Relazione
"Slayin"

Biagioni Giacomo,
Mencaccini Mattia, Morbidelli Cristian,
Morri Lorenzo, Morsucci Federico

June 25, 2024

Contents

1 Analisi

L'analisi dei requisiti rappresenta un passaggio cruciale nello sviluppo del nostro clone semplificato del gioco "Slayin". In questa fase, vengono definite le funzionalità che l'applicazione deve offrire per soddisfare le aspettative degli utenti e per garantire un'esperienza di gioco coerente con l'originale. Ecco un elenco dettagliato dei requisiti funzionali e non funzionali.

Requisiti funzionali:

- **Schermate del Gioco**
  - **Schermata Principale**: Deve permettere al giocatore di avviare una nuova partita o uscire dall'applicazione.
  - **Ambiente di Gioco**: Deve mostrare il protagonista, i nemici, il punteggio attuale e altre informazioni rilevanti durante il gioco.
  - **Game Over**: Deve visualizzare il punteggio finale e offrire l'opzione di riprovare o tornare alla schermata principale.
- **Controlli del Protagonista**
  - Il protagonista deve poter muoversi a sinistra e a destra.
  - Il protagonista deve poter saltare (o attivare della abilità speciali al posto del salto) per evitare nemici e attacchi.
  - Il protagonista deve poter attaccare con la spada o altre armi.
- **Nemici**
  - Devono esserci due tipi di nemici basici, ognuno con pattern di movimento distinti.
  - I nemici devono apparire in ondate successive con difficoltà crescente.
  - Deve essere implementato un boss che appare ogni tot ondate.
- **Sistema di Punteggio e Combo**
  - Ogni nemico sconfitto deve incrementare il punteggio del giocatore.
  - Colpire nemici in sequenza senza subire danni deve aumentare un moltiplicatore di combo, incrementando ulteriormente il punteggio.

Requisiti non funzionali

- **Usabilità**
  - L'interfaccia deve essere intuitiva e facile da navigare.
  - I comandi del gioco devono essere reattivi e facili da apprendere.

- **Performance**
  - Il gioco deve mantenere un frame rate stabile per garantire una giocabilità fluida.
  - Il tempo di caricamento tra le schermate deve essere ridotto al minimo.
- **Compatibilità**
  - L'applicazione deve essere eseguibile su una varietà di dispositivi con specifiche tecniche ragionevoli.

Descrizione del Dominio Applicativo

Il gioco che intendiamo sviluppare è un platform game. Il gioco si sviluppa attraverso livelli successivi, caratterizzati da un aumento progressivo della difficoltà. Ecco le componenti principali del dominio applicativo:

- **Protagonista**
  - Il giocatore controlla un eroe che si muove, salta e attacca i nemici con le sue armi o attiva delle abilità speciali al posto del salto.
- **Nemici**
  - I nemici appaiono in ondate e si muovono secondo pattern predefiniti. Alcuni nemici potrebbero semplicemente avanzare verso il protagonista, mentre altri potrebbero avere movimenti più complessi.
  - Dopo un certo numero di ondate, il giocatore deve affrontare un boss, che rappresenta una sfida maggiore con pattern di attacco unici e più complessi.
- **Sistema di Punteggio**
  - Il gioco premia il giocatore con punti per ogni nemico sconfitto. Il sistema di punteggio è pensato per incentivare l'abbattimento di nemici in sequenza rapida, grazie a un sistema di combo che moltiplica i punti ottenuti.
- **Ambiente di Gioco**
  - L'ambiente di gioco è costituito da una singola scena in cui avvengono tutte le azioni. Questa scena deve essere progettata per favorire il movimento del protagonista e l'apparizione dei nemici.

Sfide Principali

- **Gestione delle Collisioni**: È necessario implementare un sistema di collisioni accurato tra il protagonista, la sua spada e i nemici. Questo è fondamentale per garantire che i colpi vengano registrati correttamente e che il gioco risponda in modo prevedibile agli input del giocatore.
- **Movimento e Comportamento dei Nemici**: I nemici devono seguire pattern di movimento specifici che variano in base al tipo di nemico e alla fase del gioco. Implementare questi pattern richiede un'attenta pianificazione per evitare movimenti innaturali o comportamenti imprevedibili.
- **Movimento e Comportamento del Character**: Il personaggio principale deve rispondere in modo fluido e preciso ai comandi del giocatore. È fondamentale implementare meccaniche di movimento che permettano al character di eseguire azioni come correre, saltare e attaccare con immediatezza. Il comportamento del character deve adattarsi dinamicamente alle situazioni di gioco, per assicurare che il gameplay sia coinvolgente e soddisfacente.

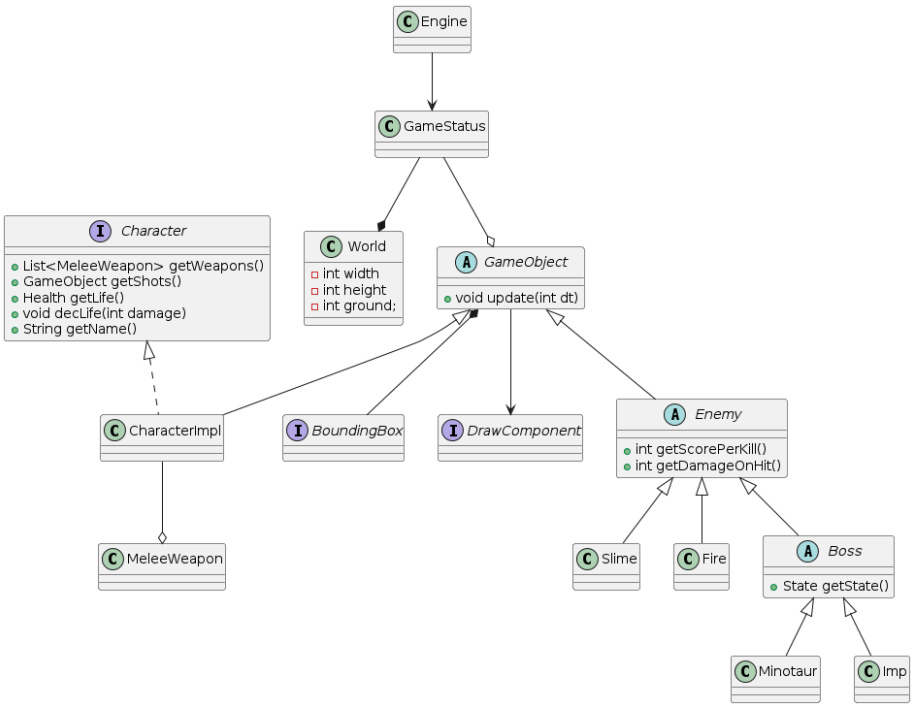


Figure 1: UML rappresentativo degli elementi del dominio

2 Design

2.1 Architettura

Nello specifico, il modello `MVC` è stato implementato attraverso le classi `Engine`, `SceneController`, `GameStatus` che rappresentano rispettivamente Controller, View e Model. L'Engine è il main core dell'applicazione, ovvero il cervello su cui gira il Game Loop; si occupa quindi di intercettare degli input dell'utente, del dialogo tra i Model, la risoluzione dei calcoli e il richiamare le funzioni di rendering grafico. Il GameStatus rappresenta lo stato del dominio di gioco, ovvero tutto ciò che concerne i modelli che si trovano nella scena corrente. Lo SceneController sarà invece la parte adibita alla gestione della view venendo coordinata quindi dall'engine.

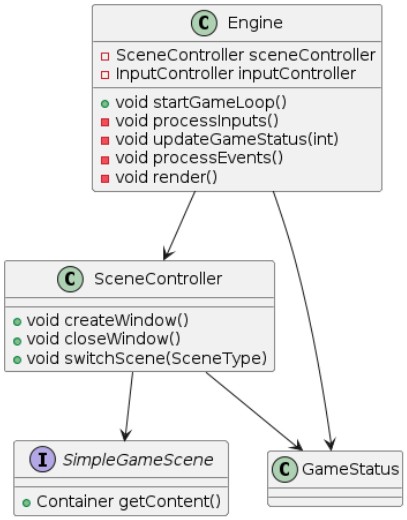


Figure 2: Architettura MVC

Cristian Morbidelli

Gestione del GameLoop

Descrizione del problema

Un problema fondamentale nello sviluppo del gioco è l'implementazione delle 3 fasi fondamentali del Loop di gioco: `processamento degli input`, `aggiornamento dello stato del gioco`, `rendering grafico dello stato attuale`. Queste 3 fasi devono susseguirsi in maniera ordinata ad ogni frame di gioco per permettere il corretto funzionamento senza incongruenze o ritardi visivi.

Descrizione della soluzione proposta

Il Loop di gioco è stato preso in carico dalla classe dell'Engine il cui compito principale sarà quindi quello di scandire i tempi delle diverse operazioni e richiamare quando necessario le varie operazioni dialogando con model e view.
Lo stato del gioco viene rappresentato tramite una singola classe `GameStatus` che si occupa di gestire in maniera ordinata tutto ciò che riguarda i modelli di gioco (protagonista, nemici, punteggi, etc...). Tale classe deve poter essere interrogata dall'engine e ne deve poter ricevere le direttive per aggiornarsi in maniera coerente e corretta.
L'engine rappresenterà quindi l'entry point per tutto ciò che riguarda le operazioni del controller quindi la risoluzione di determinate situazioni e la chiamata di determinate operazioni nei giusti momenti; lo Status è invece l'entry point per tutto ciò che concerne i singoli modelli con cui l'engine dovrà lavorare.
La gestione della view è spiegata più nel dettaglio in seguito.

Livelli

Descrizione del problema

La gestione dei livelli rappresentava un punto critico del modello del progetto. Avendo deciso di strutturare il gioco in un numero finito di livelli ben definiti, era necessario prevedere un sistema che definisse ed identificasse questi livelli. Prendendo come riferimento il gioco originale, ogni "livello" rappresenta un insieme di nemici che si dovranno affrontare nella scena, oppure boss speciali più ardui da sconfiggere. In caso di eliminazione di tutti i nemici all'interno di un livello, si passa a quello successivo.

Descrizione della soluzione proposta

Per rappresentare ogni livello ho deciso di costruire una classe `Level`, tenendo conto dei seguenti punti focali:

- **Cos'è un livello**: Un Level è, all'interno del progetto, un elenco dinamico degli Enemy che vengono gradualmente incontrati durante il livello. Ogni livello è identificato a livello di design da un identificatore numerico.

- **Gestione del dispiegamento**: La classe in sé non si preoccupa di come e quando i nemici che contiene nella sua lista saranno aggiunti alla scena di gioco. Tale compito è stato delegato all'engine che, durante lo scorrimento del GameLoop, deciderà quando estrarre nuovi elementi dalla lista del Level basandosi sul tempo passato e su quanti nemici si trovano già nella scena per evitare una congestione eccessiva.

- **Generazione di un livello**: Per generare ogni livello ho seguito il `Factory Method` costruendo una LevelFactory istanziata dall'Engine ed in grado di istanziare i singoli livelli. Per la lista dei nemici, utilizza a sua volta una EntityFactory che è la classe che si occupa di generare ciascun Enemy in relazione al mondo di gioco, attribuendo inoltre a ciascuno di essi un id univoco.

- **Caratterizzazione dei livelli**: Ogni singolo livello, a parità di identificatore, ha dei dati sempre costanti a prescindere dalla partita quali le categorie e le quantità dei nemici e la "capacità" massima di un livello. Per evitare di inserire questi dati in maniera hard-coded all'interno del gioco, la classe `LevelFactory` si occupa di volta in volta di reperire queste informazioni da un JSON leggendolo ed elaborandone i dati.

- **Fine dei livelli e del gioco**: Il passaggio da un livello al successivo viene incaricato all'Engine di gioco che intercetterà il momento in cui l'ultimo nemico di un livello verrà eliminato e richiederà subito la generazione del successivo. La Factory dei livelli continuerà a fornire gli oggetti Level; quando non sarà più in grado di leggere i dati di livelli successivi, fornirà all'engine un livello "vuoto" che fungerà per segnalare il raggiungimento del "Game Over".

Nell'implementazione finale, il gioco prevede 6 livelli differenti.

Schema UML

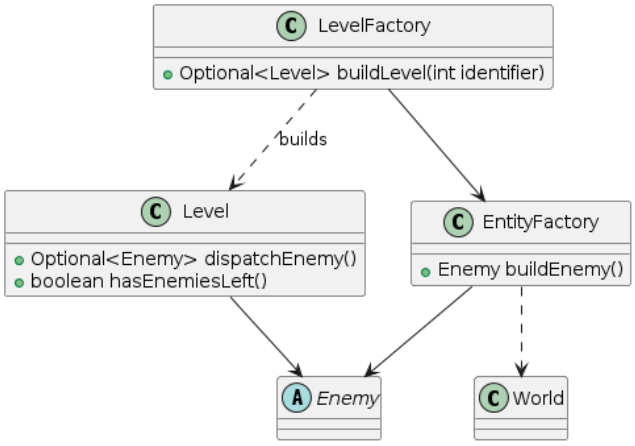
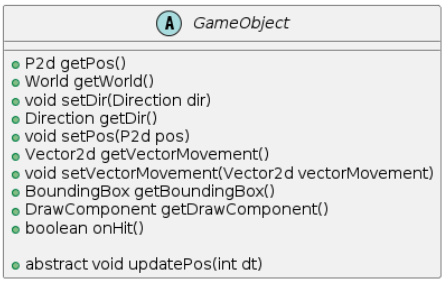


Figure 3: Schema UML del Design della creazione di Livelli

Federico Morsucci

Introduzione del GameObject



La classe **GameObject** è una classe astratta utilizzata come base per tutti gli altri oggetti di gioco. Per praticità abbiamo scritto questa classe abstract e non come interfaccia perchè abbiamo ritenuto che molti metodi altrimenti si sarebbero ripetuti inutilemente in giro per il progetto, l'unico metedo lasciato incopleto della classe è **updatePos(int dt)** che viene lasciato libero di essere completato in base all'implementazione specifica dell'oggetto, il parametro "dt" sarebbe il delta time cioè quanto tempo è passato dal ultimo tick di gioco(riferimento alla spiegazione dell'engine).
Abbiamo deciso che ogni **GameObject** deve avere:

- Una posizione che indica le cordinate centrali del personaggio
- Un vector per rappresentare la direzione e la velocità del movimento

Gestione dei personaggi

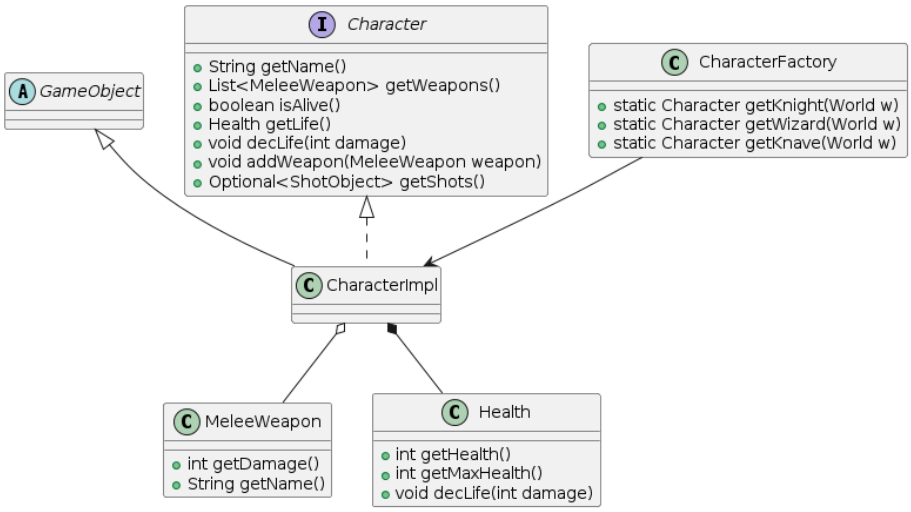
Descrizione del problema

Nel contesto di un gioco, è necessario gestire diversi tipi di personaggi, ognuno con le proprie caratteristiche e abilità. Ogni personaggio può avere armi, stato di salute, e deve interagire con il mondo di gioco (`World`). La gestione di questi aspetti deve essere flessibile e modulare, permettendo la creazione di nuovi tipi di personaggi e la loro evoluzione nel tempo. Il problema da risolvere è come organizzare il codice in modo da facilitare la creazione, gestione e aggiornamento dei personaggi nel gioco.

Descrizione della soluzione proposta

La soluzione proposta prevede l'uso di una classe `CharacterFactory` che funge da fabbrica per creare diversi tipi di personaggi (`Character`). Ogni personaggio ha attributi come nome, stato di salute (`Health`), e armi da mischia (`MeleeWeapon`). Le relazioni tra le classi sono rappresentate nel diagramma UML.questa scelta utilizza la `CharacterFactory` è ciò porto ad avere una maggiore modularità, facilitando l'aggiunta di nuovi tipi di personaggi e una Centralizzazione della logica di creazione dei personaggi, rendendo il codice più facile da modificare.

Schema UML



Pattern di progettazione utilizzati

Factory Method

La classe `CharacterFactory` rappresenta il Factory Method, fornendo metodi statici per creare diversi tipi di personaggi (`getKnight`, `getKnightModify`, `getWizard`, `getKnave`).

Gestione della rappresentazione grafica

Descrizione del Problema

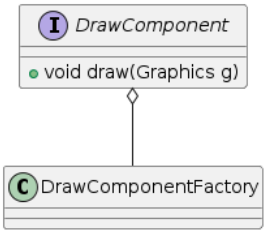
Il problema che si vuole risolvere riguarda la rappresentazione grafica degli oggetti di gioco (`GameObject`). La sfida è trovare un modo efficiente e flessibile per disegnare vari tipi di oggetti di gioco sullo schermo, come personaggi, nemici, ostacoli, ecc., in modo che ogni tipo di oggetto abbia il proprio modo di essere disegnato.

Descrizione della Soluzione

La soluzione proposta prevede l'uso di un'interfaccia `DrawComponent` che dichiara il metodo `void draw(Graphics g)`. Questo metodo sarà implementato da varie classi concrete che definiscono il modo specifico in cui ciascun tipo di `GameObject` viene disegnato.
Una fabbrica (`DrawComponentFactory`) è stata creata per generare le varie istanze di `DrawComponent` necessarie. I metodi statici della fabbrica (graphicsComponentCharacter, graphicsComponentSlime, ecc...) creano e restituiscono le istanze appropriate di `DrawComponent` per ogni tipo di oggetto. Ciò rende facile disegnare i `GameObject`. Inoltre la logica di disegno è separata dalla resto della logica.Un grosso contro di questa implementazione è Il metodo `draw` che prende come parametro `Graphics g`, il che crea una forte dipendenza dalla libreria grafica utilizzata. Se si decidesse di cambiare la libreria grafica, sarebbe necessario modificare l'interfaccia `DrawComponent`, il che rappresenta un grave errore di progettazione.

Schema UML

Di seguito è riportato uno schema UML che illustra la struttura della soluzione:



Pattern di progettazione utilizzati

- **Factory Method**

Gestione delle collisioni

Descrizione del Problema

Il problema da risolvere riguarda la gestione delle collisioni nel nostro gioco. È necessario implementare un sistema che rilevi efficacemente quando due oggetti di gioco si scontrano, evitando algoritmi complessi e pesanti come il SAT (Separating Axis Theorem), che risulterebbero superflui per le nostre esigenze.

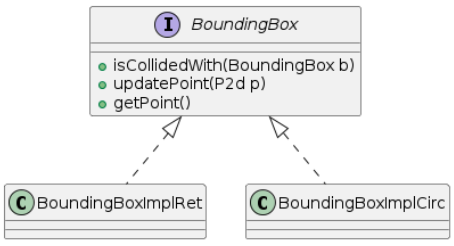
Descrizione della Soluzione

La soluzione proposta prevede l'uso di un'interfaccia `BoundingBox` che permette di aggiornare il punto di riferimento di un oggetto, ottenere il punto di riferimento attuale e determinare se due `BoundingBox` collidono.
Abbiamo considerato varie alternative per la gestione delle collisioni:

- **Uso di algoritmi complessi come SAT:** Sebbene accurati, questi algoritmi sono troppo complessi e computazionalmente costosi per le nostre esigenze.
- **Uso di un sistema di rilevamento delle collisioni basato su forme semplici:** Abbiamo optato per questa soluzione per garantire efficienza e semplicità, utilizzando rettangoli e cerchi come forme base.l'unico contro è che in alcuni casi, l'uso di rettangoli e cerchi può portare a piccole approssimazioni nella rilevazione delle collisioni

Schema UML

Il diagramma UML seguente rappresenta l'implementazione della gestione delle collisioni nel nostro gioco:



Uso di Pattern di Progettazione

La soluzione implementa il pattern **Strategy** per la gestione delle collisioni. L'interfaccia `BoundingBox` rappresenta la strategia, e le sue implementazioni concrete sono `BoundingBoxImplRet` e `BoundingBoxImplCirc`.

- **Strategy:**

- **Interfaccia della strategia:** `BoundingBox`
- **Implementazioni concrete:** `BoundingBoxImplRet`, `BoundingBoxImplCirc`

Implementazione delle Classi

BoundingBoxImplRet

Questa classe rappresenta un rettangolo e include proprietà come larghezza, altezza e punto di riferimento.

BoundingBoxImplCirc

Questa classe rappresenta un cerchio e include proprietà come raggio e centro.

Mattia Mencaccini

Gestione dei nemici

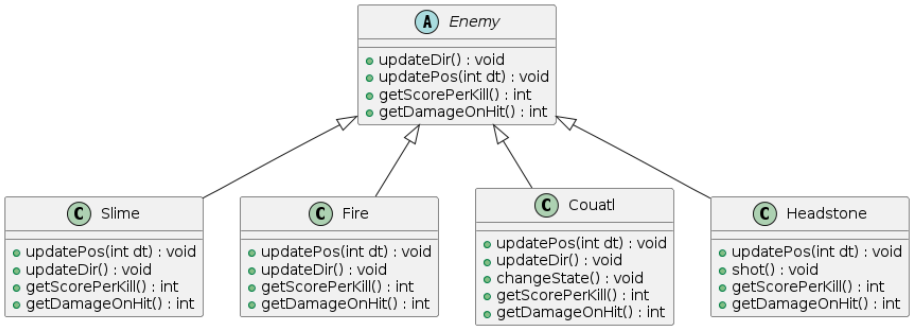
Descrizione del problema

Nel gioco *Slayin*, i giocatori devono affrontare vari nemici, ognuno con comportamenti e caratteristiche uniche. Gestire questi nemici richiede un sistema che permetta di definire comportamenti comuni e specifici in modo modulare ed estensibile. I nemici devono essere in grado di muoversi, interagire con il mondo di gioco, e reagire alle collisioni e agli eventi.

Descrizione della soluzione proposta

La soluzione prevede l'uso di una classe astratta `Enemy` che definisce il comportamento comune a tutti i nemici, e classi concrete che estendono `Enemy`, implementando comportamenti specifici. Questo modo di agire segue il pattern di programmazione *Template Method*: permette alle sottoclassi di ridefinire alcune fasi della superclasse senza modificarne la struttura complessiva. Usando questo pattern è facile creare nuovi tipi di nemici, evitando di riscrivere più volte lo stesso codice e ogni sottoclasse descrive un comportamento specifico, ma tutte le sottoclassi dipendono da `Enemy`.

Schema UML



Gestione dei proiettili dei nemici

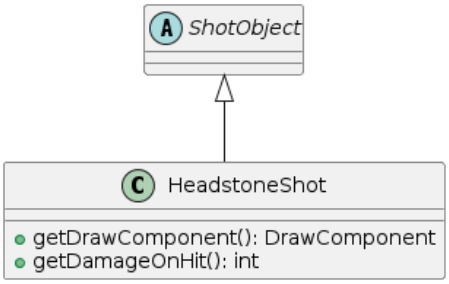
Descrizione del problema

Gestire i proiettili dei nemici

Descrizione della soluzione

La soluzione prevede l'uso di una classe astratta ShotObject che definisce il comportamento comune a tutti i proiettili, e una classe concreta HeadstoneShot che estende ShotObject e implementa comportamenti specifici. Anche qui il pattern di programmazione è Template Method.

Schema UML



Lorenzo Morri

Gestione Boss implementation e della BossBattle

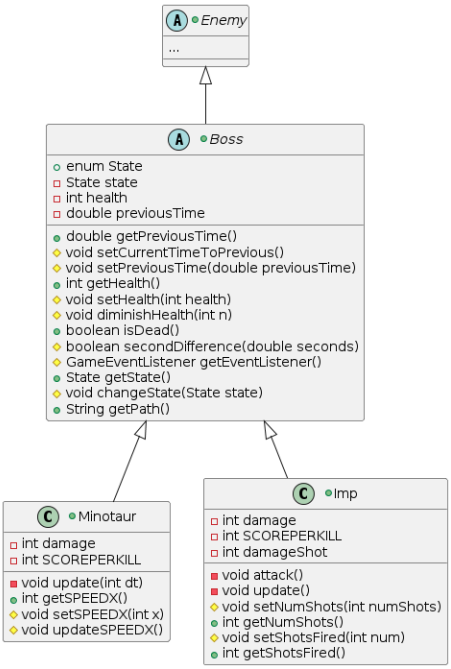
Descrizione del problema

Gestire i diversi comportamenti del Boss, come attacchi o movimenti speciali, e i vari momenti della boss battle.

Descrizione della soluzione proposta

Per gestire la logica e il movimento del Boss, si è adottato il pattern "Template Method". Questo approccio permette di definire uno scheletro dell'algoritmo di comportamento generale all'interno della classe base Boss, ereditate poi dalle sue sottoclassi. La classe Boss sovrascrive e implementa altri metodi oltre a quelli ereditati da Enemy per definire i vari comportamenti, come movimenti particolari, attacchi speciali e reazioni ai danni subiti. Questo metodo promuove il riutilizzo del codice e la coerenza nel comportamento dei boss, e facilita l'estensione e l'implementazione di nuovi boss.

Schema UML



Implementazione dei Colpi generati dalle entità

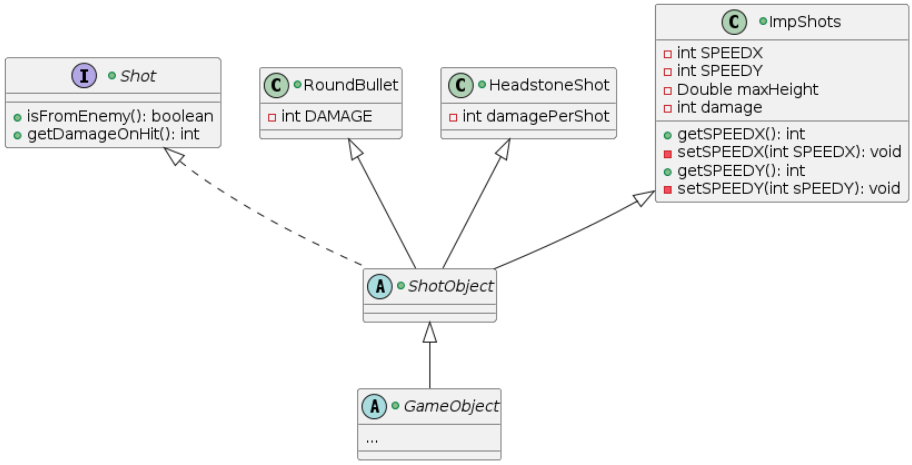
Descrizione del problema

Necessità di gestire colpi in un gioco con movimenti personalizzati e comportamenti differenziati se sparati da un nemico. Un oggetto ShotObject deve essere visto sia come un GameObject nella scena che come un colpo effettuato da un personaggio.

Descrizione della soluzione proposta

Per risolvere questa problema, è stata implementata un'interfaccia Shot. Questa interfaccia permette alla classe ShotObject di essere trattata polimorficamente come un GameObject o un colpo (Shot) a seconda delle esigenze del gestore della scena. L'implementazione sfrutta il pattern Bridge per separare l'astrazione dall'implementazione, permettendo così a ShotObject di variare indipendentemente dalle classi che lo usano. Inoltre, viene utilizzato il pattern Template Method per definire lo scheletro dei colpi, permettendo alle sue sottoclassi di personalizzare alcuni dei caratteri specifici del colpo senza cambiare la base.

Schema UML



Giacomo Biagioni

Gestione delle Scene

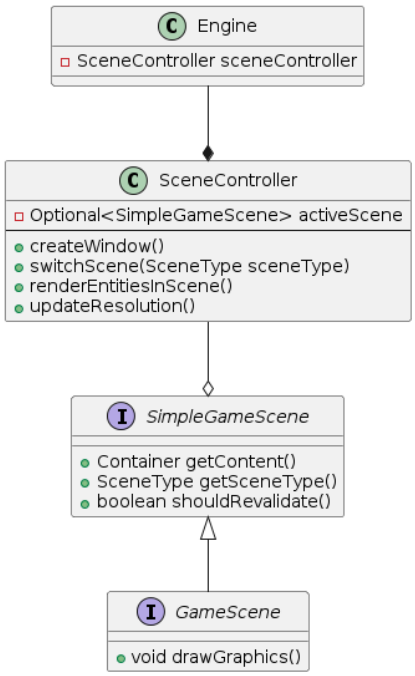
Descrizione del problema

Gestione delle scene di un gioco, inclusa la creazione, il passaggio tra diverse scene (menu principale, livello di gioco, menu di pausa, ecc.), e la chiusura del gioco. La gestione delle scene deve essere flessibile per supportare l'aggiunta di nuove scene.

Soluzione

La soluzione proposta prevede l'uso di una classe SceneController che funge da gestore centrale per tutte le scene del gioco. Questo controller è responsabile della creazione della finestra del gioco, del passaggio tra le scene, e della chiusura della finestra. Utilizza un'architettura basata su interfacce (SimpleGameScene e GameScene) per definire le operazioni comuni a tutte le scene. Per scene statiche (che non hanno aggiornamenti di grafica) basterà implementare SimpleGameScene mentre se si vuole creare una scena dinamica bisognera usare GameScene.

Schema UML



Pattern di Progettazione Utilizzati

La soluzione combina i seguenti due pattern:

- Factory Method: SceneController utilizza il pattern Factory Method per creare istanze delle varie scene (MainMenuScene, GameLevelScene, ecc.) in base al tipo di scena. Questo permette di centralizzare la logica di creazione delle scene all'interno del controller, rendendo il codice più gestibile e facilmente estendibile.
- Strategy: Le scene implementano l'interfaccia SimpleGameScene o GameScene (a seconda del tipo di scena che si vuole rappresentare), permettendo al controller di trattarle in modo uniforme.

Gestione dello score

Descrizione del problema

Il problema che si vuole risolvere riguarda il punteggio del giocatore tenendo conto di un fattore combo. Questo fattore combo aumenta ogni volta che il punteggio viene incrementato e si azzera se il timer combo scade. È necessario un meccanismo che permetta di aumentare il punteggio, aggiornare il timer della combo, riprendere il timer della combo e ottenere i valori attuali del punteggio, del tempo rimanente e del fattore combo.

Soluzione

La soluzione proposta prevede l'implementazione dell'interfaccia ScoreManager nella classe GameScore che gestisce il punteggio e il fattore combo. Questa classe offre metodi per aumentare il punteggio (increaseScore), aggiornare il timer della combo (updateComboTimer), riprendere il timer della combo (resumeComboTimer) e ottenere componenti grafici per disegnare il punteggio e il fattore combo (getDrawComponent). Inoltre, la classe fornisce metodi per ottenere il punteggio attuale (getScore), il tempo rimanente della combo (getRemainingTime) e il fattore combo (getComboFactor).

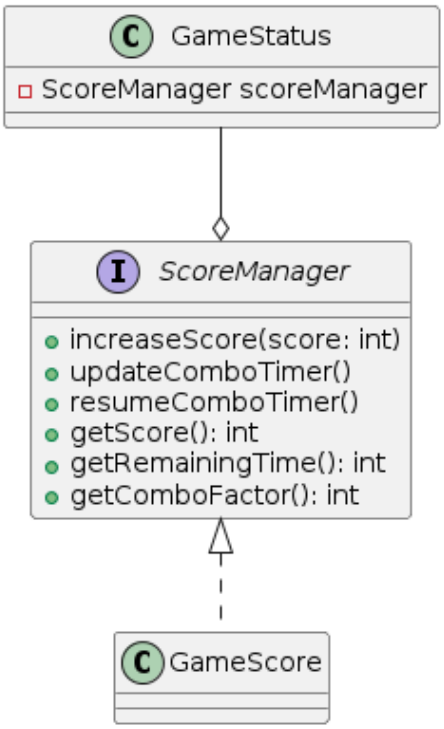
Pro della soluzione scelta

- Tutte le funzionalità relative al punteggio e al combo sono centralizzate in una singola classe, rendendo il codice facile da gestire e comprendere.

Contro della soluzione scelta

- La classe può diventare troppo grande se in futuro si aggiungono altre funzionalità legate al punteggio.
- Meno flessibile rispetto ad un approccio modulare, per come è impostato ora l'implementazione di ScoreManager permetterà solo di creare gestione del punteggio con un modificatore "combo".

Schema UML



Pattern di Progettazione Utilizzati

Il pattern utilizzato tra GameScore e ScoreManager è il Pattern Strategy. Nel contesto del Pattern Strategy, l'interfaccia ScoreManager definisce come dovranno esser impostate le classi di gestione del punteggio, mentre GameScore è una delle possibili implementazioni concrete di questa interfaccia.
Quindi per l'implementazione avremo:

- ScoreManager: Interfaccia che definisce i metodi che l'implementazione dovrà avere
- GameScore: classe che implementa l'interfaccia ScoreManager e che gestisce il punteggio del giocatore

Gestione degli eventi

Descrizione del problema

Il gioco richiede un sistema di gestione degli eventi, che verrà utilizzato per eseguire azioni in risposta a determinati avvenimenti. Gli eventi possono essere di vario tipo, come ad esempio il completamento di un livello, l'acquisizione di un oggetto speciale, o il raggiungimento di un obiettivo. Ogni evento deve poter essere riconosciuto, gestito e reagito in modo appropriato dal sistema di gioco.

Soluzione

La soluzione proposta utilizza un'interfaccia GameEvent per rappresentare gli eventi che possono essere attivati durante il gioco. Questa interfaccia definisce la base per tutti gli eventi del gioco, permettendo di creare implementazioni specifiche per diversi tipi di eventi. Gli eventi saranno poi processati durante l'esecuzione del loop di gioco e saranno risolti in ordine di chiamata. Il corretto ordinamento degli eventi sarà gestito da GameEventListener, che salverà inizialmente gli eventi in un buffer per poi inserirli nell'array effettivo solo una volta che gli eventi del ciclo precedente saranno stati processati. In questo modo si evita che l'array degli eventi venga modificato mentre viene letto.

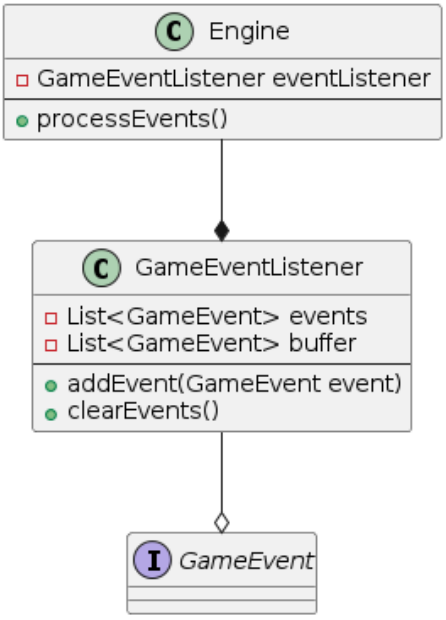
Pro della soluzione scelta

- Utilizzando un'interfaccia (GameEvent), possiamo facilmente aggiungere nuovi tipi di eventi.

Contro della soluzione scelta

- Gli eventi non vengono gestiti subito, ma in ordine di chiamata, quindi se ad esempio si vuole chiudere il gioco, prima che ciò avvenga saranno processati inutilmente gli altri eventi.

Schema UML



Pattern di Progettazione Utilizzati: Per gestire gli eventi abbiamo deciso di utilizzare un pattern di tipo Observer. GameEvent agisce come soggetto, mentre le classi che usano GameEventListener agiscono come osservatori. Questo pattern permette di disaccoppiare il codice che genera eventi da quello che li gestisce, migliorando la modularità e la gestione del codice.
Quindi per l'implementazione avremo:

- GameEvent: Interfaccia che rappresenta l'evento
- GameEventListener: Classe che deve essere implementata dalla classe che vuole gestire gli eventi
- Engine: Classe principale che catturerà e gestirà gli eventi

Gestione degli Asset statici e pesanti

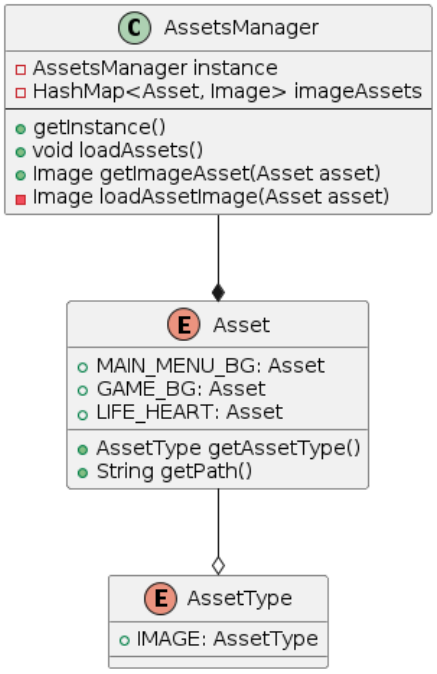
Descrizione del problema

Il problema da risolvere è la gestione efficiente del caricamento e dell'uso delle risorse (asset) statiche e pesanti nel gioco, come immagini di sfondo, entità non animate, etc. Il gioco richiede che questi asset siano disponibili rapidamente e facilmente accessibili durante l'esecuzione, il che implica che devono essere caricati e memorizzati in modo efficiente per evitare ritardi o interruzioni nel gameplay.

Soluzione

La soluzione proposta prevede l'uso di una classe AssetsManager per caricare e gestire gli asset del gioco. Gli asset sono definiti all'interno di un'enumerazione Asset, che specifica il tipo e il percorso di ciascun asset. L'AssetsManager carica gli asset all'avvio del gioco e li memorizza in una mappa hash per un rapido accesso.

Schema UML



Pattern di Progettazione Utilizzati

La soluzione combina i seguenti pattern:

- Singleton: La classe AssetsManager viene progettata come un singleton per garantire che esista una sola istanza che gestisce tutti gli asset e che sia facilmente accessibile senza dover passare riferimenti.
- Factory Method: Il metodo loadAssets nella classe AssetsManager agisce come un factory method per la creazione di assets a partire dai percorsi specificati nell'enumerazione Asset

3 Sviluppo

3.1 Testing automatizzato

TestMinotaur: Viene testato il corretto spostamento del Minotauro, e il corretto funzionamento della gestione degli Stati (quando sta fermo, corre, è stunnato o può essere colpito).

TestImp: Viene testato il corretto spostamento dell'Imp, e il corretto funzionamento della gestione degli Stati (quando sta fermo, si teletrasporta, attacca, genera il colpo, o può essere danneggiato).

TestBoundingBoxImplRet: Viene testata la corretta collisione tra una bounding box rettangolare con alcune circolari o rettangolari.

TestBoundingBoxImplCirc: Viene testata la corretta collisione tra una bounding box circolari con alcune circolari o rettangolari.

TestImpshots: Viene testato il corretto movimento (se lineare o a zig zag a seconda del parametro).

TestCharacterFactory: Viene testato le corrette collisioni del knight e wizard con il mondo e il corretto salto del knave e knight.

TestLevelFactory: Viene testata la creazione di un livello 0 e la possibilità di estrarre i nemici da quel livello.

TestEntityFactory: Viene testata la creazione di un nemico "fantoccio" di prova.

TestGameStatus: Viene testata l'operazione di rimozione e aggiunta di nemici nella effettiva scena di gioco.

TestPauseMenu: Viene testato il corretto funzionamento di cambio di scena, nello specifico il passaggio dalla scena di gioco al menu di pausa

TestScore: Viene creata la scena di gioco e si controlla il corretto incremento del punteggio

3.2 Note di sviluppo

Giacomo Biagioni

Uso di libreria di terze parti (Java Swing)
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/9b9748d08b8c9d30801ee54221813da9c7013
src/main/java/slayin/core/SceneController.java#L39
Uso di Optional
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/9b9748d08b8c9d30801ee54221813da9c7013
src/main/java/slayin/core/SceneController.java#L27
Uso di Lambda Expression
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/9b9748d08b8c9d30801ee54221813da9c7013
src/main/java/slayin/views/CharacterSelectionScene.java#L42
Funzione per la lettura del contenuto di un file
Autore: Neeme Praks
Link allo StackOverflow: https://stackoverflow.com/a/3849771
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/9b9748d08b8c9d30801ee54221813da9c7013
src/main/java/slayin/model/utility/LevelFactory.java#L42

Mattia Mencaccini

lambda
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/75d8f074070ac4a282a10d1a8838eb7d780ecb
src/main/java/slayin/model/entities/graphics/DrawComponentFactory.
java#L132
https://github.com/Morsu7/OOP23-slayin/blob/75d8f074070ac4a282a10d1a8838eb7d780ecb56/
src/main/java/slayin/model/entities/graphics/DrawComponentFactory.
java#L306

Cristian Morbidelli

Uso di libreria di terze parti (Json)
Utilizzata in vari punti della classe. Permalink: https://github.com/Morsu7/
OOP23-slayin/blob/main/src/main/java/slayin/model/utility/LevelFactory.
java#L125
Uso di Optional
Utilizzata in vari punti della classe. Permalink: https://github.com/Morsu7/
OOP23-slayin/blob/main/src/main/java/slayin/model/utility/LevelFactory.
java#L77

Lorenzo Morri

Stream
Collisione dei colpi con il personaggio. Permalink: https://github.com/Morsu7/
OOP23-slayin/blob/9b9748d08b8c9d30801ee54221813da9c7013002/src/main/

java/slayin/core/Engine.java#L155

Federico Morsucci

Uso di Optional
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/character/CharacterImpl.java#L34
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/character/CharacterImpl.java#L46
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/character/CharacterImpl.java#L108

Uso Lambda expressions
Permalink:https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/graphics/DrawComponentFactory.java#L56-L123
Utilizzata in vari punti della classe,Permalink: https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/character/CharacterFactory.java

Uso Stream
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/core/Engine.java#L140
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/core/Engine.java#L151
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/graphics/DrawComponentFactory.java#L77
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/World.java#L70
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/character/CharacterImpl.java#L183
https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/entities/character/CharacterImpl.java#L189

Uso Generici
Permalink: https://github.com/Morsu7/OOP23-slayin/blob/main/src/main/java/slayin/model/utility/Pair.java

Crediti:

- Repository ufficiale del corso tenuto dal professor Ricci: gitHub
- Pair.java presa dal codice di laboratorio del corso

4 Commenti finali

4.1 Autovalutazione e lavori futuri

Cristian Morbidelli

Nel complesso, l'attività di gruppo svolta è stata decisamente formativa: le sfide a cui si viene sottoposti sono molteplici e farne fronte dà poi possibilità di autocritica e di crescita. Progettare quasi da zero un'intera applicazione ti porta a dover ragionare da subito a 360 gradi e con un occhio al futuro sviluppo e alla crescita durante l'implementazione. Gestendo una parte di progetto centrale, ho sentito maggiormente anche la necessità di collaborare con tutti gli altri membri del gruppo durante tutto lo sviluppo in maniera tale da permettere e facilitare l'unione delle diverse parti di dominio che andavamo a sviluppare man mano. Questa necessità porta a maggior consapevolezza durante la propria produzione e rappresenta una sfida non indifferente, ma d'altra parte permette anche di migliorare rispetto ad un lavoro fatto totalmente in solitaria, ma permette anche di potersi confrontare nelle idee e nelle proposte oltre che riconoscere più facilmente i propri errori o le proprie mancanze. Valuto in maniera complessivamente positiva questa possibilità, con la consapevolezza che è stata assolutamente utile per migliorare ed apprendere cose nuove, e dunque potrò farne uso in futuro.

Lorenzo Morri

Ho apprezzato molto questa attività, poiché l'ho trovata un'esperienza dinamica e arricchente. Uno dei principali vantaggi di lavorare in gruppo è la possibilità di suddividere i compiti, ottimizzando così il flusso di lavoro. La revisione del codice da parte dei membri del team migliora notevolmente la qualità del prodotto finale. Questo processo non solo aumenta la robustezza del codice, ma promuove anche l'apprendimento collettivo. Mi sono principalmente occupato di implementare le Boss battle e la gestione dei colpi generati dai personaggi. Per risolvere i problemi descritti in precedenza, ho sviluppato diverse classi che, col senno di poi, riconosco potevano essere realizzate in modo più efficiente e potrebbero necessitare di alcune correzioni. Lo scambio di idee e opinioni con i compagni del gruppo è stato fondamentale per migliorare le mie competenze e rendere il mio lavoro più solido ed efficiente. Il risultato di questa esperienza può essere soltanto che positiva, soprattutto per aver imparato ad usare strumenti come GitHub e VisualStudio che si sono rivelati fondamentali per lo sviluppo di questo progetto.

Mattia Mencaccini

Durante il lavoro sul progetto, ho avuto l'opportunità di valutare criticamente il mio contributo, riconoscendo sia i punti di forza che quelli di debolezza del mio lavoro. Uno dei miei punti di forza è sicuramente la capacità di collaborare

efficacemente in gruppo. Ho trovato stimolante lavorare insieme agli altri membri per sviluppare le prime sezioni del progetto, contribuendo significativamente alla creazione dei primi nemici del gioco.
Un punto di debolezza che ho identificato riguarda la mia comprensione e utilizzo del codice scritto da altri. A volte ho avuto difficoltà a comprendere appieno alcune parti del codice già esistente e ho dovuto chiedere chiarimenti ai miei colleghi. Questo processo mi ha aiutato a migliorare nel corso del progetto, ma è stato un aspetto su cui ho dovuto lavorare costantemente.
Nel mio ruolo all'interno del gruppo, ho svolto un compito importante soprattutto nelle fasi iniziali, contribuendo alla creazione e all'integrazione dei primi nemici del gioco. Successivamente, il mio ruolo è stato più di supporto, focalizzandomi principalmente sull'implementazione dei requisiti specifici assegnati.
È importante sottolineare che ognuno di noi ha valutato il proprio lavoro e il proprio contributo in modo individuale, riflettendo le proprie esperienze e opinioni personali, pur mantenendo un atteggiamento collaborativo e rispettoso nei confronti del team.

Federico Morsucci

L'esperienza di collaborare a un progetto è stata estremamente istruttiva. Ho scoperto quanto il lavoro di gruppo possa arricchire in termini di conoscenze, creatività e problem solving. Sebbene avessi già lavorato in team, non avevo mai collaborato direttamente sullo stesso progetto.
Git e GitHub si sono rivelati essenziali per una gestione efficiente del codice e per una collaborazione fluida. Ho rapidamente compreso l'importanza del controllo di versione per tracciare modifiche, risolvere conflitti e ripristinare versioni precedenti.
Ho trovato utile sviluppare funzionalità separatamente e integrarle nel progetto principale solo quando pronte, migliorando la qualità e l'organizzazione del codice. Tuttavia, ho capito che la comunicazione nel team è fondamentale. Una maggiore interazione e pianificazione avrebbero potuto evitare errori e migliorare l'efficienza.
Il problema principale è stata la fase iniziale di progettazione: probabilmente avremmo dovuto aggiungere più interfacce e strutturare meglio il progetto per facilitarne lo sviluppo e la manutenzione.
In futuro, darò maggiore enfasi alla comunicazione e alla coordinazione del team, oltre a utilizzare strumenti simili per mantenere alta la qualità dei progetti.

Giacomo Biagioni

Nel valutare il mio lavoro, ho identificato alcuni punti di forza e di debolezza significativi. Tra i punti di forza, ho acquisito una buona capacità di coordinarmi con i miei colleghi durante il processo di sviluppo grazie all'utilizzo di Git. Tuttavia, ho incontrato delle difficoltà durante la fase di progettazione, trovando più complesso del previsto definire architetture e soluzioni senza avere una base concreta.

All'interno del gruppo, il mio ruolo principale è stato quello di gestire la visualizzazione delle viste del progetto; ho gestito anche il punteggio del giocatore e la base per il sistema di eventi.

Appendice A

A Guida Utente

A.1 Avvio dell'applicativo

Per avviare l'applicazione occorre aver installato all'interno della propria macchina una versione di Java 17 o pi'u recente.

- Posizionarsi all'interno della cartella contenente il jar relativo all'applicazione
- Eseguirlo tramite il comando java -jar slayin.jar

A.2 Istruzioni di gioco

Attraverso un semplice menù di gioco si può cambiare la risoluzione della finestra di gioco e tramite il tasto *ESC* mettere il gioco in pausa, in qualsiasi momento, una volta iniziata la partita. Il personaggio principale controllato dall'utente inizia la partita con i punti vita al massimo (10 LP).

Comandi

 - Muove il personaggio a sinistra/destra.

- In base al personaggio scelto, prima di avviare la partita, esegue la sua azione:

- Knight: Salta.
- Wizard: si trasforma in un tornado e spara un proiettile.
- Knave: Salta.

Piccola guida per i nemici e boss

Ogni nemico possono infliggere un quantitativo diverso di danni, specificato successivamente:

- Slime: 1 LP.
- Fire: 1 LP.
- Couatl: 2 LP.
- Headstone: 2 LP. I suoi colpi infliggono 1 LP.

I boss invece per ogni colpo subito possono enfatizzare le loro azioni come segue:

- Minotaur: Raddoppia i danni che infligge e i danni subiti ogni due colpi. Per essere sconfitto deve essere colpito 5 volte quando va a sbattere contro il lato dello schermo.
- Imp: Inizia a sparare solo dopo la prima hit. Aumenta il danno e il numero di colpi sparati di 1 ogni 3 hit. Per essere sconfitto deve essere colpito 10 volte in qualsiasi stato, eccetto quando sparisce dalla scena.

Come si muovono i nemici

- **Slime** : spawna sotto il terreno di gioco e sale fino a quando non incontra il terreno, per poi muoversi randomicamente a destra o sinistra, fino a quando non tocca il bordo di gioco o cambia direzione randomicamente
- **Fire** : spawna a metà dell'altezza del mondo, anche lui inizialmente si muove randomicamente a destra e sinistra, ma ogni tanto scende e si avvicina al giocatore, anche se un fire tenderà a rimanere in alto, ma mai più in alto della sua altezza iniziale
- **Coualt** : spawna a metà dell'altezza del mondo, inizialmente si muove randomicamente a destra o sinistra, ma ogni tanto questo nemico si fermerà, per poi fiondarsi sul terreno, rimanerci per qualche secondo e poi risalire lentamente
- **Headstone** : spawna ai lati dello schermo e sarà sempre rivolta verso il lato opposto, essendo una tomba, essa rimane ferma, ma spara micidiali proiettili che devono essere saltati o distrutti da particolari personaggi

Screen GamePlay