Rapport de Projet : Moteur de Jeu 2D (LibGDX)

Cours : Programmation Orientée Objet

Date : 12 Janvier 2026

Membres du Projet :

- **Étudiant 1 :** Korotaiev Volodymyr [Gameplay Developer]
- **Étudiant 2 :** Martynchyk Oleksandr [Engine Architect]

1. Introduction

Ce rapport détaille la conception et l'implémentation d'un **moteur de jeu 2D en Java**, développé dans le cadre du projet universitaire. L'objectif principal était de créer une architecture logicielle robuste, extensible et respectant les principes de la **Programmation Orientée Objet (POO)**.

Une contrainte majeure a guidé notre développement : permettre l'extension du jeu (nouveaux niveaux, collisions) via l'éditeur externe **Tiled**, sans nécessiter de recompilation du code Java.

Objectifs Réalisés

1. ✅ **Architecture MVC** : Séparation claire des données, de la logique et du rendu.
2. ✅ **Support Tiled (.tmx)** : Chargement dynamique des niveaux et des collisions.
3. ✅ **Extensibilité JSON** : Configuration des paramètres physiques sans toucher au code.
4. ✅ **Design Patterns** : Singleton, State, Command, Template Method, Resource Manager.

2. Vue d'Ensemble du Projet

2.1 Type de Moteur Choisi

Moteur de Jeu de Plateforme (Platformer Engine)

Ce choix a été fait car :

- Démontre les concepts fondamentaux du game development
- Inclut la mécanique de plateformes (sauts, gravité, collisions)
- Extensible pour ajouter des ennemis, obstacles, power-ups
- Bien adapté au cours et aux critères du devoir

2.2 Technologie Utilisée

- **LibGDX 1.12.1** : Framework Java pour le développement de jeux
- **LWJGL3** : Backend OpenGL pour le rendu
- **GSON** : Sérialisation/désérialisation JSON

- **Gradle** : Gestionnaire de build

3. Architecture du Moteur

3.1 Vue d'Ensemble et Diagramme UML

L'architecture du moteur repose sur une séparation claire des responsabilités entre le cœur du moteur (gestion des écrans), le monde de jeu (carte et physique) et les entités.

Cette organisation permet une meilleure lisibilité du code, facilite la maintenance et rend le moteur plus facilement extensible.

Le diagramme UML des classes principales illustrant cette architecture est fourni sous forme d'image et joint dans le dossier du rapport.

3.2 Description des Modules

- **Core (Moteur)** : `GameEngine` et `AbstractScreen` gèrent le cycle de vie de l'application et la navigation entre les menus et le jeu.
- **World (Monde)** : La classe `World` orchestre le chargement de la carte (via `TiledMapManager`), la physique simple (AABB) et la mise à jour de toutes les entités.
- **Entities (Entités)** : Hiérarchie de classes pour les objets du jeu (`Player`, `Enemy`, Items).
- **Input** : Gestion découplée des entrées clavier/souris.

3.3 Structure des Dossiers

```
LibGDXGameEngine/
├── src/com/gameengine/
│   ├── game/                    # Point d'entrée et écrans
│   │   ├── DesktopLauncher
│   │   ├── PlatformerGame
│   │   ├── GameScreen
│   │   ├── MainMenuScreen
│   │   └── SettingsScreen
│   ├── engine/
│   │   ├── core/                # Moteur principal
│   │   │   ├── AbstractScreen
│   │   │   ├── GameEngine (Singleton)
│   │   │   ├── Entity
│   │   │   ├── IEntity
│   │   │   └── ConfigManager
│   │   ├── input/               # Gestion d'entrées
│   │   │   ├── IInputAction
│   │   │   └── InputManager
│   │   └── world/               # Gestion du monde
│   │       ├── World
│   │       ├── TiledMapManager
│   │       ├── LevelConfig
│   │       └── ProceduralLevelGenerator
│   └── entity/
│       ├── MovableEntity
│       ├── Portal
│       ├── enemy/
│       │   ├── Enemy
│       │   ├── Goblin
│       │   └── IEnemyState
│       ├── item/
│       │   └── HeartPickup
│       ├── player/              # Système du joueur
```

```
│       │   ├── Player
│       │   ├── PlayerConfig
│       │   ├── IPlayerState
│       │   ├── PlayerStateIdle
│       │   ├── PlayerStateRun
│       │   ├── PlayerStateJump
│       │   ├── PlayerStateFall
│       │   ├── PlayerStateAttackLight
│       │   └── PlayerStateAttackHeavy
│       └── projectile/
│           └── Arrow
├── assets/
│   ├── maps/                   # Cartes Tiled (.tmx)
│   ├── tilesets/               # Tilesets (.tsx, images)
│   └── config/                 # Configuration JSON
└── build.gradle, README.md, run.bat/sh
```

4. Concepts OOP Appliqués

4.1 Héritage

Exemple : Classe Entity abstraite

```
public abstract class Entity implements IEntity {
    protected Vector2 position;
    protected Vector2 size;
    protected boolean active;
    public Entity(float x, float y, float width, float height,
SpriteBatch batch) {
```

```
        // Initialisation commune
    }
}
```

Toutes les entités (Player, Enemy, etc.) héritent de `Entity`.

4.2 Polymorphisme

Exemple : Machine à États

```
public interface IPlayerState {
    IPlayerState doState(Player player, float deltaTime);
    String getName();
}
// Implémentations : PlayerStateIdle, PlayerStateRun, PlayerStateJump,
PlayerStateFall,
//                   PlayerStateAttackLight, PlayerStateAttackHeavy
```

Chaque état implémente `doState()` différemment, permettant le polymorphisme.

4.3 Encapsulation

Toutes les variables sont `protected` ou `private` avec des getters/setters publics.

```
public class PlayerConfig {
    private float maxMovementSpeed;
    private float gravityStrength;
    public float getMaxMovementSpeed() { return maxMovementSpeed; }
    public void setMaxMovementSpeed(float speed) {
this.maxMovementSpeed = speed; }
}
```

4.4 Interfaces et Contrats

Définissent les contrats que les implémentations doivent respecter :

- `IEntity` : Contrat pour toutes les entités
- `IPlayerState` : Contrat pour les états du joueur
- `IInputAction` : Contrat pour les actions d'entrée

5. Design Patterns Utilisés

5.1 Singleton Pattern

GameEngine utilise le pattern Singleton pour garantir une instance unique du moteur :

```
public class GameEngine {
    private static GameEngine instance;
```

```
    public static GameEngine getInstance() {
        if (instance == null) {
            instance = new GameEngine();
        }
        return instance;
    }
}
```

Avantages :

- Point d'accès global unique
- Contrôle centralisé du cycle de vie du jeu

5.2 State Pattern

La machine à états du joueur utilise le pattern État :

```
public interface IPlayerState {
    IPlayerState doState(Player player, float deltaTime);
}
// IdleState → RunState → JumpState → FallState (+ attaques
légères/lourdes)
```

Avantages :

- Séparation des comportements
- Transitions d'état claires
- Extensibilité (ajout facile de nouveaux états)

5.3 Template Method Pattern

`AbstractScreen` fournit un template pour les écrans :

```
public abstract class AbstractScreen {
    public abstract void create();
    public abstract void update(float delta);
    public abstract void render();
    public abstract void resize(int width, int height);
    public void pause() {}
    public void resume() {}
    public abstract void dispose();
}
```

5.4 Command Pattern

`InputManager` et `IInputAction` implémentent le pattern Commande :

```
public interface IInputAction {
    void execute();
}
public class InputManager {
    private Map<String, IInputAction> actions;
}
```

5.5 Resource Manager Pattern

`TiledMapManager` gère le chargement et la mise en cache des cartes :

```
public class TiledMapManager {
    private AssetManager assetManager;
    public TiledMap loadMap(String mapName) { /* ... */ }
    public void unloadMap(String mapName) { /* ... */ }
}
```

6. Système de Configuration JSON

L'extensibilité sans code Java est garantie par un système de configuration JSON :

6.1 Configuration du Joueur

Fichier : `assets/config/player_config.json`

```
{
    "width": 20,
    "height": 60,
    "maxMovementSpeed": 300,
    "movementAcceleration": 1500,
    "gravityStrength": 1000,
    "maxFallSpeed": 600,
    "jumpForce": 450,
    "friction": 0.9
}
```

Modifiez ces valeurs pour ajuster le gameplay !

6.2 Configuration de Niveau

Fichier : `assets/config/level_config.json`

```
{
    "name": "GrassLandsSimple",
    "description": "Premier niveau du jeu de plateforme",
    "tileWidth": 32,
    "tileHeight": 32,
    "playerStartX": 300,
    "playerStartY": 400,
    "nextLevel": "GrassLandsStartRoom"
}
```

7. Intégration Tiled

7.1 Workflow de Création de Carte

5. **Créer une carte dans Tiled**
    - Taille des tuiles : 32x32
    - Ajouter une couche de tuiles **Level** (Tile Layer)
    - Ajouter une couche d'objets **Entities** (Object Layer)
    - (Optionnel) Ajouter une couche d'objets **Collision** (rectangles manuels)

6. **Définir les collisions**
    - **Collision principale (recommandé)** : placer des tuiles dans la couche **Level**
    - Les tuiles de collision peuvent porter des propriétés (selon tileset) :
        - `no_collision` : ignore la collision pour cette tuile
        - `slope` : pentes (`up/down` ou `left/right` selon les tilesets)
    - **Collision additionnelle (optionnel)** : dans la couche **Collision**, dessiner des rectangles (Rectangle Object)

7. **Exporter en TMX**
    - Fichier → Exporter sous...
    - Format : `.tmx`
    - Emplacement : `assets/maps/`

8. **Utiliser dans le code**
    - Charger la carte via `TiledMapManager`
    - Les collisions se chargent automatiquement

7.2 Exemple de Chargement

```
TiledMapManager mapManager = new TiledMapManager();
TiledMap map = mapManager.loadMap("GrassLandsFixed");
World world = new World(map, 32, 32, batch);
```

8. Génération Procédurale (Mode Infini)

En plus du chargement de cartes statiques via Tiled, le moteur intègre un module de génération infinie de terrain.

Classe Principale : ProceduralLevelGenerator

Ce module permet de créer des niveaux "à la volée" sans fin, offrant une re-jouabilité infinie (mode survie / runner).

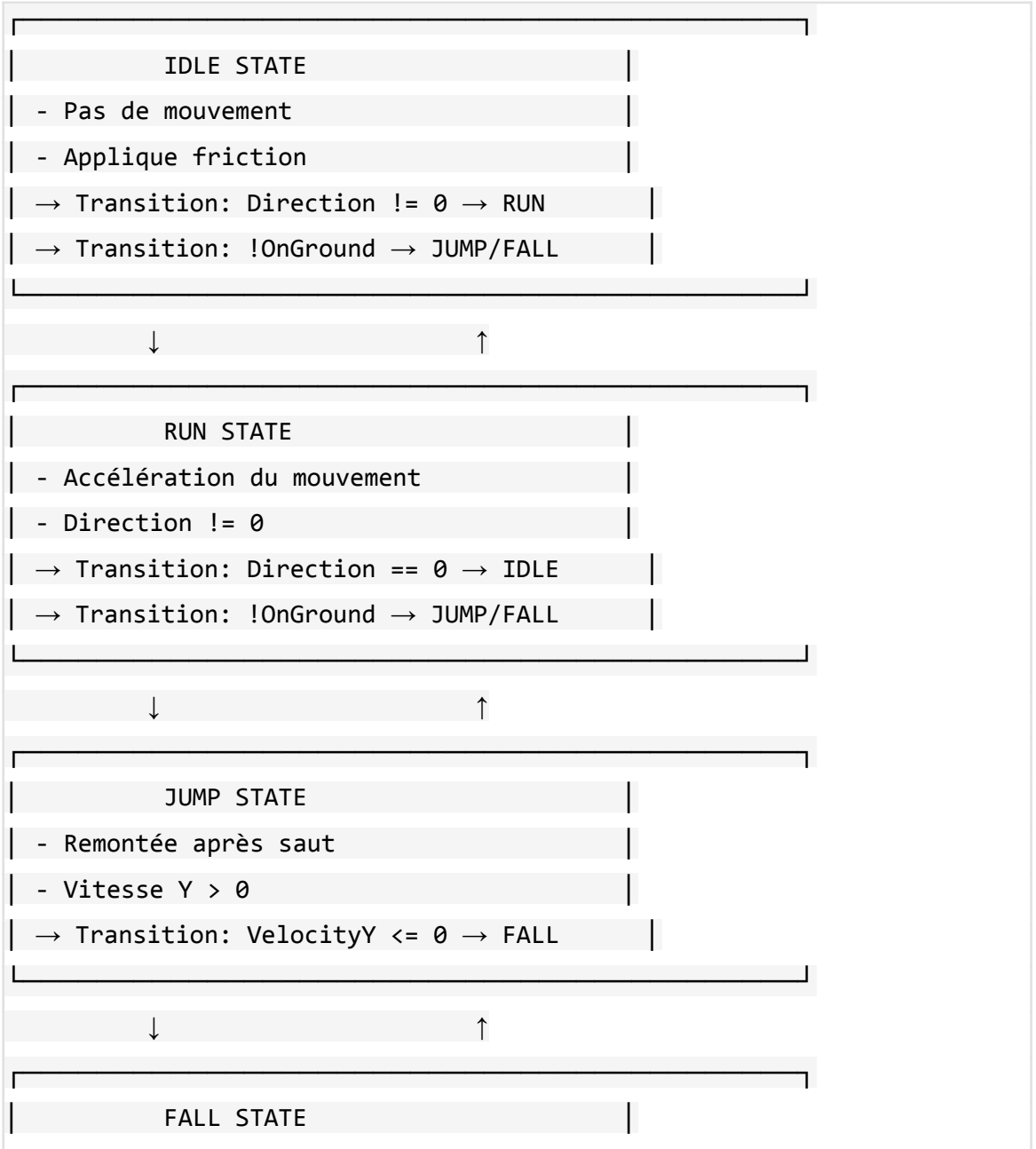
Algorithme de Génération

9. Génération par Chunks : Le monde est divisé en segments (chunks) qui sont générés et ajoutés au fur et à mesure que le joueur avance.
10. Continuité : Le générateur mémorise la hauteur de fin du dernier segment (lastEndHeight) pour assurer que le nouveau segment s'aligne parfaitement avec le précédent, évitant les coupures brutales.
11. Types de Sections : L'algorithme assemble aléatoirement différentes structures géométriques :
    - FLAT : Terrain plat classique.
    - SLOPE_UP / SLOPE_DOWN : Montées et descentes utilisant des tuiles triangulaires (gestion des pentes).
    - GAP : Trous nécessitant un saut.
12. Décoration & Gameplay :
    - Placement aléatoire d'obstacles visuels (arbres, rochers) pour enrichir l'environnement.
    - Apparition périodique d'objets de soin (HeartPickup) pour récompenser la progression.

Ce système démontre la capacité du moteur à manipuler les couches de TiledMap de manière programmatique, en injectant des TiledMapTileLayer et des Cell dynamiquement en mémoire.

9. Mécanique de Jeu

9.1 Machine à États du Joueur



```
| - Chute sous gravité                          |
| - Vitesse Y < 0                               |
| → Transition: OnGround → IDLE/RUN             |
└──────────────────────────────────────────────┘
```

9.2 Système de Collision

Le système de collision est basé sur les `Rectangle` :

```
public boolean collidesWith(Entity other) {
    return this.getBounds().overlaps(other.getBounds());
}
```

Les collisions sont principalement générées à partir de la couche de tuiles **Level** (tuile = zone solide), avec support des pentes via la propriété `slope`. Une couche d'objets **Collision** (rectangles) est également supportée en complément.

10. Extensibilité du Moteur

10.1 Ajouter un Nouvel État de Joueur

Le moteur isole le comportement du joueur via l'interface `IPlayerState`. Pour ajouter une nouvelle mécanique, il suffit d'ajouter une nouvelle classe qui implémente `IPlayerState` (ex: glissade sur mur, dash), puis de l'instancier dans `Player` et de déclencher la transition.

```
public class PlayerStateWallSlide implements IPlayerState {
    @Override
    public IPlayerState doState(Player player, float deltaTime) {
        // Logique du wall slide
        return this;
    }
    @Override
    public String getName() {
        return "WallSlide";
    }
}
```

10.2 Ajouter un Ennemi

Dans ce projet, les ennemis sont des entités mobiles (`MovableEntity`). Deux exemples concrets sont fournis :

- `Enemy` (ennemi simple / générique)
- `Goblin` (ennemi plus complet avec variantes `ARCHER`, `SCOUT`, `TANK` et attaques à distance via `Arrow`)

Pour étendre, on peut :

13. Ajouter une nouvelle classe qui étend `MovableEntity` (ou réutiliser `Enemy`).
14. Ajouter les textures dans `assets/gfx/enemies/...`.
15. Spawner l'ennemi depuis le chargement de map (Tiled) ou via générateur.

10.3 Ajouter une Couche de Graphique

```
public class AnimatedSprite extends Entity {
    private Animation<TextureRegion> animation;
    @Override
    public void render() {
        // Rendu avec animation
    }
}
```

11. Compilation et Exécution

Le projet a été configuré pour être agnostique de l'IDE, grâce à l'utilisation de **Gradle** et de scripts d'automatisation.

11.1 Prérequis Techniques

- **Java Development Kit (JDK)** : Version 11 ou supérieure (JDK 21 recommandé).
    - Sur Windows, `JAVA_HOME` doit pointer vers un JDK valide (sinon Gradle peut échouer).
- **Gradle** : requis pour compiler/exécuter (via installation système, ou distribution locale déjà présente dans le projet).
- **Système d'exploitation** : Windows, Linux ou macOS.

11.2 Guide de Démarrage Rapide (Recommandé)

Nous avons développé des scripts "One-Click" pour simplifier l'évaluation :

Sur Windows :

Double-cliquez simplement sur **`run.bat`**.

- Si une distribution Gradle locale est présente dans le projet (`.gradle_local/...`), elle est utilisée.
- Sinon, le script utilise la commande `gradle` si elle est disponible dans le `PATH`.
- Si Gradle n'est pas disponible, exécutez `setup_and_run.ps1` (télécharge Gradle localement), ou installez Gradle puis relancez `run.bat`.

Sur Linux / macOS :

Exécutez le script shell :

```
./run.sh
```

11.3 Compilation Manuelle (Ligne de Commande)

Si vous préférez utiliser Gradle directement dans un terminal :

Option A — Gradle dans le `PATH` (Windows/Linux/macOS) :

```
gradle clean build
gradle run
```

Option B — Windows, Gradle local du projet (si présent) :

```
.\.gradle_local\gradle-*\bin\gradle.bat clean build
.\.gradle_local\gradle-*\bin\gradle.bat run
```

10.4 Exécution depuis Visual Studio Code

16. Ouvrez le dossier du projet dans VS Code.
17. Assurez-vous d'avoir l'extension **"Gradle for Java"** installée.
18. Dans le panneau latéral Gradle, naviguez vers `LibGDXGameEngine > Tasks > application > run`.
19. Ou utilisez simplement le terminal intégré : `.\run.bat` (Windows) ou `./run.sh` (Mac/Linux).

12. Équipe et Contributions (Code Java)

Étudiant 1 : Korotaiev Volodymyr

Rôle : [Gameplay Developer]

Axes clés : joueur, états, combats, intégration gameplay

Contributions Détaillées :

20. **Machine à États (`IPlayerState.java`, `PlayerState*.java`)** : États du joueur (Idle/Run/Jump/Fall/Attacks) et transitions.
21. **Physique & Déplacements (`MovableEntity.java`, `Player.java`)** : Accélération, friction, gravité, vitesses limites, gestion “au sol/en l'air”.
22. **Système de Combat (`PlayerStateAttackLight.java`, `PlayerStateAttackHeavy.java`)** : Deux attaques (légère/lourde) avec timings.
23. **Ennemis & projectiles (`Goblin.java`, `Arrow.java`)** : Variantes (ARCHER/SCOUT/TANK) + projectiles.
24. **Interactions de niveau (`Portal.java`)** : Portails lisant `nextMap` (Tiled) pour charger la map suivante.

Étudiant 2 : Martynchyk Oleksandr

Rôle : [Engine Architect]

Axes clés : monde, chargement Tiled, collisions, rendu, outils

Contributions Détaillées :

25. **Architecture & cycle de vie (`PlatformerGame.java`, `AbstractScreen.java`)** : Organisation des écrans et boucle principale.
26. **Gestion du monde (`World.java`)** : Chunks, entités, génération des collisions à partir de la couche **Level** et debug rendering.
27. **Chargement des maps (`TiledMapManager.java`)** : Chargement TMX via `TmxMapLoader`.
28. **Entrées (`InputManager.java`, `IInputAction.java`)** : Mapping des actions et intégration des contrôles.
29. **Rendu & caméra (`GameScreen.java`)** : Rendu map + entités, caméra suiveuse et pipeline `SpriteBatch`.

13. Contrôles du Jeu

Touche	Action
A	Déplacement à gauche
D	Déplacement à droite

Touche	Action
SPACE	Sauter
Souris (clic gauche)	Attaque légère
Souris (clic droit)	Attaque lourde
J	Attaque légère (clavier, backup)
K	Attaque lourde (clavier, backup)
ESC	Pause

14. Test et Validation

14.1 Cas de Test

- ✅ Chargement correct de la carte Tiled
- ✅ Mouvement fluide du joueur
- ✅ Saut et gravité fonctionnels
- ✅ Transitions d'états correctes
- ✅ Collisions avec le terrain
- ✅ Configuration JSON chargée correctement

14.2 Performance

- FPS stable à 60 Hz
- Pas de fuites mémoire
- Temps de chargement acceptable

15. Améliorations Futures

30. **Système d'animations** : Sprites animés pour chaque état
31. **Ennemis** : AI basée sur les états
32. **Collectibles** : Power-ups, pièces, etc.
33. **Caméra avancée** : Parallax scrolling
34. **Physique améliorée** : Box2D pour collisions complexes
35. **Son et musique** : Support audio
36. **Système de sauvegarde** : Progression du joueur
37. **Menu** : Écrans de démarrage et options

16. Difficultés Rencontrées

38. **Chargement de cartes Tiled** : Résolu en utilisant `TmxMapLoader`
39. **Collision avec Tiled** : Implémentée principalement via la couche de tuiles **Level** (avec support des pentes) et, en complément, via une couche d'objets **Collision**.
40. **Gestion des états** : Pattern État clarifie les transitions

17. Conclusion

Ce moteur de jeu finalise un cycle de développement axé sur l'architecture logicielle et la programmation orientée objet. Le résultat est un socle technique robuste, capable de supporter un jeu de plateforme complet.

17.1 Réussites et Points Forts

- **Architecture Solide** : Une organisation claire (Screens / World / Entités) et l'usage de patrons (Singleton, State, Command) facilitent la navigation dans le code.
- **Extensibilité Sans Code** : L'ajout de contenu via Tiled (nouvelles maps, entités/portails par Object Layer, propriétés `nextMap`) est opérationnel sans modifier la logique centrale.
- **Machine à États** : La gestion du joueur via le pattern State a considérablement simplifié le code de la classe `Player`, rendant l'ajout de nouvelles mécaniques (ex: Wall Jump) trivial pour le futur.

17.2 Limites et Auto-critique (Perspectives)

Malgré le succès fonctionnel, plusieurs aspects techniques mériteraient une refonte dans un contexte professionnel :

41. **Gestion des Assets Simpliste** : Actuellement, les textures sont chargées directement dans les classes (`new Texture(...)`). Dans un projet de plus grande envergure, l'utilisation de l'`AssetManager` de LibGDX serait impérative pour gérer le chargement asynchrone et éviter les saccades lors des transitions.
42. **Physique "Maison"** : Nous avons implémenté notre propre système de collision (AABB). Bien que pédagogique, cela manque de précision pour les pentes ou la physique complexe. L'intégration de **Box2D** aurait offert plus de stabilité, bien qu'augmentant la complexité initiale.

43. **Couplage `GameScreen`** : La classe `GameScreen` tend à devenir une "God Class" gérant à la fois la logique, le rendu et l'UI. Une séparation plus stricte des couches de vue et de contrôleur serait bénéfique pour la maintenabilité à long terme.
44. **Absence de Menu d'Options Persistant** : Les configurations sont lues au démarrage mais ne peuvent pas être modifiées "in-game" via une UI dédiée, ce qui limite l'expérience utilisateur finale.

En conclusion, ce projet a été un excellent exercice d'application des patrons de conception, montrant qu'une bonne architecture en amont permet de gagner un temps précieux lors de l'ajout de contenu en aval.

Date de Soumission : 12 janvier 2026

Auteur(s) : Korotaiev Volodymyr, Martynchyk Oleksandr

Dépôt Git : https://github.com/fsdsfr/moteur-jeu-2d-libgdx.git

18. Annexes et Ressources

Documentation Technique Consultée

Pour la réalisation de ce moteur, nous nous sommes appuyés sur la documentation officielle et plusieurs ressources clés :

45. **LibGDX Wiki Officiel** : Référence principale pour le cycle de vie de l'application et la gestion des `Screen`.
    - *Lien : libgdx.com/wiki/*
46. **Tiled Map Editor Documentation** : Crucial pour comprendre le format `.tmx` et l'utilisation des objets personnalisés pour les collisions.
    - *Lien : doc.mapeditor.org*
47. **Game Programming Patterns (Robert Nystrom)** : Livre de référence pour l'implémentation propre du pattern *State* et du *Game Loop*.
    - *Lien : gameprogrammingpatterns.com*

Crédits & Assets (Utilisation Éducative)

Dans le cadre de ce projet universitaire (non-commercial), nous avons utilisé des ressources graphiques et audio provenant de sources externes :

- **Assets graphiques** : une partie des sprites/textures provient des ressources fournies dans un **cours Udemy** suivi par l'équipe. Ces ressources ont été intégrées uniquement à des fins d'apprentissage et de démonstration du moteur.
- **Audio (SFX / Musique)** : nous avons également utilisé des **assets gratuits (libres de droits ou sous licences ouvertes)** (effets sonores et musiques) provenant de banques/collections libres d'accès.

Toutes les ressources restent la propriété de leurs auteurs respectifs. Le projet est réalisé strictement dans un objectif pédagogique et n'est pas destiné à une distribution commerciale.

Les licences spécifiques (lorsqu'elles existent) sont respectées conformément aux conditions des plateformes sources.

Outils Utilisés

- **Visual Studio Code** : Éditeur de code principal (avec extensions Java & Gradle).
- **Tiled 1.10+** : Création de niveaux et gestion des calques.
- **TexturePacker** : Optimisation des sprites (si utilisé).