Université de Valrose DL2 Maths-InfoAnnée Universitaire 2025 - 2026
RAPPORT DE PROJET
Développement d’un Moteur de Jeu 2D avec LibGDX
Projet : "JRPG Engine"

Réalisé par : Loiseau Maël

Date de rendu : 12 Janvier 2026

Section 1. Introduction

1.1 Contexte du projet

Ce projet s'inscrit dans le cadre de l'unité d'enseignement de Programmation Orientée Objet et de développement logiciel. Il vise à mettre en pratique les concepts fondamentaux de l'architecture logicielle en Java (héritage, polymorphisme, interfaces, gestion des états) à travers la réalisation concrète d'un moteur de jeu vidéo en 2D.

Le choix technologique s'est porté sur LibGDX, un framework Java open-source reconnu pour sa performance et sa flexibilité. Contrairement à des moteurs "clés en main" comme Unity ou Godot, l'utilisation de LibGDX impose de construire soi-même la boucle de jeu, la gestion des rendus graphiques et la physique, offrant ainsi une opportunité pédagogique idéale pour comprendre les rouages internes d'un moteur de jeu.

Le projet a été réalisé de manière individuelle, ce qui a nécessité une gestion autonome de l'ensemble de la chaîne de production : de l'architecture du code (backend) à l'intégration des assets graphiques et au level design.

1.2 Objectifs du projet

L'objectif principal de ce devoir n'était pas seulement de créer un jeu fonctionnel, mais de développer un moteur de jeu extensible piloté par les données (data-driven). La contrainte majeure et directrice du projet était de permettre l'enrichissement du contenu sans nécessiter de modification ou de recompilation du code source Java.

Pour répondre à cette problématique, le moteur a été conçu autour des objectifs techniques suivants :

Intégration poussée avec Tiled Map Editor : Le moteur doit déléguer la configuration du jeu à l'éditeur de cartes. Cela inclut non seulement la topologie du niveau (murs, sols), mais également la logique de jeu : positionnement des zones de collision, points de spawn des ennemis, placement des PNJ (Personnages Non Joueurs) et définition des interactions (dialogues, changements de carte).

Architecture Modulaire : Séparer clairement la logique d'exploration (déplacement sur la carte) de la logique de combat (système au tour par tour typique des JRPG), en utilisant un gestionnaire d'états (State Pattern).

Gestion de données externe : Utiliser des fichiers JSON pour définir les propriétés des objets, des compétences et des boutiques, permettant un équilibrage du jeu dynamique.

En somme, l'ambition de ce projet est de fournir un outil où un game designer pourrait théoriquement créer une nouvelle ville, y placer un marchand et définir ses dialogues, uniquement en manipulant des fichiers .tmx et .json, sans jamais écrire une ligne de Java.

Section 2. Présentation du projet

Cette section détaille les choix techniques, les fonctionnalités offertes par le moteur de jeu, ainsi que la méthodologie pour étendre le contenu via l'éditeur Tiled.

2.1 Technologies et Outils Utilisés

Le projet repose sur un ensemble d'outils standards de l'industrie du développement Java :

LibGDX (Backend LWJGL3) : Utilisé comme framework principal pour gérer le cycle de vie de l'application, le rendu graphique (OpenGL), les entrées (clavier/souris) et l'audio.

Tiled Map Editor : L'outil central pour le level design. Il est utilisé non seulement pour dessiner les décors, mais surtout comme éditeur de données pour configurer les collisions et les entités du jeu.

JSON (JavaScript Object Notation) : Format de données utilisé pour externaliser les statistiques des objets, des compétences et les inventaires des boutiques (items.json, skills.json, shop.json). Cela permet l'équilibrage du jeu sans recompilation.

Gradle : Outil d'automatisation utilisé pour la gestion des dépendances et la compilation du projet.

Git & GitHub : Pour le versionning du code et le suivi de l'avancement.

2.2 Fonctionnalités implémentées

Le moteur de jeu "JRPG" propose les fonctionnalités suivantes, structurées autour d'une architecture par états (State Pattern) :

Exploration et Interactions :

Déplacement du personnage sur une grille (Tile-based) avec gestion des collisions.

Système de transitions entre cartes (Map Switching) via des zones de sortie configurables.

Interaction avec des PNJ (Personnages Non Joueurs) : Lancement de dialogues ou ouverture de menus spécifiques.

Système de Combat (Tour par tour) :

Une scène de combat dédiée (CombatState) qui se déclenche lors des rencontres.

Gestion de l'ordre des tours entre le joueur et les ennemis (ex: Gobelins).

Actions variées : Attaques physiques, Compétences magiques (sorts), Utilisation d'objets (potions) et Fuite.

Gestion de l'Économie et Inventaire :

Système de boutiques (ShopState) permettant d'acheter des équipements et consommables.

Inventaire fonctionnel permettant d'équiper des armes/armures et de consommer des potions.

Persistance des données :

Système de sauvegarde et chargement (SaveManager) permettant de conserver la progression du joueur (position, inventaire, statistiques).

2.3 Configuration et Ajout de Contenu avec Tiled

C'est le cœur du sujet : le moteur a été conçu pour que Tiled agisse comme un éditeur de scène complet. La classe MapLoader.java est responsable de l'interprétation des fichiers .tmx.

Processus d'ajout d'une nouvelle carte :

Création visuelle : Le créateur dessine le décor sur les couches de tuiles (Tile Layers) dans Tiled (herbe, murs, maisons).

Définition des Collisions : Un calque d'objets nommé "Collisions" est utilisé. Tout rectangle dessiné sur ce calque est automatiquement converti en zone infranchissable par le moteur physique (CollisionSystem).

Placement des Entités (PNJ et Ennemis) :

Le créateur place un point ou un rectangle sur un calque d'objets.

Il attribue une propriété personnalisée nommée type (ex: HealerNPC, MerchantNPC, Goblin).

Au chargement, le moteur instancie automatiquement la classe Java correspondante aux coordonnées de l'objet Tiled.

Configuration des Portes (Changement de carte) :

Pour créer un passage vers une autre zone, on crée un objet avec la propriété type = MapExit.

On ajoute une propriété targetMap (ex: foret.tmx) pour indiquer la destination.

Grâce à ce système, il est possible de créer un village entier, peuplé et interactif, uniquement en manipulant l'interface graphique de Tiled.

2.4 Compilation et Exécution

Le projet utilise le "Gradle Wrapper", ce qui signifie qu'il n'est pas nécessaire d'avoir Gradle installé globalement sur la machine, seul un JDK (Java Development Kit) version 17 ou supérieure est requis.

Prérequis :

Java JDK 17+

Une connexion internet (pour le premier téléchargement des dépendances LibGDX).

Étapes de compilation et lancement :

Ouvrir un terminal à la racine du projet JRPG.

Sous Windows : Exécuter la commande :
Bash

gradlew.bat lwjgl3:run

Sous macOS / Linux : Donner les droits d'exécution puis lancer :
Bash

chmod +x gradlew
./gradlew lwjgl3:run

Cette commande va compiler les sources Java, traiter les assets, et lancer la fenêtre de jeu via le module Desktop (lwjgl3).

Section 3. Présentation technique du projet et contributions

Cette section détaille l'architecture logicielle du moteur, les choix de conception orientée objet, et la manière dont le développement a été orchestré.

3.1 Architecture Générale du moteur de jeu

L'architecture du projet repose sur une séparation stricte des responsabilités via l'utilisation de Design Patterns éprouvés, garantissant la modularité et la maintenabilité du code.

Le Patron État (State Pattern) La structure globale du jeu est gérée par une machine à états. La classe principale Main gère un gestionnaire d'écrans (hérité de com.badlogic.gdx.Game) qui permet de basculer fluidement entre les différentes phases du jeu. Chaque phase est encapsulée dans une classe implémentant l'interface IGameState (ou étendant ScreenAdapter) :

ExplorationState : Gère l'affichage de la carte Tiled, le déplacement du joueur, la caméra et la détection des collisions via CollisionSystem.

CombatState : Gère la logique de combat au tour par tour, instanciée uniquement lorsqu'une rencontre survient.

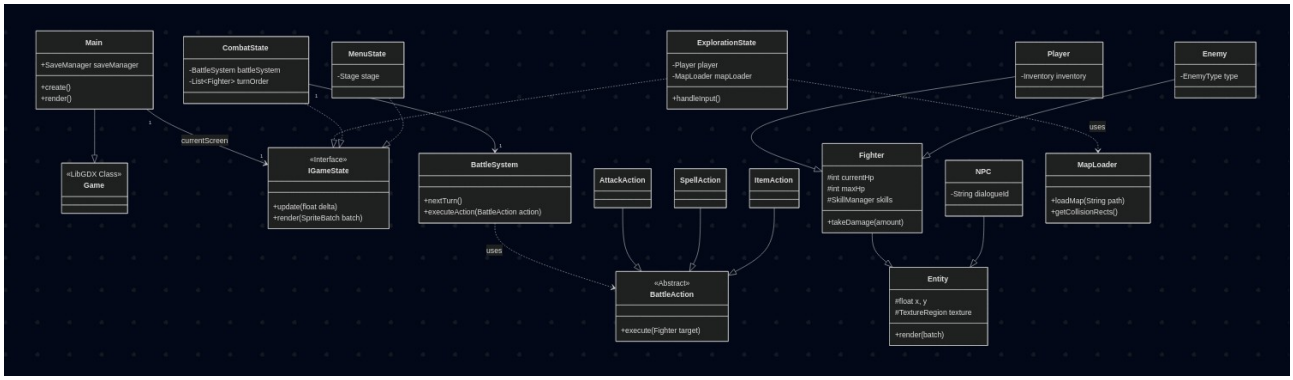
MenuState / ShopState : Gèrent les interfaces utilisateur (UI) et les interactions marchandes.

Modèle MVC (Modèle-Vue-Contrôleur) Bien que LibGDX ne force pas ce modèle, le projet tend vers cette séparation pour clarifier le code :

Modèle (Données) : Les classes du package model.entities (Player, Enemy, SwordMan) contiennent les statistiques et l'état logique des personnages, sans se soucier du rendu. Les données de sauvegarde sont isolées dans SaveData.

Vue (Affichage) : Le rendu est délégué aux méthodes render() des différents états, qui utilisent les SpriteBatch pour dessiner les textures en fonction de l'état du Modèle.

Contrôleur (Logique) : La gestion des entrées utilisateur est capturée par les processeurs d'input (InputProcessor), qui modifient le Modèle (ex: changer les coordonnées x,y du joueur) en réponse aux actions.



"Le diagramme ci-dessus illustre l'architecture modulaire du projet :

Gestion des États : La classe principale Main orchestre les changements d'écrans via l'interface IGameState. Cela permet une séparation nette entre la logique d'exploration (ExplorationState) et la logique de combat (CombatState).

Hiérarchie des Entités : L'utilisation de l'héritage (Entity > Fighter > Player) permet de partager le code de rendu et de positionnement, tout en spécialisant les comportements (seuls les Fighter ont des PV et peuvent combattre).

Système de Combat Extensible : Le pattern Command est utilisé via la classe abstraite BattleAction. Cela permet d'ajouter n'importe quel type d'action (Attaque, Sort, Objet) sans modifier le cœur du BattleSystem, respectant ainsi le principe Open/Closed."

3.2 Utiliser et étendre la librairie du moteur de jeu

La force principale de ce moteur réside dans son extensibilité. L'utilisation du Polymorphisme permet d'ajouter de nouveaux comportements sans briser le code existant.

Ajout de nouvelles actions de combat (Pattern Command/Strategy) : Le système de combat repose sur une classe abstraite ou interface BattleAction. Pour créer un nouveau type d'attaque (par exemple, une attaque drainant de la vie), il suffit de créer une classe LifeStealAction héritant de BattleAction et d'implémenter la méthode execute(). Le BattleSystem pourra traiter cette nouvelle action de manière transparente, sans modification de son propre code.

Ajout de nouveaux types d'ennemis : Pour ajouter un monstre spécifique, un développeur doit :

Créer une classe héritant de Enemy (ex: DragonEnemy).

Définir ses statistiques dans le constructeur.

Dans Tiled, placer un objet avec la propriété type = Dragon. Le MapLoader utilisera la réflexion ou une factory simple pour instancier la bonne classe Java au chargement de la carte.

Ajout d'objets et compétences (Data-Driven) : Pour les éléments qui ne nécessitent pas de logique spécifique (comme une nouvelle épée ou une potion standard), aucune programmation Java n'est

requise. Il suffit d'ajouter une entrée dans le fichier assets/data/items.json. Le ItemLoader se charge de convertir ces données JSON en objets Java manipulables par le jeu.

3.3 Répartition des Tâches (Gestion de projet individuel)

Ce projet ayant été réalisé de manière individuelle, j'ai assumé l'intégralité des rôles nécessaires à la production d'un jeu vidéo. Pour maintenir une rigueur professionnelle, j'ai séquencé le développement en plusieurs "casquettes" distinctes :

Architecte Logiciel (Backend) :

Conception de la structure des classes et des interfaces (IGameState, Entity).

Mise en place du système de chargement de cartes (MapLoader) et de la gestion des collisions.

Développement du système de sauvegarde (SaveManager).

Programmeur Gameplay :

Implémentation du système de combat au tour par tour (BattleSystem).

Codage de la logique des compétences (SkillManager) et de l'inventaire.

Programmation des interactions PNJ (dialogues, boutiques).

Level Designer & Intégrateur :

Création des cartes sous Tiled Map Editor (Village, Forêt, Intérieurs).

Configuration des couches d'objets pour définir les zones de collision et les transitions.

Gestion et découpage des SpriteSheets pour les animations des personnages.

Cette centralisation des tâches a permis une cohérence totale entre le code et les données : connaissant parfaitement la structure du SaveData, il a été plus simple d'implémenter les mécanismes de persistance sans friction entre différentes équipes.

Section 4. Conclusion et Perspectives

4.1 Bilan du projet

Ce projet de développement d'un moteur de jeu JRPG avec LibGDX a été une expérience enrichissante, permettant de cristalliser les compétences théoriques acquises en Programmation Orientée Objet.

Objectifs atteints : L'objectif principal, qui était de concevoir un moteur "orienté données" (data-driven), a été rempli. La distinction entre le code (Java) et le contenu (Tiled/JSON) est fonctionnelle. Il est aujourd'hui possible de créer une carte complète, d'y placer des ennemis et des interactions, et de jouer ce niveau sans modifier une seule ligne de code source. L'architecture basée sur les états (State Pattern) assure une navigation fluide entre l'exploration, les combats et les menus.

Défis rencontrés : En tant que développeur unique, le principal défi a été la gestion de la complexité croissante du projet.

Gestion des transitions : Conserver l'état du joueur (position exacte, points de vie) lors du passage de l'état ExplorationState à CombatState puis au retour sur la carte a nécessité une gestion rigoureuse des instances.

Logique de Collision : L'intégration des calques d'objets de Tiled avec le système de coordonnées de LibGDX a demandé un temps d'adaptation pour gérer correctement les "hitboxs".

Polyvalence : Il a fallu alterner constamment entre le rôle de développeur (logique pure), d'intégrateur (gestion des assets graphiques) et de concepteur de niveaux.

4.2 Perspectives d’amélioration

Bien que fonctionnel, le moteur possède une marge de progression importante. Voici les fonctionnalités prioritaires qui pourraient être implémentées dans une version future ("V2") :

Système de Quêtes et Dialogues avancés : Actuellement, les PNJ ont des dialogues simples. Une amélioration majeure serait d'implémenter un système de gestion de quêtes (suivi d'état "Non commencée", "En cours", "Terminée") stocké dans le SaveData, influençant dynamiquement les réponses des PNJ.

Animations de Combat enrichies : Le système de combat est statique. L'utilisation d'une librairie d'interpolation (comme LibGDX TweenEngine) permettrait d'animer les déplacements des sprites lors des attaques pour rendre l'action plus visuelle.

Intelligence Artificielle (IA) des ennemis : Les ennemis agissent actuellement de manière aléatoire ou séquentielle. L'intégration du pattern Strategy permettrait de donner des comportements spécifiques (ex: un soigneur ennemi qui ne soigne que si un allié a moins de 50% de PV).

Interface Utilisateur (UI) avec Scene2D : L'interface actuelle est fonctionnelle mais basique. Une refonte utilisant le module Scene2D de LibGDX permettrait de créer des menus plus ergonomiques, avec des fenêtres redimensionnables et des effets de transition.

Annexes

Bibliographie et Ressources Documentaires

Pour mener à bien ce projet, je me suis appuyé sur la documentation officielle et des ressources communautaires :

Documentation LibGDX : https://libgdx.com/wiki/

Référence principale pour la gestion du cycle de vie de l'application et le rendu des TileMaps.

Documentation Tiled Map Editor : https://doc.mapeditor.org/

Utilisée pour comprendre le formatage des propriétés personnalisées des objets.

Tutoriels et Communauté :

LibGDX Discord & Java-Gaming.org pour la résolution de bugs spécifiques liés à Gradle.

Assets graphiques : Les sprites utilisés (personnages et tuiles) proviennent de packs libres de droits (OpenGameArt / Itch.io) adaptés pour les besoins du projet.

Lien vers le dépôt du projet

L'intégralité du code source, ainsi que l'historique des commits montrant l'évolution du projet, est disponible à l'adresse suivante :

GitHub : https://github.com/LoiseauMael/ProjetJava