Project Report

Development of a 2D Game Engine with LibGDX

Project Realisation

- **BELHOUT Oussama**

1 Introduction

This project presents the development of a framework for building Role-Playing Games (RPGs) using **LibGDX**, a cross-platform Java game development framework, and **Tiled**, a powerful tool for designing and configuring 2D maps. The framework is designed to serve as a robust foundation for creating RPG games, enabling developers to work efficiently with tiled maps and implement game logic with ease.

As part of this learning project, the opportunity was taken to apply widely recognized **object-oriented design principles** and **design patterns**. This not only ensured a sound structural design but also facilitated a deeper understanding of best practices in software engineering.

A key focus of the project was to create a system that is highly **extendable** and **maintainable**. These characteristics were prioritized to allow the framework to support future enhancements and adapt to a variety of RPG game development scenarios with minimal modification to the core codebase.

To demonstrate the framework, a simple implementation example was provided, serving as a template for understanding and extending the system. This example highlights the framework's practical application and its potential to streamline RPG game development.

2 Project Presentation

2.1 Technologies and Tools Used

The development of this framework utilized several technologies and tools that facilitated the creation of a robust and efficient 2D RPG engine. These include:

- **LibGDX**: A modern and versatile Java library for game development, providing comprehensive support for designing game interfaces and managing graphical assets. Its cross-platform capabilities and modular design make it a popular choice for game developers.

- **Tiled Map Editor**: A tile-based map editor that simplifies the process of building 2D maps by abstracting the complexity involved. It enables designers to create and configure maps with ease, making the design process significantly more intuitive and efficient.
- **Gradle**: A powerful build automation tool that streamlines project configuration, dependency management, and builds. Gradle's flexibility and compatibility with the LibGDX ecosystem made it an ideal choice for managing the project's structure and dependencies.
- **Visual Studio Code (VS Code)**: A lightweight yet feature-rich source code editor that supports multiple programming languages, including Java. Its integrated tools and extensions enhanced productivity during the development process.

It is worth noting that the deliberate avoidance of extensive use of external libraries or sub-libraries, such as `Box2D`, was a key decision in this project. This approach allowed for a stronger emphasis on applying object-oriented analysis and design principles, ensuring that the focus remained on creating a robust and extendable framework rather than relying heavily on pre-built solutions.

2.2 Implemented Features

As this system is designed to function as a framework, its primary goal is to provide a highly **customizable** foundation for developing Role-Playing Games (RPGs). The following key aspects of the framework are customizable, allowing developers to adapt the system to their specific needs:

- **Player Skin**: The framework supports dynamic customization of the player's visual representation. Animations or textures for various player states, such as *standing*, *jumping*, and *walking*, can be easily replaced. These customizations are subject to specific constraints, primarily naming conventions, which ensure consistency and proper integration with the system.
- **Player Behavior**: Using a well-defined state-based design pattern, developers can extend the player's behavior lifecycle by adding new states seamlessly. This flexibility allows for diverse gameplay mechanics and enriched player interactions.
- **Map Management**: Maps can be dynamically loaded into the game by simply placing the desired map file in the `assets` directory and naming the current map as `mc.tmx`. This approach minimizes manual configuration and ensures that new maps can be integrated with minimal effort.

Additionally, the framework allows for extensive customization of the game environment, including player graphics, sound effects, and interactions between objects. These capabilities significantly broaden the possibilities for RPG game development, enabling the creation of diverse game worlds and unique gameplay experiences. The system's design prioritizes ease of extension and adaptability, making it a versatile tool for developers exploring various RPG concepts and ideas.

2.3 Configuration and Adding Content with Tiled

The framework is designed to integrate seamlessly with the **Tiled Map Editor**, enabling straightforward configuration and addition of game content. This section describes the process for configuring animations and adding new content, specifically for the main entity, the player.

2.3.1 Animations and skining

For each state implemented for the player entity, the corresponding animation resources must follow a structured naming and directory convention. Each state should have its own dedicated directory within the player's main directory, named in lowercase to ensure consistency. Within the state's directory, a sequence of image files representing the animation frames for that state should be placed, named in an ordered numerical sequence. For example, consider the *walking* state of a player named `steve`. The directory structure should resemble the following:

```
assets/
 steve/
     walking/
         1.png
         2.png
         3.png
         ...
```

In this configuration:

- The directory `walking` contains the animation frames for the *walking* state of `steve`.
- Each frame is represented by a sequentially numbered image file, e.g., `1.png`, `2.png`, `3.png`, etc.
- Additional states, such as *jumping* or *standing*, should follow a similar pattern, with separate directories and appropriately ordered frames.

This structured approach simplifies the process of adding animations and ensures that the system can dynamically load and interpret the resources during runtime. By adhering to these conventions, developers can efficiently extend the visual representation of the player and maintain consistency across different states.

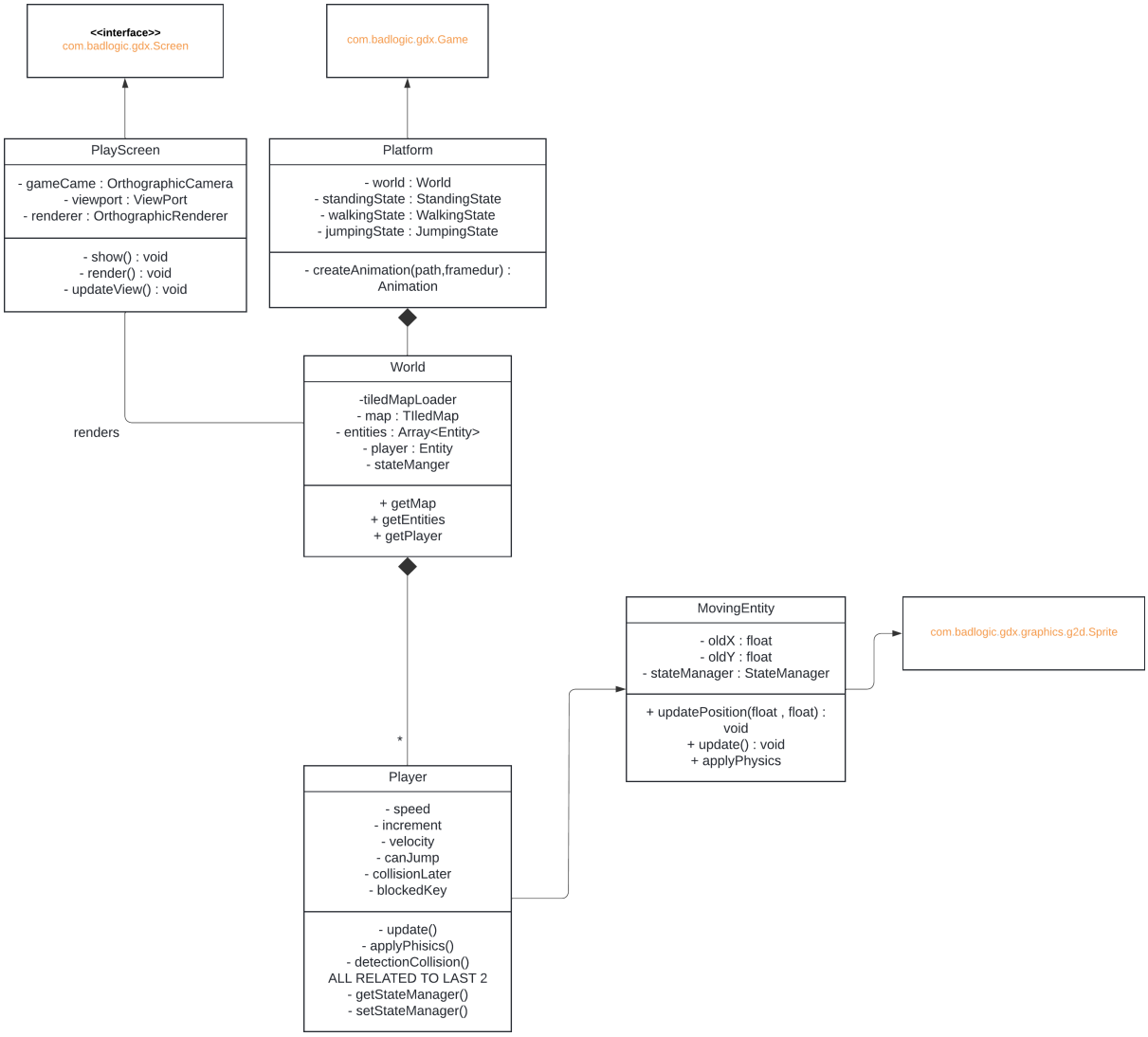
2.3.2 Initialization configurations

The `Platform` class, which extends the `Game` class from the **LibGDX** framework, serves as the primary initialization class for all game components and logic. This class is responsible for setting up the game environment and orchestrating the necessary configurations to ensure smooth gameplay.

Within the `create()` method of the `Platform` class, the following tasks are performed:

- Initialization of all player states, ensuring that each state is properly loaded and ready for use.

- Creation and loading of animation and sound objects, then binding them to their corresponding states.
- Setup of the game world's logic, including interactions between game objects.
- Setting up game screen.



This centralized configuration process simplifies the management of game components during the initial stages of development. However, it is recognized that this approach may become less efficient as the complexity of the game increases.

Future Enhancements: To address potential scalability issues, better approaches for configurations, such as the use of dependency injection or configuration files (e.g., JSON or XML), are proposed in the conclusion section of this document. These methods would allow for a more modular and flexible configuration process, reducing the dependency on hardcoded initialization logic within the `Platform` class.

2.4 Compilation and Execution

Requirements: Java 11+, LibGDX SDK (version 1.13.1).

Compilation Steps:

1. Clone the GitHub repository using the following command:
   `git clone https://github.com/Oussama-belhout/RPG.git`
2. Navigate to the project directory.
3. Run the `Lwjgl3Launcher` class located at:
   `Minecraft/lwjgl3/src/main/java/com/supermario/lwjgl3/Lwjgl3Launcher.java`.

Link to the repository: `https://github.com/Oussama-belhout/RPG.git`

3 General Architecture of the Game Engine

The architecture of the game engine is structured into multiple modules, which are organized into a core directory and a template instance located in a directory called `thegame`. This modular approach enables easy customization and extension of the game framework. The template instance within `thegame` implements all three core modules, providing a practical example of how to use the framework.

Since the framework follows a modular design, the core components are divided into three main modules:

- `behaviours`
- `models`
- `thegame` (Template instance)

The framework adheres to the **Model-View-Controller (MVC)** architectural design pattern, where the `thegame` directory contains a template that demonstrates the application of the three modules.

Two important modules, `behaviours` and `models`, are responsible for modeling dynamic and static elements, respectively. Let us examine them in more detail.

3.1 Behaviours Modeling

The state-machine for modeling entity behaviors can be defined as:

$$A = (\Sigma, Q, \delta, q_0)$$

Where:

- Σ is the alphabet, defined as the set of all objects that have `Event` as a superclass:

$$\Sigma = \{\text{for every object that has } \texttt{Event} \text{ as a super class}\}$$

- Q is the set of states, defined as the set of all objects that have `State` as a superclass:

$$Q = \{\text{for every object that has } \texttt{State} \text{ as a super class}\}$$

- δ is the set of transitions, which are defined in every state for each received event:

$$\delta = \{\text{transitions defined in every state for each received event}\}$$

- q_0 is the initialization state, defined as the `StandingState`:

$$q_0 = \texttt{StandingState}$$

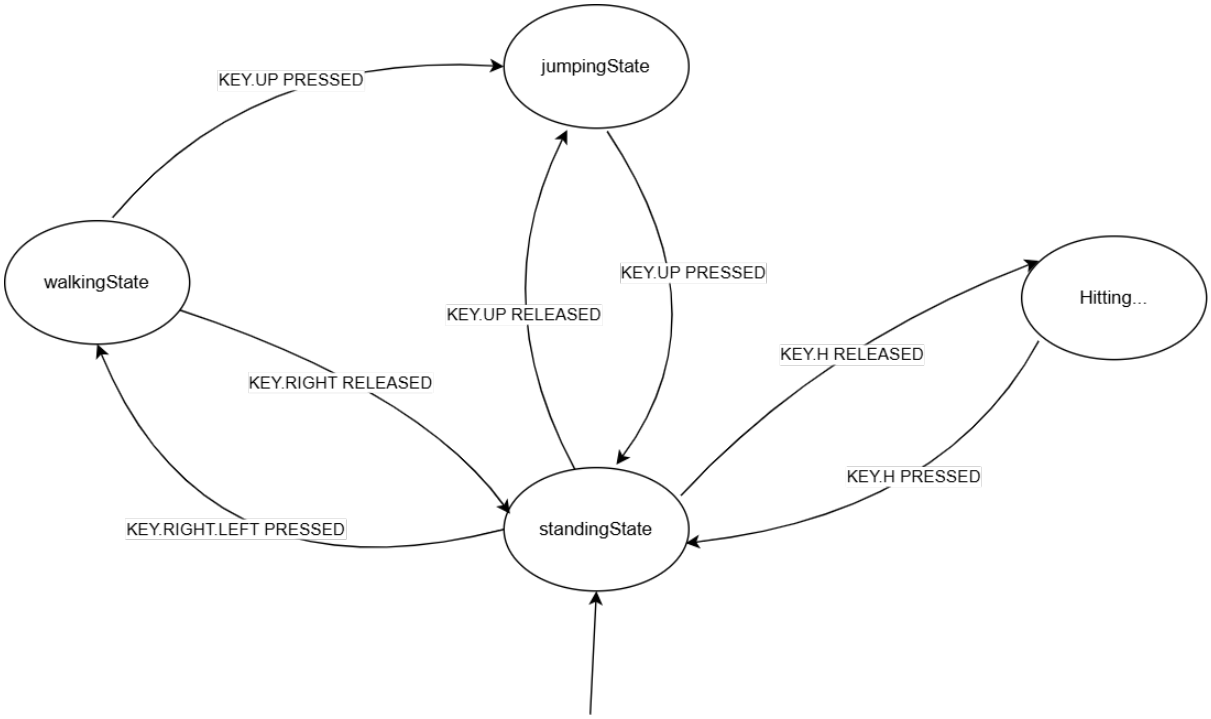


Figure 1: State Transition Diagram for Entity Behaviours

3.2 Behaviour Implementation

The entity's behaviors are primarily modeled based on three design patterns: **State**, **Observer**, and **Mediator**. These patterns help manage the complexity of dynamic interactions and transitions between different behaviors.

3.2.1 Event Class

The `Event` class is an abstract class that enforces the definition of the `equals()` method. This method is crucial for determining if an event corresponds to a transition event from the current state. It provides a uniform interface for comparing events and handling state transitions.

3.2.2 KeyEvent Class

The `KeyEvent` class extends the `Event` class and adds the concept of key input events. It contains:

- A key code (e.g., `KeyEvent.KEY_G`) representing a specific key press.
- A state indicating whether the key was pressed or released.

These two pieces of information provide a complete reference for comparing key events using the `equals()` method. This allows for efficient detection of key event transitions.

Users can create custom events by extending the `Event` class. For example:

- `ExposureToSunlightEvent` for handling exposure to sunlight.

- `CoinAcquisitionEvent` for tracking coin collection.

These custom events are placed in the `behaviours.events` package.

3.2.3 StateManager Class

The `StateManager` class is a singleton that functions as the manager of the entity's states and transitions. It acts as a **Mediator**, providing a centralized point of access to the entity's objects, ensuring that state transitions occur smoothly and that the correct behavior is executed.

The `StateManager` ensures that the appropriate behavior is triggered when an event occurs, helping to maintain consistency across the game's logic and interactions.

- A reference to the current state and the previous state, ensuring traceability for specific states.
- Embedded physical properties to represent the entity's state effectively.

Its primary responsibilities include:

- Managing state transitions based on events.
- Updating the player entity, including physics, rewards, and other dynamic attributes.

3.2.4 State Class

The `State` class is an abstract representation of an entity's behavior, characterized by:

- **Animation:** A sequence of frames defining the visual representation of the state.
- **Sound:** An optional auditory component associated with the state.
- **Transition Table:** A mapping of events to the next possible states.

This class interacts with the `StateManager` to either:

- Act as the current state of the entity.
- Fetch the next state from its transition table based on received events.

Classes extending the `State` class must implement the following methods:

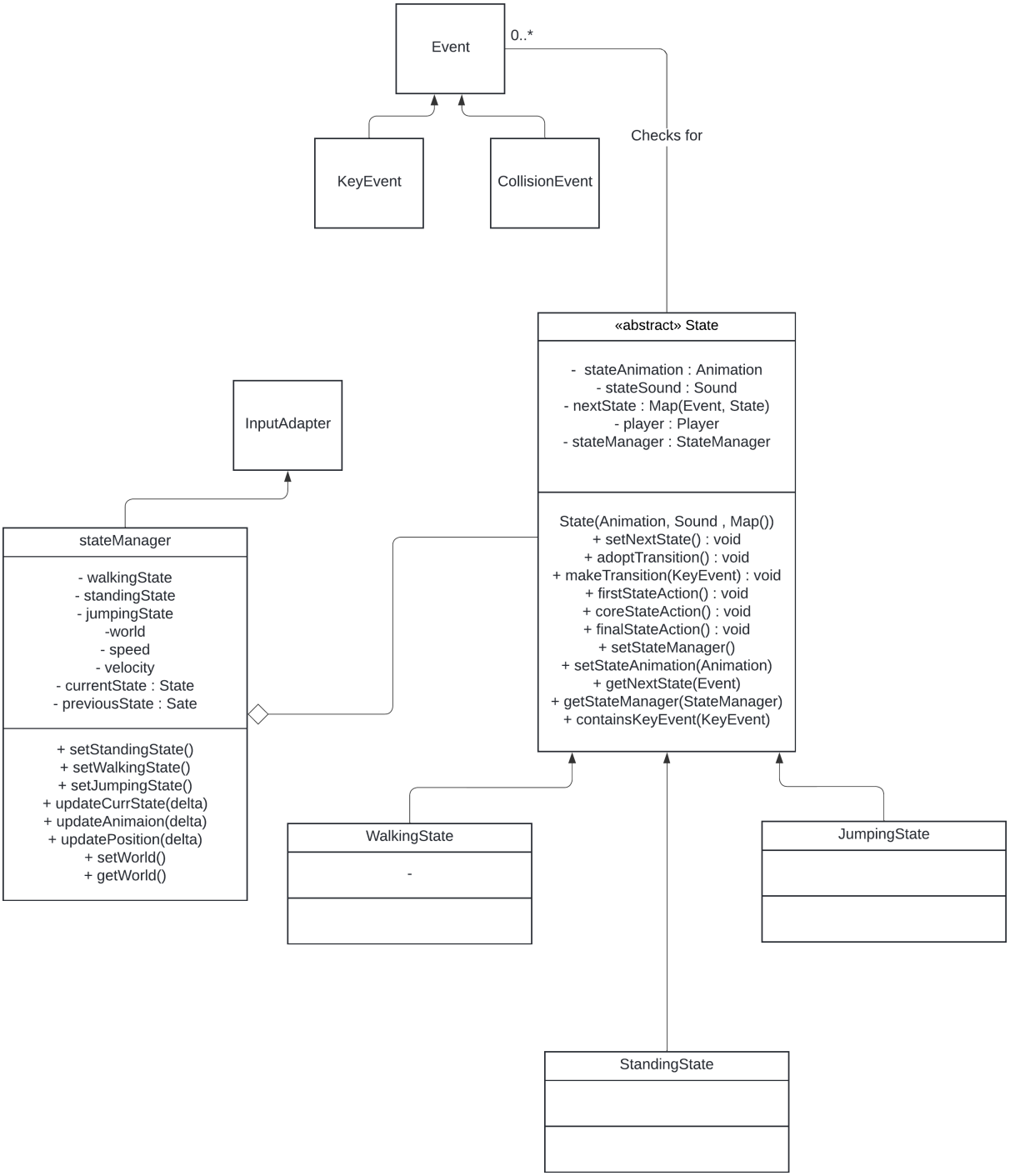
- `firstStateAction()`: Defines the actions to perform when the state is adopted. For instance, initiating player movement when transitioning to the *Walking* state.
- `coreStateAction()`: Specifies ongoing changes to the entity during the state. This method is invoked within LibGDX's event loop. For example, setting the next frame in the animation sequence.
- `finalStateAction()`: Handles cleanup and finalization when transitioning out of the state. For instance, resetting the velocity to zero when exiting a moving state.

3.3 Event Loop and Entity State Management

The interaction of the core components can be traced through the game engine's event loop, starting from the `MainScreen` object. The process unfolds as follows:

1. The `MainScreen`'s `render()` method is invoked, which iterates through all entities in the game world.
2. For each entity, the `entity.draw()` method is called to handle updates and rendering.
3. Within the `entity.draw()` method, the entity delegates state-related updates to its `StateManager` by invoking the `StateManager.updateCurrentState()` method.
4. The `StateManager`, in turn:
    - Manages state transitions based on triggered events.
    - Executes the appropriate state logic using the methods defined in the current state (`coreStateAction()`, `firstStateAction()`, or `finalStateAction()` as necessary).
    - Updates other entity properties, such as physics and awards, ensuring synchronization with the game world.

This design emphasizes modularity and delegation, ensuring that each component of the framework—entities, states, and the state manager—operates independently while seamlessly interacting within the broader system.



3.4 Interaction Modeling

Entities in the game engine are categorized into two primary types:

- **Movable Entities:** These entities possess multiple states managed by a `StateManager`.
- **Nature Entities:** These are typically represented by `TiledMapTileLayer.Cell` objects, which hold a set of properties.

3.4.1 NatureEntity Class

The `NatureEntity` class is an abstract class extending `TiledMapTileLayer.Cell`. It serves as the base for all static entities and includes a factory method that maps cell properties to their corresponding concrete class. For example, a cell with the property `type = solid` would be mapped to a `Soil` object.

3.4.2 Concrete NatureEntity Classes

Concrete classes derived from `NatureEntity` define the specific types of static entities used in map design. Each tile is associated with a unique `NatureEntity` subclass, such as `Soil` or `Meteor`, ensuring that every tile corresponds to a predefined type.

3.4.3 Interactions and Behaviors

To model interactions:

- **Shared Behaviors:** Nature entities exhibit common behaviors, such as blocking movement (`Blocking`) or causing damage (`Sanctioning`).
- **Impactful Nature Entities:** For tiles with impactful properties (e.g., fire or solid obstacles), the corresponding `NatureEntity` subclasses implement specific interfaces (`Blockable`, `Sanctioning`, etc.).
- **Decorator Pattern:** Interfaces provide default methods to define generalized impacts. For example, a `Meteor` would implement both `Blocking` and `Sanctioning` interfaces.
- **InteractableEntity Interface:** All impactful entities implement the `InteractableEntity` interface, which defines the `impact()` method to consolidate the behaviors of all implemented interfaces into a single point of interaction.

This design ensures modularity, extensibility, and clear mapping between tile properties and their in-game effects, supporting dynamic and diverse interactions within the game world.

3.5 EntityBubble Object

The `EntityBubble` is a key wrapper object designed as the primary interface between movable entities and their surrounding environment (the map). It ensures seamless interaction and impact management between the movable entity and the cells within its vicinity.

3.5.1 Structure and Components

The `EntityBubble` holds references to five objects:

- Each object's type corresponds to the type of cell in a specific direction relative to the movable entity:
  - **Right Cell:** The cell immediately to the right of the entity.
  - **Left Cell:** The cell immediately to the left of the entity.

- **Up Cell:** The cell directly above the entity.
- **Down Cell:** The cell directly below the entity.
- **Center Cell:** The cell the entity currently occupies.

3.5.2 Coupling and Update Mechanism

- Each `EntityBubble` is tightly coupled to a `StateManager`, which itself is linked to a `MovableEntity`.
- During each frame, the `EntityBubble` updates the references to the surrounding objects (representing the cells) based on the entity's current position.

3.5.3 Interaction and Notifications

After updating its surroundings, the `EntityBubble` notifies the surrounding cells (if impactful) to invoke their respective `impact()` methods. These methods determine how the surrounding cells affect the movable entity based on their properties and implemented behaviors. Examples of impacts include:

- `Blocking:` Preventing the entity's movement.
- `Sanctioning:` Reducing the entity's health.
- `Awarding:` Granting bonuses or collectibles.

This dynamic and modular design allows for efficient interaction handling, ensuring that the movable entity responds appropriately to its environment in real time.

3.6 Using and Extending the Library

The game engine library has been designed to offer a modular, extensible, and developer-friendly framework for creating 2D games with ease. By adhering to object-oriented principles and established design patterns, the library facilitates seamless integration of custom features while maintaining a clear and manageable codebase. Below, we outline the steps to use and extend the library effectively.

3.7 Using the Library

3.7.1 Project Setup

- Begin by setting up a new project using the provided core directory as the foundation.
- Include the required dependencies such as `libGDX`, `Gradle`, and the library's modules.

3.7.2 Creating a Game Instance

- Instantiate the `Platform` class and define the desired configurations within its `create()` method.
- Use the provided template instance (`thegame`) as a starting point for rapid prototyping.

3.7.3 Building the Game World

- Design maps using the `Tiled` editor and place the resulting files in the `assets` directory.
- Define game entities such as players, enemies, and interactable objects, ensuring they are properly integrated with the `EntityBubble` and `StateManager`.

3.8 Extending the Library

3.8.1 Customizing Entities

- Extend the `MovableEntity` class to create new player or NPC types.
- Implement custom states by extending the `State` class and defining `firstStateAction()`, `coreStateAction()`, and `finalStateAction()` methods.

3.8.2 Enhancing Interactions

- Create new `NatureEntity` subclasses to represent additional map elements.
- Implement interfaces such as `Blocking`, `Sanctioning`, and `Awarding` to define new types of impacts on movable entities.

3.8.3 Extending Behaviors

- Add new event types by extending the `Event` class, such as environmental or item-based triggers.
- Update the `StateManager` to accommodate additional transitions and logic.

3.8.4 Custom Input Handling

- Modify the `StateManager` class to handle unique input scenarios if required.
- For advanced setups, create additional input processors and integrate them with the main game loop.

3.9 Proposed Enhancements

Although the current design provides a robust foundation, future iterations can include:

- External configuration files for defining states, events, and transitions, reducing dependency on hardcoded logic.
- Advanced physics engines like `Box2D` for more realistic simulations.
- Plugin support for integrating new game mechanics or external tools.

By combining its modular architecture with these extensibility features, the library empowers developers to create a wide range of 2D games while providing the flexibility to meet project-specific needs.

Project Summary

This project introduces a modular and extensible 2D game engine framework designed to streamline the development of games built on the **libGDX** platform. The framework adheres to the **MVC (Model-View-Controller)** architectural pattern and leverages key software design patterns such as **State**, **Observer**, **Mediator**, and **Decorator**, ensuring a robust and maintainable structure.

Key Features

1. **State-Based Architecture:**
    - Each entity behavior is modeled using a finite state machine.
    - States are implemented with an abstract `State` class, which encapsulates animations, sounds, and transition logic.
2. **Dynamic and Static Modeling:**
    - Dynamic behaviors are managed through the `StateManager`, which processes inputs, transitions between states, and manages physics and entity updates.
    - Static properties of map elements (`NatureEntity`) are modeled using a factory-based approach, enabling interaction-specific behaviors.
3. **Entity Interactions:**
    - Interactions between movable entities and the environment are facilitated by the `EntityBubble`, which tracks surrounding tiles and delegates impact logic.
    - The impact system is enhanced by interfaces such as `Blocking`, `Sanctioning`, and `Awarding`, allowing for flexible and reusable interaction definitions.
4. **Map Integration:**
    - Maps are designed using the **Tiled** editor, enabling developers to visually layout game worlds.
    - Each tile is mapped to a specific `NatureEntity` type, defining its behavior and interactions.
5. **Customizable Framework:**
    - Developers can extend the framework by adding new states, events, and interactions, adapting it to a wide range of game genres and mechanics.

Usage Scenarios

- **Adventure Games:** Implement complex player and NPC interactions with the state system.
- **Platformers:** Use the physics integration to handle movements, collisions, and environmental effects.
- **Puzzle Games:** Leverage the decorator pattern for interactive tiles with unique properties (e.g., buttons, traps, or rewards).

Future Directions

- Introducing external configuration files for defining states, events, and transitions dynamically.
- Enhanced physics using **Box2D** for advanced mechanics like object rotation and momentum.
- Support for multiplayer systems with synchronized state management across players.

This project offers a powerful starting point for developers aiming to create engaging 2D games, combining ease of use with flexibility and scalability.