RELAZIONE PROGETTO “JBUBBLEBOBBLE”

Matricola: 1933858
Corso MZ-presenza

Asia Mazzotta

Introduzione

Il progetto **JBubbleBobble** segue il pattern **MVC (Model-View-Controller)** ed è suddiviso in tre principali pacchetti: **Model**, **Controller**, e **View**. Questo approccio facilita la separazione delle responsabilità e consente una manutenzione e un'estendibilità più efficienti.

1. Package Model

Il package **Model** contiene tutte le classi che rappresentano gli elementi logici e le entità del gioco.

- **GameObject**: Classe astratta che rappresenta un oggetto nel gioco con coordinate e hitbox. Ad esempio, la classe `Wall` rappresenta un muro fisico.
- **MovableObject**: Estende `GameObject` e rappresenta oggetti che possono muoversi. Aggiunge attributi per il controllo del movimento, direzione e velocità. Dichiara i metodi astratti `updatePosition()` e `applyGravity()`. Implementa inoltre l'interfaccia **Collidable** per la gestione delle collisioni. Esempi di classi derivate includono:
  - **Bubble**: Oggetto con cui il giocatore attacca.
  - **Boulder**: Oggetto con cui il nemico attacca.

  Per entrambi, `applyGravity()` è vuoto perché non soggetti a gravità, e `updatePosition()` ha un'implementazione specifica.

L'implementazione del metodo `updatePosition()` prevede il controllo della direzione in cui l'oggetto desidera muoversi creando una nuova hitbox temporanea. Successivamente, viene verificata la presenza di eventuali collisioni esaminando la lista di tutti gli oggetti presenti nell'ambiente con tale hitbox. Se non vengono rilevate collisioni, l'oggetto può muoversi liberamente verso la nuova posizione. In caso contrario, viene riposizionato su coordinate specifiche, determinate in base alla situazione e al tipo di collisione riscontrata.

- **Entity**: Estende `MovableObject`, rappresentando oggetti che possono attaccare e saltare. Implementa l'interfaccia **Observable** e fornisce un'implementazione generica di `applyGravity()`. Estende a classi come:
  - **Enemy**: Rappresenta un nemico generico con stati e metodi per il movimento e il comportamento. Sono implementati tre nemici con logiche diverse:
    - **ZenChan**: Si muove verso il giocatore.
    - **Mighta**: Attacca solo se il giocatore entra in un certo range.
    - **Blubba**: Si muove in tutte le direzioni senza gravità né capacità di saltare.
  - **Player**: Gestisce il numero di vite del giocatore, il movimento e i comportamenti in caso di colpi subiti. Gestisce una lista di **PlayerObserver** per la notifica di eventi legati al giocatore.
  - **PowerUp (!!)**: Contiene un tipo e dei punti. Implementa una versione personalizzata di `applyGravity()` e del metodo `collision()` per l'interazione con il giocatore. Include una **nested class PowerUpFactory** per creare oggetti `PowerUp`, anche randomicamente.

L'implementazione del metodo `applyGravity()` segue una logica simile a quella di `updatePosition()`, con l'aggiunta di un controllo per verificare se l'oggetto deve saltare o sta già saltando.

Per la classe `Player`, il movimento è determinato direttamente dall'input del giocatore, tramite metodi specifici invocati dal `keyHandler` (attraverso il GameController) quando un tasto viene premuto. Al

contrario, per le altre entità, come i nemici, è stato implementato un metodo `move()`, che decide casualmente come muovere l'oggetto, utilizzando un numero generato in modo randomico.

- **User e UserDatabase**: Rappresentano un giocatore fisico e le sue informazioni, con metodi per salvare i dati su file. L'uso di **Stream** ottimizza la ricerca di utenti.
- **Utility**: Contiene valori statici e un metodo per determinare posizioni libere nei livelli (utilizzo di Stream).
- **CollisionDetection**: Fornisce i metodi per il rilevamento delle collisioni e gestisce gli oggetti **Collidable**.
- **LevelComponent e LevelMap**: Rappresentano rispettivamente le componenti di un livello (come muri e nemici) e la mappa del livello, indicando quali file leggere e come creare gli oggetti in base ai caratteri.

2. Package Controller

Il package **Controller** gestisce la logica di controllo e le funzionalità tecniche del gioco.

- **GameController**: Gestisce il flusso di gioco, gli stati, i livelli, e l'aggiornamento delle componenti. Utilizza il pattern **Singleton** per garantire un'unica istanza durante l'esecuzione. Implementa l'interfaccia **Runnable** per il loop di gioco e **PlayerObserver** per monitorare eventi legati al giocatore, come il "Game Over". Si occupa anche di gestire l'audio attraverso l'**AudioManager** e collega il **Player** alla vista tramite **PlayerView** (in generale si occupa di passare le componenti alla GUI). Inoltre si occupa di caricare i valori del database che stanno nel file userdatabase.txt situato nella directory res e specifica tutti i metodi per lavorarci.

Nel ciclo principale del `GameController` viene invocato il metodo `update()`. Se il gioco si trova nello stato `RUNNING`, tutte le componenti istanziate vengono aggiornate, utilizzando i metodi `move()` o `updatePosition()` a seconda del caso. Se le entità stanno eseguendo azioni specifiche, il controller reagisce di conseguenza: ad

esempio, se il giocatore sta attaccando, il controller crea una nuova istanza di un oggetto `Bubble`. Nel caso in cui tutti i nemici del livello siano stati sconfitti, viene chiamato il metodo `changeLevel()`, che aggiorna tutte le componenti al livello successivo, se disponibile. In caso contrario, viene segnalata la vittoria e i risultati vengono salvati nel database.

- **AudioManager**: Classe Singleton che gestisce l'audio del gioco caricando file dalla directory `res/Sound`.
- **KeyHandler**: Implementa l'interfaccia **KeyListener** e segue il pattern Singleton, assicurando che ci sia un unico gestore degli input da tastiera. Comunica con il `GameController` per gestire gli stati del gioco in base agli input.
- **JBubbleBobble**: Classe principale che contiene il metodo `main` del gioco.

3. Package View

Il package **View** contiene tutti gli elementi grafici e le componenti dell'interfaccia utente.

- **MainFrame:** Classe principale per la gestione dell'interfaccia grafica, che utilizza un CardLayout per gestire i vari pannelli. Contiene un container principale e metodi per aggiungere e recuperare pannelli specifici tramite i loro nomi.
- **Base Panel :** Classe astratta che estende `JPanel`. Fornisce le dimensioni standard per i pannelli, definite nella classe Utility. Questa classe viene ereditata da tutti i pannelli che rappresentano le diverse schermate del gioco:
    - **StartPanel:** Pannello iniziale che visualizza un'icona e del testo di benvenuto. Funziona come introduzione al gioco.
    - **MenuPanel:** Pannello del menu principale, che contiene diverse opzioni come `JLabel` per ogni voce del menu.

Fornisce metodi per selezionare e muovere la selezione tra le opzioni del menu.
    - **RegisterPanel:** Pannello per la registrazione dei giocatori. Contiene delle `JLabel` che mostrano immagini che un giocatore può scegliere come icona personale, insieme a una barra di input per inserire il nickname.
    - **ScrollLogin:** Classe che estende `JScrollPane`, contiene un `JPanel` che funge da container per visualizzare la lista degli utenti fornita dal GameController. Le informazioni degli utenti vengono "disegnate" dinamicamente in base ai dati passati.
    - **PlayerView:** Classe che estende `JLabel` e implementa l'interfaccia **Observer**. Quando viene istanziata dal GameController, viene associata al giocatore per monitorarne le azioni. Contiene liste che gestiscono le animazioni del giocatore, e un `Timer` per aggiornare la visualizzazione e il repaint() delle animazioni.
    - **GamePanel:** Pannello principale del gioco, responsabile di visualizzare tutte le animazioni dei nemici, delle bolle e delle altre componenti di gioco. Includendo anche delle `JLabel` per visualizzare le informazioni correnti (punti, vite, ecc.), incorpora una classe interna chiamata **LevelPanel**, che è responsabile del rendering grafico delle animazioni del livello e della PlayerView.
    - **GameOverPanel** e **VictoryPanel**: Due pannelli separati che mostrano del testo informativo in caso di sconfitta o vittoria.
    - **RankPanel:** Estende `JScrollPane` e mostra la classifica degli utenti in base al loro punteggio più alto (highestScore). Utilizza Collection.sort() per ordinare la lista degli utenti in ordine decrescente.
- **GameFont:** Classe utilizzata per caricare il font del gioco dal file .ttf.
- **GameImage:** Enumerazione che memorizza come costanti tutte le immagini utilizzate nel progetto. Consente una gestione centralizzata delle risorse grafiche.