Tiled en définissant la propriété correspondante.

Ajouter un Nouveau Power-Up

L'ajout d'un power-up nécessite l'extension de l'énumération TypePowerUp dans la classe PowerUp, l'implémentation de l'effet dans la méthode dédiée, l'ajout de la texture, puis l'utilisation dans Tiled.

Ajouter un Nouvel Obstacle

Pour créer un obstacle avec des propriétés différentes, il suffit d'étendre l'énumération TypeObstacle, d'ajouter la logique spécifique dans GestionnaireCollisions si nécessaire, puis de l'utiliser dans Tiled.

Ajouter un Nouveau Niveau

L'ajout de niveaux est le cas le plus simple ne nécessitant aucune programmation. Il suffit de créer le fichier TMX dans Tiled avec les couches requises, le sauvegarder dans assets/cartes, et ajouter son chemin dans le fichier de configuration JSON.

Design Patterns Utilisés

Le **pattern MVC** sépare strictement Modèle, Vue et Contrôleur, permettant de modifier une couche sans impacter les autres.

Le **pattern Strategy** est utilisé pour les comportements des ennemis via l'interface ComportementEnnemi, permettant l'assignation dynamique de différentes IA.

Le **pattern State Machine** gère les états du jeu dans ControleurJeu et la séquence de fin de niveau avec transitions bien définies.

Le **pattern Observer** permet aux collisions et événements de notifier les objets intéressés sans couplage fort.

Le **pattern Factory** dans ChargeurNiveau crée les bonnes instances selon les propriétés lues dans Tiled.

Le **pattern Singleton** pourrait être appliqué aux gestionnaires pour garantir une instance unique.

Principes SOLID Appliqués

Le **principe Single Responsibility** assure que chaque classe a une seule responsabilité clairement définie.

Le **principe Open/Closed** permet d'étendre les fonctionnalités par héritage sans modifier le code existant.

Le **principe Liskov Substitution** garantit que les sous-classes peuvent remplacer leurs classes parentes.

Le **principe Interface Segregation** utilise des interfaces ciblées contenant uniquement les méthodes nécessaires.

Le **principe Dependency Inversion** fait dépendre les classes de haut niveau d'abstractions plutôt que de classes concrètes.

Répartition des Tâches

Moussa CISSE : Développeur Modèle & Logique

Il a conçu la hiérarchie complète des entités avec la classe abstraite Entite et ses implémentations. Il a créé le système complet du joueur avec trois états de transformation, gestion des vies et invincibilité temporaire. Il a implémenté les ennemis terrestres avec le pattern Strategy pour l'IA. Il a développé les quatre types de power-ups et leurs effets. Il a conçu et implémenté le système de collisions avec algorithme AABB et résolution de toutes les interactions. Il a créé les gestionnaires de niveaux et de progression.

Andrea Kocovic : Développeur Vue & Contrôleur

Il a conçu l'interface RenduEntite et implémenté tous les renderers spécifiques. Il a développé le système de rendu multi-couches avec gestion du Z-index. Il a créé l'interface utilisateur avec HUD et menus interactifs. Il a implémenté la séquence de fin de niveau en quatre étapes avec State Machine. Il a développé le contrôleur principal gérant les sept états du jeu et la boucle principale. Il a créé le système de capture des entrées utilisateur. Il a développé ChargeurNiveau pour l'intégration complète avec Tiled. Il a implémenté tous les effets visuels incluant animations, alpha blending et caméra dynamique.

Collaboration

Les deux développeurs ont travaillé en étroite collaboration. L'architecture MVC a permis un travail en parallèle efficace. Des interfaces bien définies ont facilité l'intégration des composants. Des revues de code régulières ont garanti la cohérence. La gestion de version Git avec branches séparées a évité les conflits.

Section 4. Conclusion et Perspectives

Bilan du Projet

Le projet a atteint tous ses objectifs initiaux. Le moteur est fonctionnel et extensible grâce à l'intégration de Tiled. L'architecture MVC est propre et facilite la maintenance. Les mécaniques de gameplay sont complètes avec transformations, power-ups et séquence de fin élaborée. Les principes SOLID et design patterns ont été appliqués rigoureusement. Le jeu est jouable, fluide et agréable.

Plusieurs défis techniques ont été surmontés. La séquence de fin a nécessité une State Machine complexe à quatre étapes avec transitions fluides. Un bug majeur du bouton Rejouer a été résolu par un reset complet de toutes les variables d'état. Les transformations du joueur ont demandé une synchronisation précise entre taille, hitbox et animations. L'effet d'entrée dans le château a été réalisé par gestion du Z-index et alpha blending. Les limites de patrouille des ennemis ont été calculées dynamiquement depuis la position du drapeau.

Perspectives d'Amélioration

À court terme, l'ajout d'un système audio avec musiques et effets sonores améliorerait l'immersion. Un système de sauvegarde permettrait de reprendre sa progression. De nouveaux power-ups pourraient être ajoutés facilement.

À moyen terme, l'implémentation de boss de fin avec patterns d'attaque ajouterait de la difficulté. Des zones secrètes récompenseraient l'exploration. Un mode speedrun avec

chronomètre motiverait les joueurs compétitifs. Un mode multijoueur local permettrait de jouer à deux.

À long terme, le portage vers mobile adapterait les contrôles au tactile. Un éditeur de niveau intégré permettrait aux joueurs de créer leurs propres niveaux. L'amélioration de l'IA rendrait le jeu plus challengeant. La génération procédurale de niveaux permettrait une rejouabilité infinie.

Annexes

Ressources Utilisées

La documentation officielle de LibGDX sur libgdx.com a été précieuse pour comprendre les APIs du framework. La documentation de Tiled sur mapeditor.org a permis de maîtriser l'outil. Plusieurs tutoriels en ligne ont aidé au démarrage, notamment sur l'intégration de Tiled avec LibGDX. IntelliJ IDEA a servi d'IDE principal. Git a assuré le versioning et la collaboration.