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

Projet : DungeonEngine

Équipe : Diers Kylian

Section 1. Introduction

Ce projet s'inscrit dans le cadre de l'unité d'enseignement de Programmation et Conception Orientée Objet. L'objectif était de créer un moteur de jeu le plus facilement modifiable et avec la meilleure architecture possible.

Le contexte choisi est celui d'un Dungeon Crawler/Roguelite. Le joueur explore des donjons générés dynamiquement, combat des monstres et progresse dans des niveaux de difficulté croissante (il est même possible de combiner cela à des cartes originales comme dans un RPG).

Pour réaliser ce moteur, j'ai utilisé la librairie LibGDX et l'éditeur de cartes Tiled mais aussi des fichiers de configuration JSON et appliqué les concepts importants de la programmation orientée objets (MVC, Design Patterns).

Section 2. Présentation du projet

Technologies et Outils Utilisés

- **LibGDX (Java) :** pour la gestion du cycle de vie du jeu, le rendu graphique et les entrées.
- **Tiled :** pour créer des cartes, faire le lien entre elles, mais également créer des salles préfabriquées facilement ajoutables à la génération procédurale développée en amont.
- **JSON** : pour créer des entités de A à Z sans toucher au code Java, simplement en modifiant le fichier JSON
- **Gradle :** pour la compilation
- **Git :** pour la gestion de versions

Fonctionnalités implémentées

- Chargement dynamique des entités basé sur la configuration externe entities.json
- Génération procédurale
- Système de combat avec des entités
- Comportements d'ennemis interchangeables (Poursuite, Statique, Soigneur/Interactif) avec utilisation d'un Strategy Pattern
- Persistance des données (Pièces, Niveau) via un Singleton GameState, difficulté croissante après chaque niveau, enregistrement du meilleur niveau atteint.

Configuration et ajout de contenu

Pour ajouter une carte, il suffit de déposer une carte Tiled dans le dossier "maps" du projet. A l'intérieur d'une carte, il doit y avoir les calques de tuiles suivants : walls, demi_walls, floor et les calques d'objets entities et collisions.

Les collisions sont calculées automatiquement pour les murs placés dans le calque walls, et le calque demi_walls sert à calculer les collisions des barrières verticales souvent présentes dans les cartes.

Le calque objet collisions sert à ajouter des collisions personnalisées, si besoin.

Le calque objet entities permet de définir le spawn du joueur (insérer un objet avec la propriété type et la valeur spawn) mais aussi de définir l'emplacement de spawn des monstres en écrivant le même nom que celui présent dans assets/data/entities.json (par exemple type=big_monster)

Il est aussi possible de créer des portails entre différentes cartes en ajoutant un objet dans le calque objet entities ayant pour propriété destination et comme valeur le chemin vers la

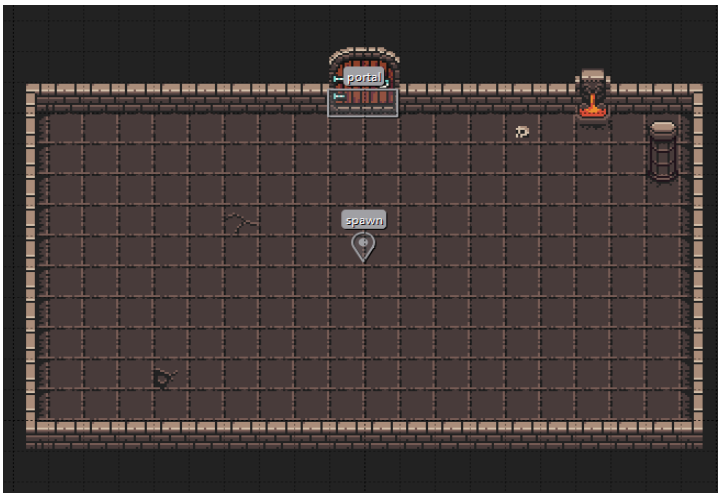
carte (par exemple maps/prefabs/room_start.tmx). En écrivant comme valeur "GENERATE", le joueur est téléporté vers une carte générée "procéduralement", à l'aide des préfabriqués dans le dossier maps/prefabs. Ajouter un préfabriqué de la forme room_i (i un nombre de 1 à n) dans le dossier préfabs l'ajoutera automatiquement aux préfabriqués séléctionnés aléatoirement pour la création de la carte procédurale.

Il est également possible d'ajouter des monstres entièrement nouveaux avec le json dans data/entities.json, en spécifiant les données nécéssaires.

Exemple d'entités :

```
"big_demon": {
  "health": 40,
  "speed": 40,
  "damage": 1,
  "width": 32,
  "height": 36,
  "texture": "anims/big_demon/",
  "behavior": "ChaseBehavior"
},
"dwarf_npc": {
  "health": 100,
  "speed": 0,
  "width": 16,
  "height": 28,
  "texture": "anims/dwarf_m/",
  "behavior": "HealerBehavior"
```

Exemple de carte (hub)



Instructions de Compilation

- Windows : run.bat
- Linux/Mac : ./run.sh
- Commande Gradle : ./gradlew desktop:run

Lien GitHub : https://github.com/KytowDev/MoteurRPG2D

Section 3. Architecture Technique et Conception

L'architecture a été conçue autour d'une séparation stricte des responsabilités (Pattern MVC) et d'une gestion de données externes pour maximiser l'extensibilité.

Noyau (Core et Screens)

- **Main** : Point d'entrée, hérite de com.badlogic.gdx.Game. Il gère le cycle de vie global et le SpriteBatch.
- **PlayScreen** : Agit comme le chef d'orchestre de la boucle de jeu. Il instancie le GameWorld (Modèle), l'EntityRenderer (Vue) et gère la caméra. C'est ici que le lien entre le temps (delta) et la logique est fait.
- **GameState (Singleton)** : Un gestionnaire d'état global accessible partout. Il assure la persistance des données critiques (points de vie, or, niveau actuel, meilleur score) lors des transitions entre le Hub et les donjons générés.

Modèle

Respectant les principes de la POO, totalement indépendant de l'affichage graphique.

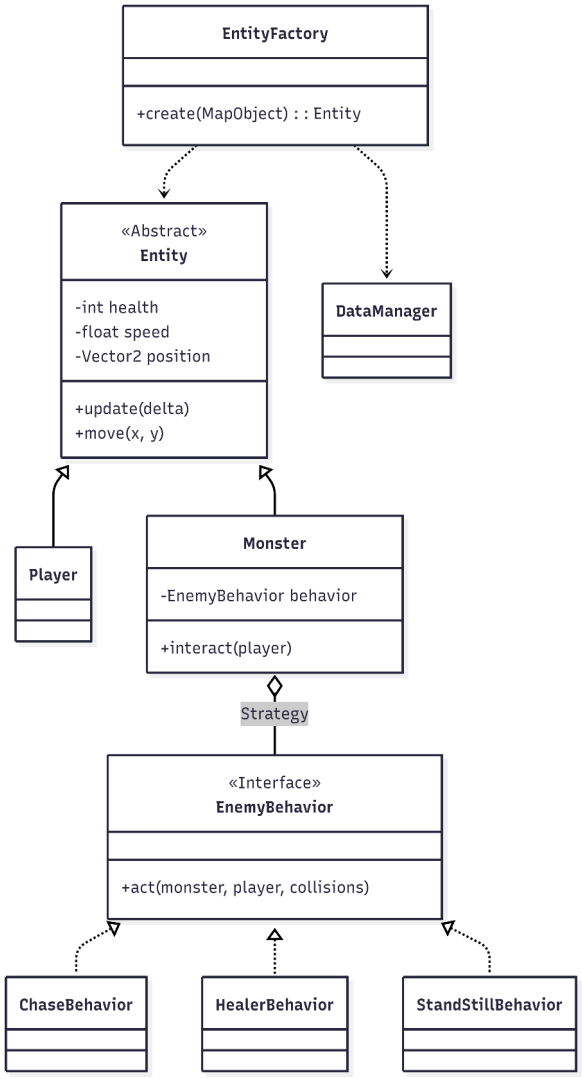
- **GameWorld** : Conteneur principal. Il stocke la liste des entités, gère la carte Tiled, calcule les collisions avec l'environnement et traite les portails.
- **Entity (Classe Abstraite)** : Classe mère de tous les objets dynamiques (Player, Monster). Elle gère la physique de base (position, hitbox, vélocité, état de santé).
- **PlayerController** : Sépare la gestion des entrées clavier/souris de l'entité Player. Il convertit les inputs utilisateur en actions concrètes sur le modèle.

IA

Utilisation du Strategy Pattern avec EnemyBehavior (Interface) qui définit le contrat act(…) et des stratégies concrètes : ChaseBehavior , StandStillBehavior, HealerBehavior.

Factory et données

On a DataManager qui charge et parse le fichier entities.json, et EntityFactory qui utilise la réflexion java pour instancier dynamiquement les classes. Elle lit une chaîne de caractères dans le JSON (ex: "ChaseBehavior") et instancie la classe Java correspondante sans avoir besoin d'un switch/case codé en dur.



Le moteur a été conçu pour permettre à un développeur tiers d'ajouter du contenu sans modifier le cœur du moteur (GameWorld ou PlayScreen). L'extension se fait principalement par **Héritage** et **Configuration**.

Exemple 1 : Ajouter une nouvelle Intelligence Artificielle

Pour créer un nouveau comportement de monstre (ex: un ennemi qui fuit ou qui tire à distance) :

1. Créer une nouvelle classe Java implémentant l'interface **EnemyBehavior**.
2. Implémenter la méthode act() avec la logique désirée.
3. Dans le fichier entities.json, assigner le nom de cette classe au champ "behavior" de l'ennemi. Aucune modification de la Factory n'est nécessaire grâce à l'instanciation dynamique.

Exemple 2 : Ajouter un nouveau type d'arme

Le système d'armes repose sur l'interface **Weapon**. Pour ajouter une arme :

1. Créer une classe implémentant Weapon (ex: Bow, Axe).
2. Définir ses statistiques (portée, dégâts, cooldown) dans les méthodes getter.
3. Implémenter la logique de hitbox dans la méthode attack().
4. Ajouter la texture correspondante dans le dossier assets avec la convention de nommage weapon_[type].png.

Exemple 3 : Ajouter de nouvelles entités

Cela ne nécessite aucun code Java. L'utilisateur doit simplement :

1. Ajouter une entrée dans data/entities.json définissant les statistiques (HP, Vitesse, Texture).
2. Ouvrir l'éditeur Tiled.
3. Placer un objet dans le calque d'objets entites avec la propriété "type" correspondant au nom défini dans le JSON.

Section 4. Conclusion et Perspectives

Ce projet a permis de mettre en pratique les concepts de PCOO dans un contexte complexe et de faire un moteur de jeu robuste grâce à l'architecture. Les principaux défis ont été de passer d'une architecture "classique" à une architecture MVC et trouver les bons designs patterns à implémenter pour avoir un code propre et facile à maintenir.

Perspectives :

- Ajouter une mécanique de tir (pistolets,…)
- Étendre la génération procédurale avec des clés et des portes verrouillées
- Ajout de menu principal, écran de mort
- Ajout d'un système d'éclairage dynamique pour l'immersion