Wizards vs Trolls

Paradigmi di Programmazione e Sviluppo

Giacomo Foschi - giacomo.foschi3@studio.unibo.it - 0001179137
Giovanni Pisoni - giovanni.pisoni@studio.unibo.it - 0001189814
Giovanni Rinchiuso - giovanni.rinchiuso@studio.unibo.it - 0001195145

24 Ottobre 2025

Indice

Introduzione

Obiettivo

L'obiettivo di questo progetto è realizzare un videogioco tower defense ispirato a Plants vs Zombies, denominato **Wizards vs Trolls**.

Il giocatore ha il compito di difendere il castello dall'invasione continua di ondate di Troll, posizionando strategicamente diversi tipi di Maghi all'interno di una griglia di gioco. La partita prosegue indefinitamente con difficoltà crescente finché un Troll non riesce a raggiungere la torre, decretando la sconfitta del giocatore.

L'applicazione permetterà di:

- gestire strategicamente le risorse (elisir) che si rigenerano automaticamente nel tempo
- posizionare diversi tipi di maghi sulla griglia di gioco, ognuno con le proprie abilità, costi e statistiche
- affrontare ondate infinite di troll nemici con difficoltà progressivamente crescente
- utilizzare maghi specializzati: generatori di elisir, attaccanti elementali (fuoco, ghiaccio, vento) e barriere difensive
- guadagnare elisir come ricompensa per ogni nemico sconfitto
- mettere in pausa il gioco
- perdere la partita quando un troll raggiunge la torre

Autori

- Pisoni Giovanni (giovanni.pisoni@studio.unibo.it)
- Rinchiuso Giovanni (giovanni.rinchiuso@studio.unibo.it)
- Foschi Giacomo (giacomo.foschi3@studio.unibo.it)

Processo di sviluppo adottato

Il gruppo ha adottato una metodologia **Agile** per lo sviluppo del progetto.

In particolare, la scelta è ricaduta su un approccio **SCRUM-inspired** per la sua flessibilità e capacità di adattarsi alle esigenze del team e del progetto, producendo ad ogni iterazione nuove funzionalità del sistema o miglioramenti a quelle esistenti.

Per la coordinazione del team e la gestione del progetto sono stati utilizzati:

- **Google Docs**: per la gestione delle attività, la pianificazione dei task nelle iterazioni e la documentazione condivisa
- **GitHub**: per la collaborazione tra i membri, il versionamento del codice e la gestione dei branch
- **Discord**: per le comunicazioni quotidiane e le call tra i membri del team
- **IntelliJ IDEA**: come ambiente di sviluppo integrato (IDE) per la scrittura del codice

Il team è stato così suddiviso:

- **Product Owner**: si occupa di redigere il Product Backlog, definire le priorità delle funzionalità e verificare il corretto funzionamento del sistema realizzato. Il ruolo è stato assunto da **Rinchiuso Giovanni**

- **Committente**: esperto del dominio, garantisce l'usabilità e qualità del risultato. Il ruolo è stato assunto da **Pisoni Giovanni**
- **Sviluppatori**: tutti i membri del team hanno assunto il ruolo di sviluppatori:
  - Pisoni Giovanni
  - Rinchiuso Giovanni
  - Foschi Giacomo

Modalità di divisione in itinere dei task

La suddivisione dei task è stata gestita in modo collaborativo durante le riunioni di pianificazione degli sprint, per permettere a tutti i membri del team di contribuire alla definizione delle attività da svolgere.

Nel primo incontro (**Sprint Planning**) sono stati individuati i task principali del progetto e sono state assegnate delle priorità in base alla rilevanza delle funzionalità, andando così a redigere il **Product Backlog**. In questa fase è stata inoltre stabilita la **Definition of Done** secondo cui una funzionalità può considerarsi completata quando:

- è stata implementata e testata con esito positivo
- rispetta quanto richiesto dall'utente
- il codice è stato revisionato da almeno un altro membro del team
- la documentazione è stata aggiornata

Nelle successive riunioni di pianificazione degli sprint, i task sono stati ulteriormente suddivisi in attività più piccole (**Sprint Backlog**) per permettere un'equa distribuzione del lavoro tra i membri del team e semplificare la gestione operativa delle attività. Al termine di ogni sprint vengono inoltre redatti una **Sprint Review** e una **Sprint Retrospective**, per valutare, rispettivamente, sia il progresso a livello di funzionalità, sia il processo di sviluppo, individuando possibili aree di miglioramento.

Meeting/iterazioni pianificate

In una prima fase di analisi e modellazione, il gruppo ha partecipato a un meeting iniziale con l'obiettivo di definire l'architettura del progetto e stabilire le tecnologie da utilizzare. In quella stessa sede sono stati inoltre stabiliti la durata degli sprint e le modalità delle successive iterazioni.

Il team ha deciso di adottare **sprint settimanali** per permettere un rilascio rapido di funzionalità e ottenere un feedback frequente sullo stato di avanzamento del progetto.

La decisione di organizzare sprint brevi è stata motivata dall'esigenza di:

- sviluppare funzionalità in tempi brevi e mantenerle verificabili
- ottenere feedback rapido dal committente
- mantenere alta la reattività del team di fronte a eventuali problemi o cambiamenti nei requisiti

Oltre alle riunioni settimanali di pianificazione, il team ha previsto brevi confronti regolari per discutere dello stato di avanzamento e affrontare eventuali criticità o problemi tecnici emersi durante l'implementazione.

Modalità di revisione in itinere dei task

Per la gestione del codice, è stato adottato un approccio basato su **branch dedicati per sprint**. All'inizio di ogni sprint veniva creato un branch specifico (ad esempio `sprint-1`, `sprint-2`, ecc.) sul quale tutti i membri del team lavoravano in parallelo alle diverse funzionalità previste per quella iterazione.

Al termine di ogni sprint, durante la Sprint Review, il codice consolidato nel branch dello sprint veniva integrato nel branch `main` tramite merge, dopo aver verificato che tutte le funzionalità fossero completate secondo la Definition of Done e che i test automatici fossero superati con successo. Questo approccio ha permesso di avere:

- una chiara separazione tra il codice in sviluppo e quello stabile in produzione
- milestone ben definite corrispondenti ai vari sprint
- una cronologia pulita e organizzata del progetto

Scelta degli strumenti di test, build e Continuous Integration (CI)

Per il testing si è scelto di utilizzare **ScalaTest** come framework di automazione, essendo una tecnologia matura e ben integrata nell'ecosistema Scala, mentre come build tool è stato scelto **sbt**, in quanto nasce specificatamente per Scala e offre un'ottima gestione delle dipendenze. Inoltre, è stato utilizzato **scalafmt** per formattare automaticamente il codice sorgente rendendolo coerente e standardizzato all'interno del team.

L'intero progetto è stato gestito tramite **GitHub**. Per automatizzare i processi di test e controllo qualità, è stata implementata una pipeline di **continuous integration** (CI) su **GitHub Actions**. Questa pipeline si attiva automaticamente a ogni nuova push o pull request sui branch principali (escluso il contenuto della cartella docs), garantendo che il codice rispetti gli standard prefissati prima di essere integrato.

Il workflow configurato esegue le seguenti azioni principali:

- **Controllo della formattazione**: Verifica che il codice sorgente rispetti gli standard di formattazione definiti nel file *.scalafmt.conf*, utilizzando il comando sbt `scalafmtCheckAll` per mantenere una codebase coerente e leggibile.
- **Generazione report PDF**: Questa pipeline si attiva specificatamente quando vengono modificati i file markdown del report nella cartella `docs/report` (o le immagini associate). Utilizza *Pandoc* per convertire i file markdown aggiornati in un unico file PDF (`docs/report.pdf`). Se il PDF generato è diverso dalla versione precedente presente nel repository, il workflow effettua automaticamente il commit del nuovo file PDF nel branch principale, mantenendo così la documentazione PDF sempre allineata con i sorgenti markdown.
- **Build e test**: Compila il codice ed esegue la suite di test automatici (sbt test) per prevenire regressioni e assicurare la correttezza del software. Questo processo viene eseguito su diverse piattaforme (Ubuntu, Windows, macOS) utilizzando JDK 21 per garantire la compatibilità cross-platform.

Requisiti

L'analisi del problema svolta nella prima fase del progetto ha permesso di evidenziare i requisiti elencati di seguito.

Requisiti di business

- **Creare un'esperienza di gioco coinvolgente e sfidante**: Il genere tower defense è stato scelto per la sua natura strategica che richiede pianificazione e decisioni tattiche continue. La scelta di un sistema a ondate infinite con difficoltà crescente, invece di livelli predefiniti, è motivata dalla volontà di massimizzare la rigiocabilità e creare una sfida sempre nuova per il giocatore
- **Utilizzo di Scala e di paradigmi funzionali**: Il progetto deve essere sviluppato in Scala 3 per permettere al team di apprendere e applicare costrutti funzionali di alto livello
- **Rispetto della deadline**: Realizzare il progetto entro il 24 ottobre 2025, pianificando gli sprint in modo da completare prima le funzionalità core e lasciare le feature opzionali per le iterazioni finali

Modello di dominio

Il dominio del progetto ruota attorno ai seguenti concetti principali:

Grid

La griglia di gioco rappresenta il campo di battaglia:

- Definisce le posizioni valide dove possono essere posizionati i maghi
- È composta da celle organizzate in righe (5) e colonne (9)
- Mantiene traccia delle celle occupate e di quelle disponibili
- Valida i tentativi di posizionamento impedendo sovrapposizioni

Wizard

Unità difensiva posizionabile dal giocatore sulla griglia:

- Ha un costo in elisir necessario per il posizionamento
- Possiede statistiche: salute, danno, raggio d'attacco, tempo di ricarica tra attacchi
- Attacca automaticamente i troll che entrano nel proprio raggio
- Rimane fermo nella posizione in cui è stato piazzato
- Viene rimosso quando la salute raggiunge zero

Esistono cinque tipi di maghi:

- **Mago Generatore**: Genera 25 elisir ogni 10 secondi invece di attaccare. Costo: 100, Vita: 150
- **Mago del Vento**: Attacco base a distanza. Costo: 150, Vita: 100, Danno: 25, Gittata: 3.0, Cooldown: 3s
- **Mago del Fuoco**: Infligge alto danno a corto raggio. Costo: 250, Vita: 100, Danno: 50, Gittata: 2.0, Cooldown: 2.5s
- **Mago del Ghiaccio**: Rallenta temporaneamente i nemici colpiti. Costo: 200, Vita: 150, Danno: 25, Gittata: 2.5, Cooldown: 4s
- **Mago Barriera**: Alta salute ma nessun attacco. Funziona come muro difensivo. Costo: 200, Vita: 300

Troll

Unità nemica che avanza verso la torre:

- Si muove automaticamente da destra verso sinistra
- Ha statistiche: salute, velocità, danno, raggio d'attacco, cooldown
- Attacca i maghi che incontra sul percorso (se nel raggio d'azione)
- Se raggiunge il lato sinistro della griglia, causa la sconfitta immediata del giocatore
- Viene rimosso quando la salute raggiunge zero
- Rilascia elisir come ricompensa quando eliminato

Esistono quattro tipi di troll:

- **Troll Base**: Statistiche equilibrate. Vita: 100, Vel: 0.10, Danno: 20, Gittata: 1.0, Cooldown: 1s
- **Troll Guerriero**: Alta salute e danno, movimento più veloce, corto raggio. Vita: 130, Vel: 0.15, Danno: 30, Gittata: 0.5, Cooldown: 1.5s
- **Troll Assassino**: Altissima velocità e danno ma bassa salute, si muove a zigzag. Vita: 70, Vel: 0.2, Danno: 60, Gittata: 1.5, Cooldown: 0.8s
- **Troll Lanciatore**: Attacca i maghi a distanza, rimanendo fermo. Vita: 40, Vel: 0.10, Danno: 10, Gittata: 5.0, Cooldown: 3s

Elixir

Risorsa economica gestita dal giocatore:

- Ha un valore corrente (inizia a 200) e un limite massimo (1000)
- Si rigenera automaticamente (+100 elisir ogni 10 secondi)
- Viene consumato per posizionare i maghi
- Viene guadagnato sconfiggendo i troll (quantità varia per tipo di troll)
- Può essere generato dai Maghi Generatori (+25 elisir ogni 10 secondi)
- Determina quali maghi possono essere posizionati in un dato momento

Castle

Obiettivo da difendere (non una vera entità, ma la condizione di sconfitta):

- Rappresenta la meta finale del percorso dei troll (lato sinistro della griglia)
- Se un troll la raggiunge, la partita termina immediatamente con la sconfitta

Wave

Ondata di troll che appare periodicamente:

- Ha un numero identificativo progressivo
- Contiene una composizione di diversi tipi di troll generata proceduralmente
- La difficoltà aumenta ad ogni ondata successiva:
  - Più troll vengono generati (max troll per ondata aumenta)
  - Le statistiche dei troll (salute, velocità, danno) aumentano progressivamente
  - Il tempo tra le generazioni diminuisce
  - La distribuzione dei tipi di troll cambia, introducendo tipi più forti
- Il sistema di spawn si attiva dopo il posizionamento del primo mago
- Non ha limite massimo: il gioco continua all'infinito fino alla sconfitta

Projectile

Proiettile lanciato dai maghi (Vento, Fuoco, Ghiaccio) o dai Troll Lanciatori:

- Ha una posizione e si muove verso il lato opposto (destra per maghi, sinistra per troll)
- Ha un tipo che determina il danno e gli effetti (Fuoco, Ghiaccio, Vento, Troll)
- Colpisce il primo bersaglio valido sulla stessa riga nella cella in cui entra
- Viene rimosso dopo aver colpito
- I proiettili del Mago del Ghiaccio applicano un effetto di rallentamento temporaneo

Requisiti funzionali

Requisiti di utente

Dal punto di vista dell'utente, il sistema deve consentire:

- **Il setup della partita**:
  - Visualizzazione del menu principale
  - Avvio di una nuova partita
  - Accesso alle informazioni di gioco
- **L'interazione di gioco**:
  - Selezionare un tipo di mago dal pannello laterale (shop) visualizzando costo, icona e nome
  - Posizionare i maghi cliccando su celle valide della griglia
  - Visualizzare in tempo reale:
    - Quantità di elisir disponibile
    - Numero dell'ondata corrente
    - Barre della vita di maghi e troll (quando non a vita piena o uguale a 0)
    - Proiettili in movimento
  - Ricevere feedback immediato per:
    - Tentativi di posizionamento non validi (cella occupata, elisir insufficiente, mago non selezionato)
    - Inizio di nuove ondate
  - Mettere in pausa il gioco e riprendere
- **La fine della partita**:
  - Ricevere notifica chiara di game over quando un troll raggiunge la fine
  - Ricevere notifica di vittoria alla fine di un'ondata
  - Possibilità di continuare alla prossima ondata (dopo vittoria) o iniziare una nuova partita (dopo sconfitta o da menu pausa)
  - Possibilità di tornare al menu principale

Requisiti di sistema

Il sistema dovrà occuparsi di:

- **Gestione delle entità di gioco**:
  - Creare e mantenere tutte le entità (maghi, troll, proiettili) con identificatori unici
  - Associare a ogni entità le sue caratteristiche (componenti: posizione, salute, statistiche, etc.)
  - Permettere ricerche efficienti di entità con specifiche caratteristiche
  - Rimuovere automaticamente entità quando vengono eliminate

- **Gestione dell'elisir**:
  - Inizializzare l'elisir al valore predefinito (200) all'inizio della partita
  - Rigenerare elisir automaticamente a intervalli regolari (+100 ogni 10s)
  - Rispettare il limite massimo di elisir accumulabile (1000)
  - Verificare che il giocatore abbia elisir sufficiente prima di permettere il posizionamento di maghi
  - Sottrarre il costo corretto quando un mago viene posizionato
  - Aggiungere elisir quando un troll viene eliminato (quantità variabile)
  - Gestire la generazione periodica di elisir dai Maghi Generatori (+25 ogni 10s)
- **Validazione del posizionamento**:
  - Verificare che il click sia all'interno della griglia
  - Verificare che la cella non sia già occupata da un altro mago
  - Verificare che il giocatore abbia elisir sufficiente per il mago selezionato
  - Fornire feedback (messaggio di errore) in caso di tentativo non valido
- **Gestione del movimento**:
  - Aggiornare continuamente le posizioni dei troll (da destra a sinistra) e dei proiettili (direzione opposta) in base alla loro velocità e al delta time
  - Implementare il movimento a zigzag per i Troll Assassini
  - Applicare gli effetti di rallentamento quando un troll viene colpito dal ghiaccio, riducendone la velocità
  - Rilevare quando un troll raggiunge il lato sinistro della griglia
  - Rimuovere i proiettili che escono dai bordi dello schermo
- **Gestione del combattimento**:
  - Implementare il targeting automatico: maghi attaccano il troll più vicino sulla stessa riga nel loro range; Troll Lanciatori attaccano il mago più vicino sulla stessa riga nel loro range
  - Gestire i tempi di ricarica per ogni entità attaccante
  - Creare proiettili quando un'entità effettua un attacco a distanza
  - Rilevare le collisioni tra entità e bersagli nella stessa cella e gestirne l'effetto (meleeAttack o projectile-entity collision)
  - Applicare effetti speciali alla collisione
  - Gestire il blocco del movimento per i Troll che attaccano in mischia
- **Gestione della salute**:
  - Processare le collisioni per ridurre la vita delle entità
  - Rilevare quando un'entità raggiunge salute zero o inferiore
  - Rimuovere le entità morte dal gioco
  - Assegnare ricompense in elisir quando un troll viene eliminato
- **Gestione delle ondate**:
  - Iniziare a generare ondate di troll solo dopo che il primo mago è stato piazzato
  - Determinare la composizione di ogni ondata (numero e tipi di troll) proceduralmente in base al numero dell'ondata
  - Aumentare progressivamente la difficoltà:
    - Incrementando il numero massimo di troll per ondata
    - Aumententando le statistiche base (salute, velocità, danno) dei troll generati
    - Diminuendo l'intervallo tra le generazioni
    - Modificando la probabilità di apparizione dei tipi di troll
  - Generare troll in "batch" a intervalli randomizzati
  - Rilevare il completamento di un'ondata (spawn terminato e nessun troll rimasto)

- **Ciclo di gioco principale**:
  - Mantenere aggiornato e consistente lo stato del gioco
  - Processare tutti gli aggiornamenti dei sistemi ECS in un ordine definito
  - Gestire correttamente la pausa e la ripresa del gioco, sospendendo gli aggiornamenti e la generazione di spawn
  - Rilevare le condizioni di vittoria e sconfitta e terminare/procedere la partita
- **Rendering e interfaccia**:
  - Disegnare la mappa di gioco e la griglia
  - Visualizzare tutte le entità (maghi, troll, proiettili) nelle loro posizioni con le loro icone
  - Mostrare l'effetto visivo per le entità congelate
  - Disegnare le barre della vita delle entità con salute non piena
  - Mostrare l'HUD con informazioni su elisir, ondata corrente, pannello shop
  - Gestire la visualizzazione e l'interazione con i menu

Requisiti non funzionali

Requisiti esterni

- **Performance**:
  - Mantenere un frame rate stabile (idealmente vicino a 60 FPS) anche con un numero elevato di entità
- **Affidabilità**:
  - Garantire stabilità durante sessioni di gioco prolungate senza crash
  - Gestire robustamente input utente non validi
  - Mantenere la consistenza dello stato durante pausa/ripresa
- **Usabilità**:
  - Interfaccia intuitiva
  - Icone e testi chiari
  - Feedback visivo immediato per le azioni
  - Informazioni essenziali sempre visibili

Requisiti interni

- **Scalabilità**:
  - Facilità di aggiungere nuovi tipi di maghi e troll modificando principalmente le configurazioni
  - Aggiungere nuove caratterisitche e funzionalità per le entità di gioco
  - Capacità di introdurre nuove meccaniche
- **Manutenibilità**:
  - Separazione netta tra logica di gioco e interfaccia grafica
  - Codice modulare
  - Utilizzo di immutabilità per ridurre effetti collaterali
  - Codice ben documentato e con nomi descrittivi
- **Testabilità**:
  - Logica di gioco testabile in isolamento dal rendering
  - Comportamento deterministico facilitato dall'immutabilità e dal game loop fisso
  - Utilizzo di DSL per creare scenari di test specifici
  - Buona copertura dei test sulle logiche critiche

Requisiti di implementazione

- **Metodologia di sviluppo**: Agile SCRUM-inspired
- **Architettura**: MVC con Model implementato tramite ECS
- **Tecnologie e linguaggio**: Scala 3.x, ScalaFX per la UI, SBT per il build
- **Testing**: ScalaTest
- **Versioning e collaborazione**: Git, GitHub, GitHub Actions per CI

Design architetturale

Il design architetturale del sistema è stato elaborato a partire dai requisiti funzionali e non funzionali identificati. L'obiettivo principale è stato creare una struttura modulare, performante ed estensibile che potesse gestire la complessità di un gioco tower defense come *Wizards vs Trolls*, garantendo alte performance con numerose entità simultanee e una chiara separazione delle responsabilità tra i vari componenti.

Pattern Architetturale MVC (Model-View-Controller)

Questo pattern è ampiamente utilizzato nello sviluppo di applicazioni software per separare la logica dalla sua rappresentazione grafica e dall'interazione con l'utente. I suoi tre componenti principali svolgono ruoli specifici:

- **Model**: Il Model rappresenta il cuore dell'applicazione. Contiene i dati e la logica di business del sistema. La sua responsabilità è gestire lo stato dell'applicazione, manipolare i dati e implementare tutte le regole del gioco. È completamente disaccoppiato dalla rappresentazione grafica e non ha conoscenza dell'interfaccia utente. Nel nostro caso, il Model contiene tutte le entità di gioco (maghi, troll, proiettili), le loro caratteristiche, la logica di combattimento, movimento, gestione risorse e progressione delle ondate. Per implementare il Model abbiamo scelto di utilizzare il pattern **Entity Component System (ECS)**, che verrà descritto in dettaglio nella sezione successiva
- **View**: La View è l'interfaccia utente. Il suo scopo è presentare i dati del Model all'utente e raccogliere gli input dell'utente. La View non contiene alcuna logica di gioco e si limita a visualizzare lo stato corrente del Model. Nel nostro progetto, la View include la griglia di gioco, la visualizzazione delle entità (maghi, troll, proiettili), l'HUD con informazioni su elisir e ondata corrente, il menu principale e la schermata di Game Over. Quando l'utente interagisce con la View (ad esempio, cliccando per posizionare un mago), questa inoltra l'input al Controller per la sua elaborazione
- **Controller**: Il Controller agisce come intermediario tra il Model e la View. Riceve l'input dall'utente tramite la View, lo elabora, aggiorna il Model di conseguenza e istruisce la View a riflettere i cambiamenti di stato. Nel nostro caso, il Controller coordina l'esecuzione del game loop, gestisce il posizionamento dei maghi, valida gli input dell'utente e determina le condizioni di game over. In sostanza, il Controller garantisce che le diverse parti dell'applicazione rimangano indipendenti e comunichino attraverso interfacce ben definite

Questa separazione è la motivazione per la quale abbiamo scelto di utilizzare il pattern MVC per il nostro progetto. Permette di sviluppare e testare ciascun componente in modo indipendente, facilita la manutenzione e l'estensione del sistema. Inoltre, consente di sostituire facilmente la View

(ad esempio, passare da ScalaFX a un'altra libreria grafica) senza dover modificare il Model o il Controller.

Pattern Entity Component System (ECS) per il Model

All'interno del Model, abbiamo scelto di implementare la logica di gioco utilizzando il pattern **Entity Component System (ECS)**, un pattern data-oriented ampiamente utilizzato nello sviluppo di videogiochi.

L'ECS si basa su tre concetti fondamentali:

- **Entity**: Un identificatore unico che rappresenta un'entità di gioco (mago, troll, proiettile). L'Entity non contiene dati né comportamenti, è semplicemente un ID che collega diversi Component
- **Component**: Strutture dati pure che rappresentano singoli aspetti di un'entità (posizione, salute, attacco, movimento). I Component non contengono logica, solo dati. Ad esempio, `PositionComponent` contiene le coordinate (x, y), `HealthComponent` la salute corrente e massima, `AttackComponent` il danno, raggio e cooldown
- **System**: Contengono tutta la logica di gioco e operano su gruppi di entità che possiedono specifici Component. Ogni System ha una responsabilità unica e ben definita: `MovementSystem` aggiorna le posizioni, `CombatSystem` gestisce gli attacchi, `ElixirSystem` gestisce le risorse, `HealthSystem` applica i danni

Questa scelta architetturale è stata motivata da diversi fattori:

1. **Performance**: La separazione tra dati (Component) e logica (System) favorisce la località dei dati in memoria, migliorando l'efficienza della cache CPU. Questo è cruciale in un tower defense dove possono esistere decine di entità simultanee che devono essere aggiornate 60 volte al secondo
2. **Composizione flessibile**: Invece di gerarchie di ereditarietà rigide, le entità sono definite dalla combinazione di Component che possiedono. Un mago attaccante ha Component di posizione, salute e attacco, mentre un Mago Generatore ha un Component generatore di elisir al posto dell'attacco. Questo permette di creare nuovi tipi di entità senza modificare gerarchie esistenti
3. **Modularità**: Ogni System gestisce un aspetto specifico del gioco in modo indipendente, rendendo il codice più comprensibile e manutenibile
4. **Estensibilità**: Aggiungere nuove funzionalità significa creare nuovi Component e/o System senza modificare il codice esistente, rispettando il principio Open/Closed
5. **Testabilità**: Ogni System può essere testato in isolamento creando scenari specifici con solo le entità e i Component necessari

Struttura del Progetto

La struttura del progetto è organizzata in quattro moduli principali, che riflettono una chiara separazione delle responsabilità ispirata al pattern Model-View-Controller (MVC), con una distinzione esplicita per il motore di gioco (Engine).

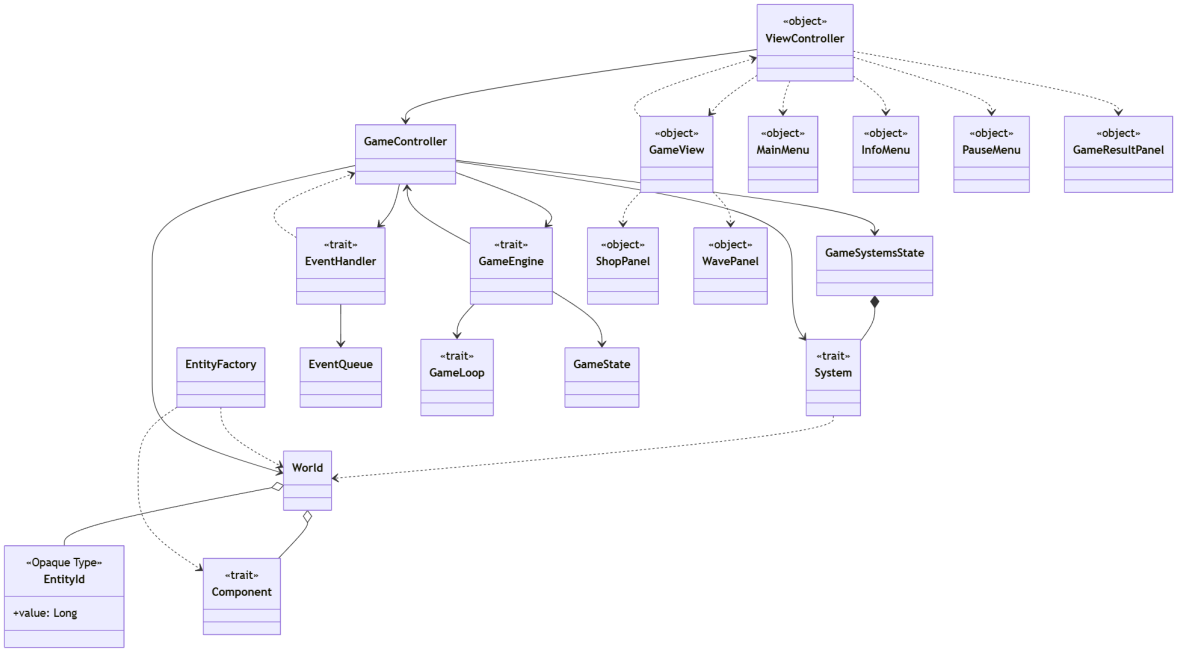

Figura 1: Architettura del Progetto

1. **Model (ECS)**: Rappresenta il nucleo logico del gioco. Implementato con il pattern Entity-Component-System, questo modulo definisce la struttura dei dati (`Component`), la logica di gioco (`System`) e il contenitore dello stato del mondo (`World`). È completamente disaccoppiato dagli altri moduli e si occupa esclusivamente delle regole e dello stato della simulazione. La EntityFactory astrae la creazione delle entità di gioco
2. **Engine**: È il cuore pulsante dell'applicazione. Il `GameEngine` gestisce il ciclo di vita del gioco, orchestrando le transizioni di stato (es. `Playing`, `Paused`) definite in `GameState`. Si appoggia al `GameLoop` per garantire un'esecuzione a timestep fisso, assicurando che la logica di gioco progredisca in modo deterministico e indipendente dal frame rate
3. **Controller**: Agisce come intermediario, collegando l'input dell'utente, la logica di gioco e il motore. Il `GameController` riceve eventi dall'`EventHandler` e comanda al `GameEngine` di aggiornare lo stato. Il `GameSystemsState` aggrega e gestisce l'esecuzione ordinata di tutti i sistemi logici del Model
4. **View**: Costituisce l'interfaccia grafica. Il `ViewController` gestisce la navigazione tra le diverse schermate (`GameView`, `MenuView`, etc.). La `GameView` si occupa del rendering degli elementi di gioco, inclusi componenti specifici come `ShopPanel` e `WavePanel`. La `View` cattura le interazioni dell'utente e le inoltra al `Controller` sotto forma di eventi, senza contenere alcuna logica di gioco

Principi di Programmazione Funzionale

L'intero progetto è stato sviluppato seguendo principi di programmazione funzionale, come richiesto dai requisiti:

- **Immutabilità**: Tutte le strutture dati (Component, configurazioni) sono immutabili. I

System restituiscono nuove versioni di sé stessi invece di modificare lo stato interno. Questo elimina bug legati a modifiche inattese e garantisce thread-safety

- **Composizione su Ereditarietà**: L'ECS favorisce la composizione tramite Component invece di gerarchie di classi rigide
- **Pure Function**: I System sono implementati come funzioni il più possibile pure: dato un input (World), producono un output (nuovo World) in modo deterministico
- **Type Classes**: Utilizzo di given instances e type classes per il polimorfismo ad-hoc nella creazione delle entità, garantendo type-safety ed estensibilità
- **Opaque Types**: Utilizzo di opaque types (EntityId) per type-safety senza overhead runtime

Design di Dettaglio

Panoramica

In questa sezione verrà approfondito il design delle componenti chiave del progetto *Wizards vs Trolls*, illustrando le principali responsabilità funzionali, le scelte implementative e le interazioni tra i moduli. L'analisi segue il pattern **Model-View-Controller (MVC)**, con un focus sull'implementazione del **Model** tramite l'architettura **Entity-Component-System (ECS)**, ispirandosi alla struttura descrittiva vista nel documento di esempio.

Model (ECS)

Il Model racchiude e gestisce l'intera logica di business del gioco, implementata tramite il pattern **Entity Component System (ECS)**. Di seguito sono riportate le principali scelte di design e responsabilità.

Gestione dello Stato del Mondo (`World`)

Il nucleo del Model è rappresentato dal `World`, una `case class` **immutabile** che funge da contenitore centrale per tutte le entità e i loro componenti. L'immutabilità garantisce che ogni operazione (eseguita dai `System`) restituisca un nuovo stato del mondo senza modificare quello precedente, aderendo ai principi funzionali e semplificando la gestione dello stato in un ambiente potenzialmente concorrente.

- **Responsabilità**:
  - Mantenere l'insieme di tutte le entità attive (`EntityId`)
  - Mappare i tipi di componenti alle entità che li possiedono (`Map[Class[_], ↩ Map[EntityId, Component]]`)
  - Fornire API funzionali per creare/distruggere entità e aggiungere/rimuovere/aggiornare/recuperare componenti (es. `createEntity()`, `addComponent()`, `getComponent()`, `getEntitiesWithComponent()`)
  - Permettere query specifiche sullo stato (es. `getEntitiesByType()`, `getEntityAt()`, `hasWizardAt()`)

Creazione delle Entità (EntityFactory)

La creazione delle diverse entità del gioco (Maghi, Troll, Proiettili) è centralizzata nell'object `EntityFactory`. Questo approccio utilizza il pattern **Factory Method** combinato con **Type**

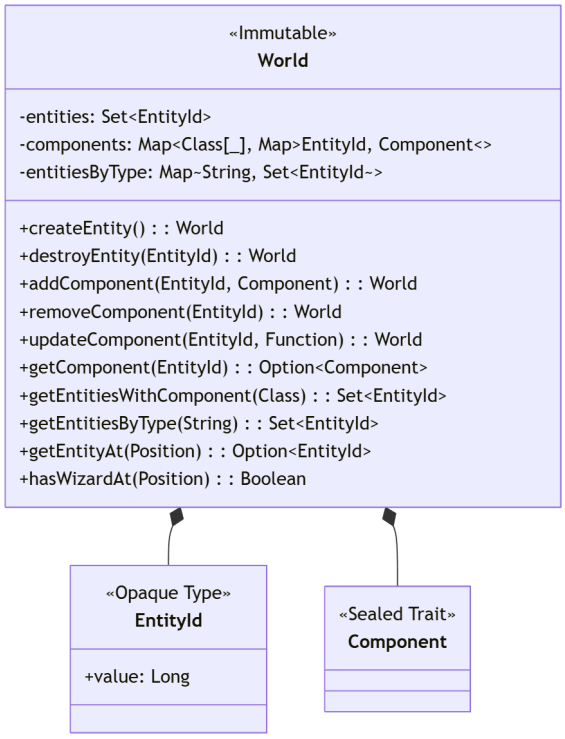

Figura 2: World Diagram

Classes (`EntityBuilder`) per assemblare le entità in modo componibile e type-safe.

- **Responsabilità**:
  - Definire **configurazioni** (`WizardConfig`, `TrollConfig`, `ProjectileConfig`) che descrivono le proprietà base di ciascun tipo di entità
  - Utilizzare `EntityBuilder` (implementati tramite `given instances`) per costruire la lista di `Component` necessari per ogni configurazione
  - Fornire **metodi specifici** (es. `createFireWizard`, `createBaseTroll`, `createProjectile`) che prendono il `World` corrente, la posizione e il tipo di entità, restituendo il `World` aggiornato con la nuova entità e il suo `EntityId`
  - Astrarre i dettagli dell'aggiunta dei singoli componenti al `World`

Logica di Gioco (Systems)

Tutta la logica comportamentale è incapsulata nei **`System`**. Ogni `System` è una `case class` (stateless) che implementa il trait `System`, definendo un metodo `update(world: World): (World, System)`. Questo metodo prende lo stato attuale del mondo e restituisce il nuovo stato modificato e, potenzialmente, una nuova istanza del sistema (anche se spesso restituisce `this` essendo stateless).

Strategie di Movimento (MovementSystem) Questo sistema gestisce lo spostamento di tutte le entità mobili.

- **Responsabilità**:
  - Aggiornare la `PositionComponent` delle entità in base alla loro `MovementComponent` e al `deltaTime`

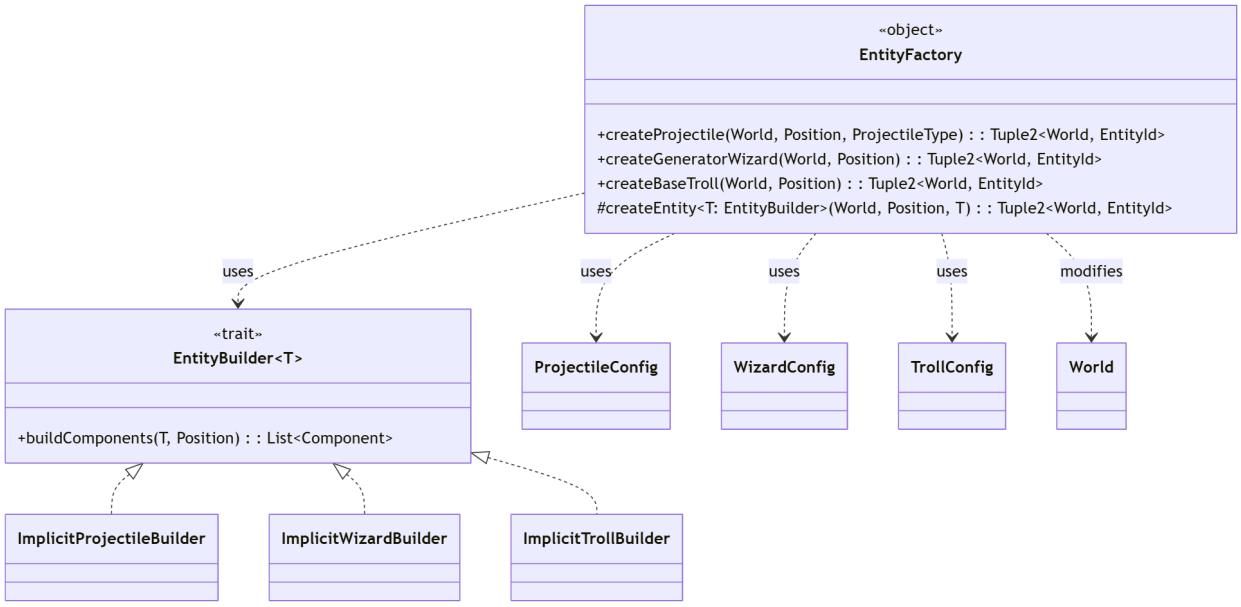

Figura 3: EntityFactory Diagram

- Applicare diverse **strategie di movimento** in base al tipo di entità (tramite pattern matching sull' `EntityTypeComponent`):
  - Movimento lineare a sinistra per i Troll (`linearLeftMovement`)
  - Movimento lineare a destra per i Proiettili dei Maghi (`projectileRightMovement`)
  - Movimento lineare a sinistra per i Proiettili dei Troll
  - Movimento a zigzag per i Troll Assassini (`zigzagMovement`), gestito tramite lo `ZigZagStateComponent` per mantenere lo stato specifico dell'entità
- Considerare gli effetti di stato come il rallentamento (`FreezedComponent`)
- Rimuovere i proiettili che escono dai limiti dello schermo

Gestione del Combattimento e Collisioni (CombatSystem, CollisionSystem)

Il combattimento è diviso in due fasi gestite da sistemi distinti:

CombatSystem:

- **Responsabilità**: Iniziare gli attacchi a distanza
- **Logica**: Identifica le entità attaccanti (Maghi, Troll Lanciatori), cerca bersagli nel raggio d'azione (`findClosestTarget`), verifica il cooldown (`CooldownComponent`) e, se possibile, crea un'entità `Projectile` usando `EntityFactory` e imposta il cooldown sull'attaccante. Gestisce anche la logica di blocco (`BlockedComponent`) per i Troll Lanciatori. Aggiorna i timer dei `CooldownComponent` e `FreezedComponent`

CollisionSystem:

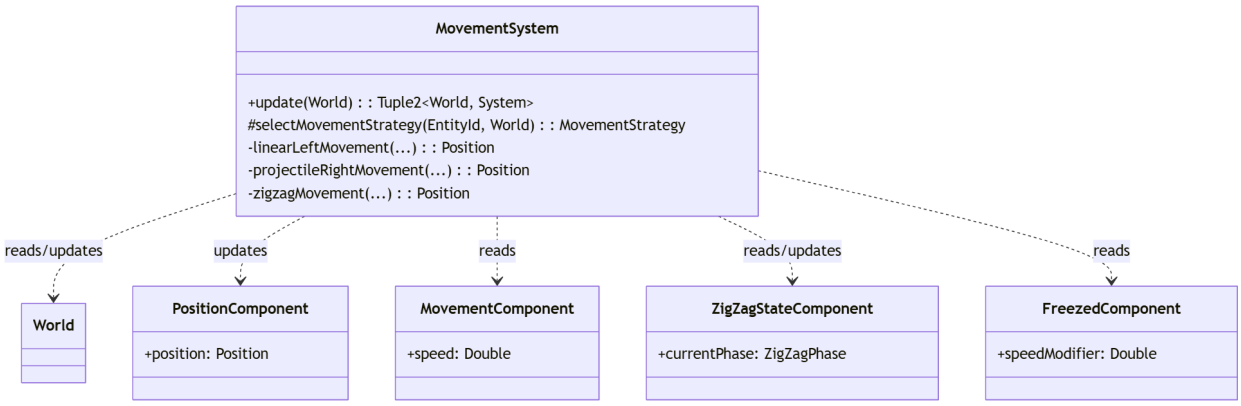

Figura 4: MovementSystem Diagram

- **Responsabilità**: Rilevare e risolvere le collisioni fisiche
- **Logica**:
  - **Proiettili**: Verifica se la cella di un proiettile coincide con quella di un bersaglio valido. Se sì, distrugge il proiettile e aggiunge un `CollisionComponent` (con il danno) al bersaglio. Applica l'effetto `FreezedComponent` se il proiettile era di ghiaccio
  - **Mischia**: Verifica se un Troll (non Lanciatore) è nella stessa cella di un Mago. Se sì, aggiunge un `BlockedComponent` al Troll, e se non è in cooldown, aggiunge un `CollisionComponent` al Mago e imposta il cooldown sul Troll

Generazione Nemici (SpawnSystem)

Questo sistema gestisce l'apparizione dei Troll sulla mappa.

- **Responsabilità**:
  - Schedulare e generare ondate di Troll (`SpawnEvent`)
  - Attivarsi solo dopo il posizionamento del primo Mago
  - Incrementare la difficoltà (`WaveLevel`) aumentando numero, tipo e statistiche dei Troll generati
  - Generare Troll in "batch" a intervalli variabili per un flusso meno prevedibile
  - Applicare lo scaling delle statistiche ai Troll creati in base all'ondata corrente
  - Gestire la pausa del gioco sospendendo e riprendendo correttamente la generazione

Gestione Risorse ed Effetti (ElixirSystem, HealthSystem)

ElixirSystem:

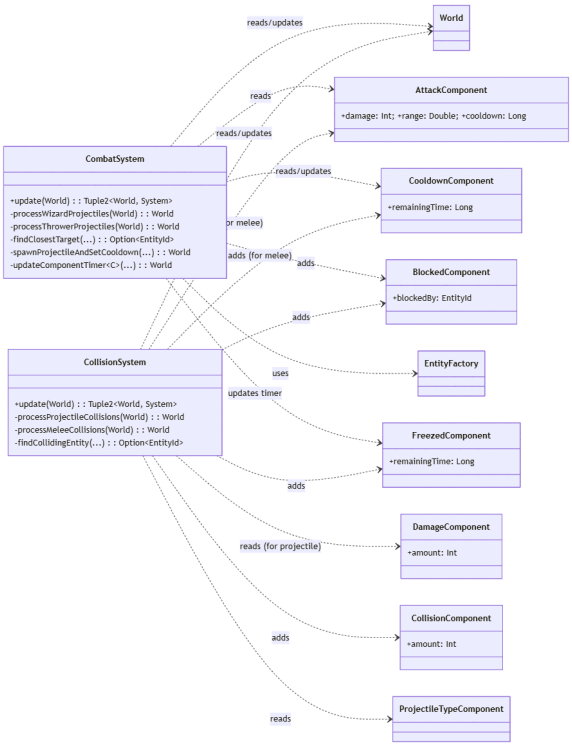

Figura 5: Combat and Collision Systems Diagram

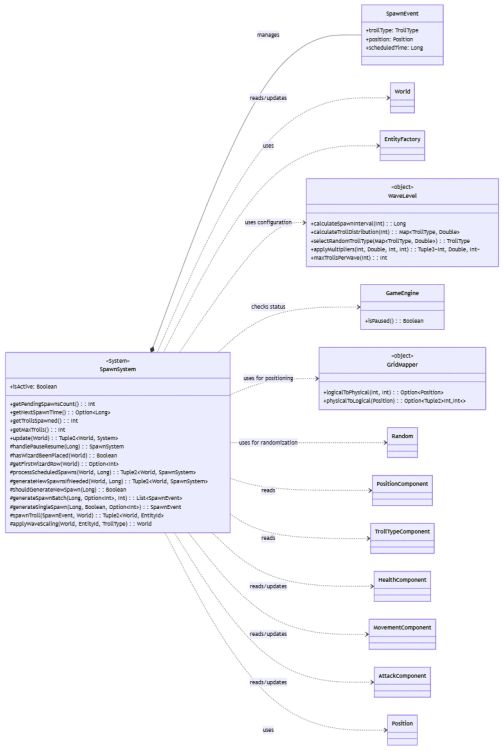

Figura 6: SpawnSystem Diagram

- **Responsabilità**: Gestire la risorsa Elixir del giocatore
- **Logica**: Traccia l'ammontare corrente (`totalElixir`), gestisce la generazione periodica automatica e quella dei Maghi Generatori (interagendo con `CooldownComponent`), permette di spendere (`spendElixir`) e aggiungere (`addElixir`) elisir, rispettando il cap massimo (`MAX_ELIXIR`)

HealthSystem:

- **Responsabilità**: Gestire la salute delle entità e le conseguenze del danno
- **Logica**: Processa i `CollisionComponent` aggiunti dal `CollisionSystem`, sottrae la salute dalla `HealthComponent`, rimuove il `CollisionComponent`. Se la salute scende a zero:
  - Marca l'entità per la rimozione
  - Se è un Troll, comunica all'`ElixirSystem` di aggiungere la ricompensa
  - Rimuove fisicamente le entità marcate dal `World` (`destroyEntity`)
  - Gestisce la rimozione a cascata dei `BlockedComponent` quando l'entità bloccante muore

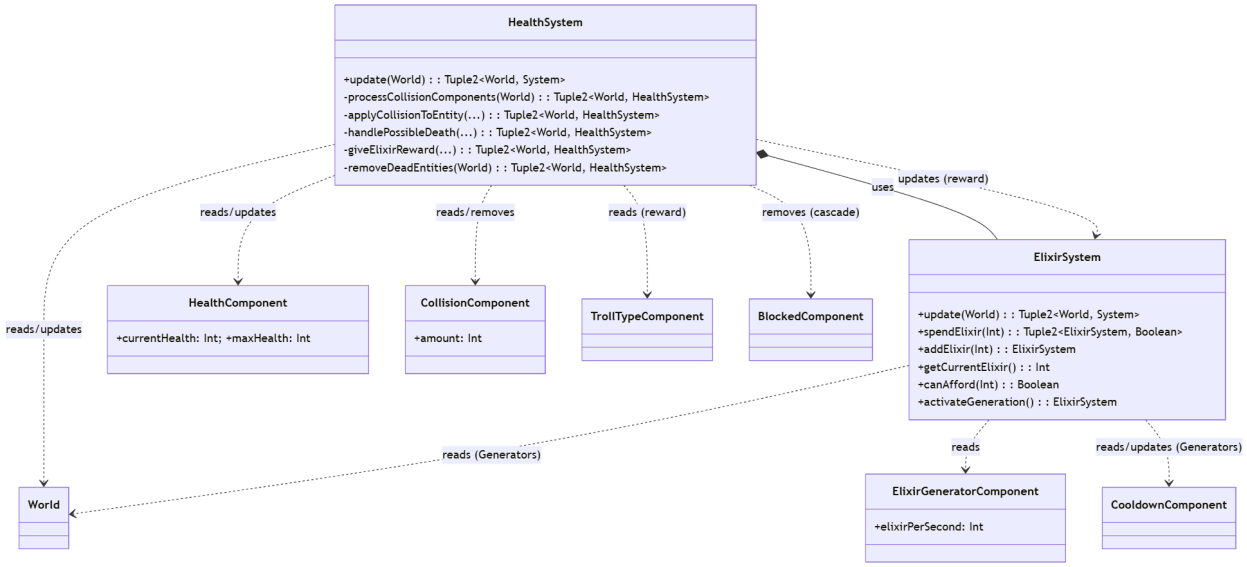

Figura 7: Elixir and Health Systems Diagram

View

La **View** si occupa della presentazione grafica dello stato del gioco e dell'interazione diretta con l'utente, utilizzando **ScalaFX**.

Gestione delle Schermate

- **Responsabilità**: Mostrare la schermata appropriata (Menu Principale, Gioco, Info, Pausa, Vittoria, Sconfitta) in base allo stato dell'applicazione

- **Componenti**:
  - `ViewController`: Gestisce le transizioni tra stati (`ViewState`) e aggiorna la `Scene` della `PrimaryStage`
  - `MainMenu`, `InfoMenu`, `PauseMenu`, `GameResultPanel`: `object` che definiscono la struttura e i controlli di ciascuna schermata statica

Rendering della Scena di Gioco

- **Responsabilità**: Disegnare lo stato corrente del `World` a schermo
- **Componenti**:
  - `GameView`: Organizza i diversi livelli grafici (`Pane` sovrapposti) e fornisce metodi (`renderEntities`, `renderHealthBars`, `drawGrid`) per aggiornare specifici livelli. Utilizza `Platform.runLater` per garantire che gli aggiornamenti avvengano sul thread UI. Gestisce i click sulla griglia
  - `RenderSystem` (nel Model, guida la View): Determina cosa deve essere disegnato
  - `HealthBarRenderSystem` (nel Model, guida la View): Sottosistema specializzato per calcolare quali barre della vita mostrare
  - `GridMapper`: Utility `object` utilizzato da `GameView` per convertire le coordinate fisiche (click del mouse) in logiche (cella della griglia) e viceversa, per disegnare la griglia (`drawGrid`) e posizionare le entità (`renderEntities`)
  - `Position`: `case class` che rappresenta le coordinate fisiche (pixel), utilizzata da `GameView` per posizionare gli elementi grafici

Creazione Componenti UI

- **Responsabilità**: Standardizzare la creazione e l'aspetto degli elementi UI riutilizzabili, simile ai factory pattern visti nell'esempio
- **Componenti**:
  - `ButtonFactory`: Crea bottoni (`Button`) con stili predefiniti (Presets basati su `ButtonConfig`) e associa direttamente `ButtonAction` che vengono tradotte in `GameEvent`
  - `ImageFactory`: Carica e gestisce `ImageView`, implementando un sistema di caching per ottimizzare l'uso della memoria e i tempi di caricamento
  - `ShopPanel`, `WavePanel`: Creano e gestiscono i pannelli specifici dell'HUD (negozio e informazioni sull'ondata)

Controller

Il **Controller** agisce come collante, orchestrando il flusso di dati e la logica applicativa tra il Model e la View.

Orchestrazione del Flusso di Gioco

- **Responsabilità**: Far avanzare lo stato del gioco nel tempo e coordinare l'esecuzione della logica
- **Componenti**:
  - `GameController`: Riceve l'impulso (`update()`) dal `GameEngine` (tramite il `GameLoop`). Mantiene lo stato corrente dei sistemi (`GameSystemsState`). Chiama il metodo `updateAll()` di `GameSystemsState` per eseguire la pipeline dei sistemi ECS nell'ordine corretto. Gestisce le azioni del giocatore ricevute come `GameEvent` dall'`EventHandler`

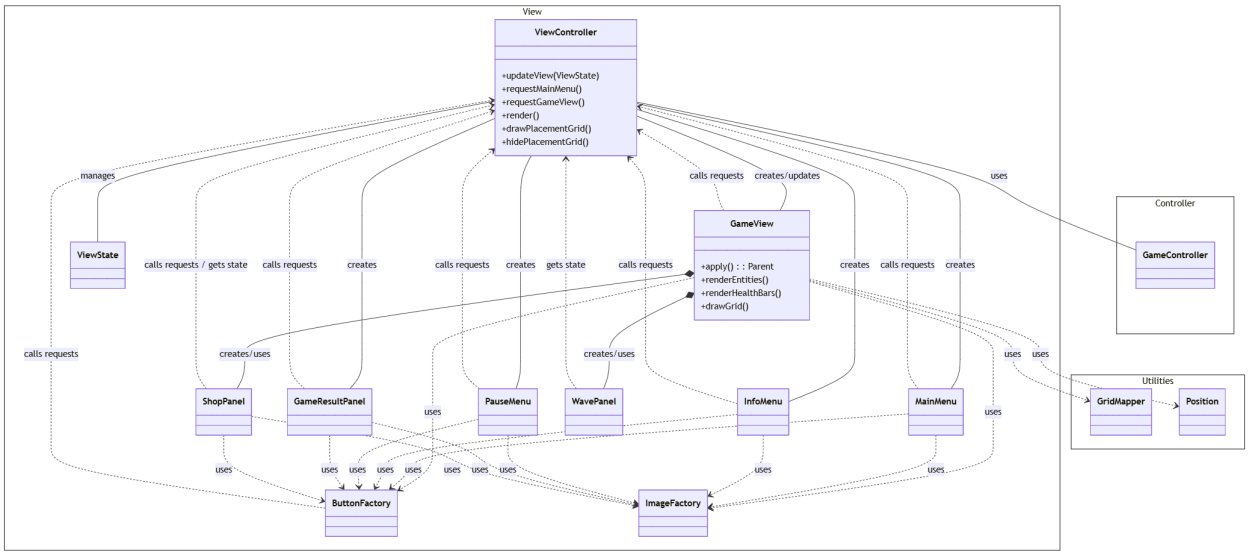

Figura 8: View Diagram

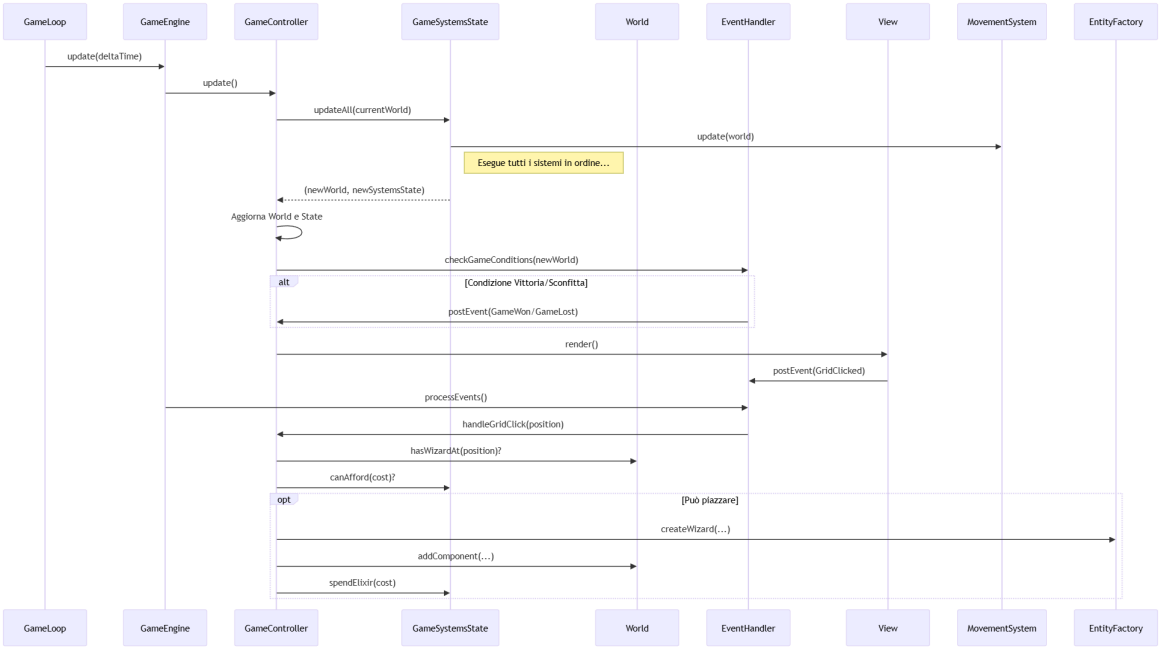

Figura 9: Controller Diagram

  - `GameSystemsState`: Raggruppa tutti i sistemi ECS e definisce l'ordine di update. Contiene anche metodi per verificare le condizioni di fine partita (`checkWinCondition`, `checkLoseCondition`)
  - `GameEngine` / `GameLoop`: Componenti architetturali che implementano il ciclo di vita principale dell'applicazione. La loro funzione è orchestrare la progressione temporale dello stato di gioco. A tal fine, invocano il metodo `GameController.update()` in modo periodico e a intervalli di tempo discreti e costanti (fixed timestep). Questo approccio garantisce che la logica di gioco evolva in maniera deterministica e disaccoppiata dalla frequenza di rendering (frame rate), assicurando un comportamento consistente indipendentemente dalle prestazioni dell'hardware. Sebbene operino a un livello architetturale superiore e siano esterni al GameController, ne governano l'esecuzione

Gestione degli Eventi

- **Responsabilità**: Disaccoppiare i componenti e gestire la comunicazione e le transizioni di stato in modo centralizzato
- **Componenti**:
  - `EventHandler`: Mantiene una coda thread-safe (`EventQueue`) di `GameEvent`. Riceve eventi da View (input utente), `GameEngine` (cambiamenti di stato globali), `GameController` (condizioni di gioco). Processa gli eventi in base alla loro priorità, invocando handler registrati o gestendo direttamente le transizioni di stato del `GameEngine` e della `ViewController` (es. passaggio da `Playing` a `Paused`)
  - `GameEvent`: ADT (`sealed trait`) che definisce tutti i tipi di eventi possibili, con una priorità associata

Gestione dell'Input

- **Responsabilità**: Validare e interpretare l'input grezzo dell'utente
- **Componenti**:
  - `InputSystem`: Riceve le coordinate grezze del mouse click dalla `GameView` (inoltrate tramite `ViewController` e `GameController`)
  - `InputProcessor`: Contiene la logica per verificare se un click (`MouseClick`) ricade all'interno dell'area valida della griglia (`isInGridArea`)
  - `ClickResult`: `case class` che rappresenta l'esito della validazione dell'input (posizione valida/invalida, eventuale messaggio di errore)
  - `GridMapper`: Utilizzato per convertire le coordinate fisiche (pixel) in coordinate logiche (riga/colonna) se il click è valido. L'`EventHandler` riceverà poi un `GridClicked` event con le coordinate logiche

Implementazione

Questa sezione analizza in dettaglio le soluzioni tecniche e le scelte implementative adottate per tradurre il design architetturale in un prodotto software funzionante. Il lavoro è stato suddiviso tra i membri del team per coprire le diverse aree del progetto, dalla logica di base del motore di gioco al comportamento delle singole entità e all'interfaccia utente.

Le sezioni seguenti illustrano i contributi di ciascun membro, evidenziando come i principi della programmazione funzionale e i pattern scelti, siano stati applicati concretamente per costruire i vari moduli del gioco.

Implementazione - Giacomo Foschi

Panoramica dei Contributi

Il mio contributo al progetto si è concentrato sulla progettazione e implementazione di diversi aspetti chiave del sistema, con un focus particolare sull'architettura ECS (Entity-Component-System) e sul rendering grafico. Le aree principali di cui mi sono occupato sono:

- **Architettura del `World` ECS**: Progettazione del `World` come contenitore **immutabile** per la gestione di entità e componenti, sfruttando le `case class` e le collezioni immutabili di Scala.
- **Struttura della View**: Implementazione del `ViewController` per la gestione degli stati della UI e del `GameView` per il rendering della scena di gioco.
- **Factory per la UI**: Creazione di `ButtonFactory` e `ImageFactory` per standardizzare e ottimizzare la creazione di elementi grafici, con gestione funzionale degli errori tramite `Option` ed `Either` e caching.
- **Logica di Combattimento e Collisioni**: Sviluppo del `CombatSystem` e del `CollisionSystem` come **funzioni pure** (`World => World`) sullo stato del mondo, comunicando tramite l'aggiunta/rimozione di componenti immutabili.
- **Utility per la Griglia**: Implementazione del `GridMapper` per la conversione tra coordinate logiche e fisiche.
- **Movimento dei Proiettili**: Definizione della logica di movimento specifica per i proiettili all'interno del `MovementSystem`.
- **Sistema di Rendering**: Creazione del `RenderSystem` per la visualizzazione delle entità a schermo, con ottimizzazioni basate su hashing dello stato per evitare rendering ridondanti.

Architettura del `World` in ECS

Il `World` è il cuore del nostro pattern **Entity-Component-System (ECS)**. È stato progettato come una `case class` **immutabile** che agisce da contenitore per tutte le entità di gioco e i loro componenti (`case class` immutabili per design). La sua immutabilità è fondamentale per aderire ai principi della programmazione funzionale: ogni aggiornamento (eseguito dai `System`) produce un *nuovo* stato del mondo senza modificare quello precedente (nessun side effect), semplificando il debugging, la gestione dello stato e garantendo la thread-safety intrinseca.

Le sue responsabilità principali sono:

- **Gestione delle Entità**: Fornisce metodi per creare (`createEntity`) e distruggere (`destroyEntity`) entità in modo sicuro. Ogni entità è rappresentata da un `EntityId` univoco
- **Gestione dei Componenti**: Permette di aggiungere (`addComponent`), rimuovere (`removeComponent`) e aggiornare (`updateComponent`) componenti associati a un'entità. I componenti sono semplici `case class` che contengono solo dati (es. `PositionComponent`, `HealthComponent`)
- **Querying**: Offre API per interrogare lo stato del gioco, come ottenere tutte le entità con un certo componente (`getEntitiesWithComponent`) o di un certo tipo (`getEntitiesByType`)

Un esempio di come un sistema interagisce con il `World` per aggiungere un componente a un'entità:

```
// Esempio di utilizzo del World
case class World(
    private val entities: Set[EntityId] = Set.empty,
    private val components: Map[Class[_], Map[EntityId, Component]] = Map.empty,
    // ...
```

```
):

  def addComponent[T <: Component](entity: EntityId, component: T): World =
    if !entities.contains(entity) then
      this
    else
      // ... logica per aggiungere il componente in modo immutabile
      val componentClass = component.getClass
      val updatedComponents = components.updatedWith(componentClass): opt =>
        Some(opt.getOrElse(Map.empty) + (entity -> component))

      val updatedEntitiesByType = updateEntityTypeMapping(entity, component)

      copy(components = updatedComponents, entitiesByType = updatedEntitiesByType)
```

Questa architettura favorisce la **composizione** sull'ereditarietà (le entità sono definite dinamicamente dalla combinazione dei loro componenti) e garantisce una netta **separazione tra dati** (Componenti, `case class` immutabili) e **logica** (Sistemi, funzioni pure `World => World`), rendendo il codice modulare, testabile e manutenibile.

Struttura della View: `ViewController` e `GameView`

L'interfaccia utente (UI), basata su ScalaFX, è gestita principalmente da `ViewController` e `GameView`, seguendo i principi del pattern MVC, che collaborano per presentare lo stato del gioco all'utente e per gestire i suoi input.

ViewController

Il `ViewController`, implementato come `object` che estende `JFXApp3`, orchestra l'intera UI. È l'entry point dell'applicazione ScalaFX e gestisce le transizioni tra le diverse schermate del gioco (`ViewState`, un Algebraic Data Type definito con `sealed trait`) aggiornando la `Scene` principale della `PrimaryStage`.

Le sue responsabilità includono:

- **Gestione degli Stati della UI**: Il metodo `updateView` riceve un nuovo `ViewState` e, tramite pattern matching, seleziona e carica il layout corretto (`MainMenu()`, `GameView()`, etc.). Mantiene lo stato corrente in una `case class ViewControllerState`.
- **Inizializzazione e Pulizia**: Si occupa di creare l'istanza del `GameController` all'avvio e gestisce la pulizia (`cleanup`) delle risorse (es. cache immagini, stato `GameView`) quando si esce da una schermata complessa come quella di gioco verso il menù principale.
- **Inoltro degli Input**: Fornisce metodi `requestXYZ` (es. `requestGameView`, `requestPlaceWizard`) che traducono le azioni dell'utente in `GameEvent` specifici, inviandoli poi al `GameController` tramite `postEvent`. Questo disaccoppia la View dalla logica del Controller.

GameView

Il `GameView`, anch'esso un `object`, è il componente responsabile del rendering della schermata di gioco principale. Utilizza uno `StackPane` per sovrapporre diversi livelli (`Pane`): Sfondo, Griglia overlay, Entità (maghi/troll), Proiettili, Barre della Vita, e UI Overlay (Shop, Wave, Pausa).

- **Sfondo**: L'immagine di background della mappa di gioco

- **Griglia**: Un `Pane` per disegnare overlay sulla griglia, come le celle valide per il posizionamento
- **Entità**: Un `Pane` dove vengono renderizzati i maghi e i troll
- **Proiettili**: Un `Pane` separato per i proiettili, per poterli gestire indipendentemente
- **Barre della Vita**: Un `Pane` per le barre della salute
- **UI Overlay**: Il livello più alto che contiene elementi come lo `ShopPanel`, il `WavePanel` e il pulsante di pausa

Il `GameView` espone metodi come `renderEntities` e `renderHealthBars` che vengono invocati dal `RenderSystem` (tramite il `ViewController`) per aggiornare la visualizzazione a schermo in modo efficiente, assicurando che le operazioni di rendering avvengano sul thread della UI di JavaFX tramite `Platform.runLater`.

Factory per la UI: `ButtonFactory` e `ImageFactory`

Per promuovere il riutilizzo del codice e garantire uno stile grafico coerente, ho implementato due factory object.

ImageFactory

`ImageFactory` centralizza la creazione e gestione di `ImageView`. La sua caratteristica principale è l'implementazione di un **sistema di caching** (`mutable.Map`) per ottimizzare performance e memoria.

```
object ImageFactory:
  private val imageCache: mutable.Map[String, Image] = mutable.Map.empty

  // Restituisce Option[Image], gestendo il fallimento del caricamento
  private def loadImage(path: String): Option[Image] =
    imageCache.get(path).orElse(loadAndCacheImage(path))

  // Restituisce Either per una gestione errori più esplicita
  def createImageView(imagePath: String, width: Int): Either[String, ImageView] =
    loadImage(imagePath)
      .toRight(s"Error loading image at path: $imagePath")
      .map(image => createFixedWidthImageView(image, width))
```

Quando viene richiesta un'immagine (`loadImage`), la factory controlla la cache; se l'immagine non è presente, tenta di caricarla (`loadAndCacheImage` usa `Option(getClass.getResourceAsStream(path))` per gestire resource non trovate) e la memorizza. Il metodo `createImageView` propaga l'eventuale fallimento usando `Either[String, ImageView]`, permettendo al chiamante (`GameView`) di gestire l'errore in modo funzionale (es. con `fold` o pattern matching) invece di usare eccezioni. Questo approccio ottimizza le performance e riduce il consumo di memoria, evitando di ricaricare più volte la stessa immagine.

ButtonFactory

`ButtonFactory` standardizza la creazione dei bottoni. Utilizza una `case class ButtonConfig` per definire l'aspetto di un bottone (testo, dimensioni, font) e una serie di `Presets` per configurazioni comuni (es. `mainMenuButtonPreset`, `shopButtonPreset`).

Questo permette di creare bottoni con uno stile omogeneo in tutta l'applicazione con una sola riga di codice, associando direttamente un'`azione` che viene eseguita `onAction`.

```
def createStyledButton(config: ButtonConfig)(action: => Unit): Button =
    createButton(config).withAction(action).withOverEffect().build()
```

Utilizza un `ButtonBuilder` interno (pattern Builder) per una configurazione fluente. Inoltre, il ButtonFactory gestisce anche l'associazione tra ButtonAction (un ADT che rappresenta le azioni possibili) e le chiamate al ViewController, tramite pattern matching, mantenendo la logica di navigazione disaccoppiata dalla definizione dei bottoni.

Logica di Combattimento e Collisioni

Il combattimento è gestito da `CombatSystem` e `CollisionSystem`, entrambi case class stateless che implementano il trait `System`, operando come funzioni pure `World => (World, System)`. Queste classi collaborano tra loro per gestire i cicli di attacco e risoluzione delle collisioni.

CombatSystem

Questo sistema inizia gli attacchi a distanza. Il metodo `update` prende il `World` corrente e restituisce un nuovo `World` modificato. La logica interna utilizza ampiamente costrutti funzionali:

- **Iterazione Funzionale**: Usa `foldLeft` sulle liste di entità (ottenute tramite `world.getEntitiesByType`) per processare attaccanti maghi e troll lanciatori in modo immutabile
- **Gestione dell'Assenza (Monadi)**: La ricerca del bersaglio (`findClosestTarget`) usa `flatMap`, `filter`, `minByOption` su `Option` e collezioni per trovare il bersaglio più vicino sulla stessa riga, restituendo `Option[EntityId]` per gestire il caso in cui non ci siano bersagli validi

In particolare, il system si occupa di:

- **Scansionare le entità**: Itera su tutte le entità che possono attaccare (maghi e troll lanciatori)
- **Ricercare dei bersagli**: Per ogni attaccante, cerca un bersaglio valido all'interno del suo raggio d'attacco (`findClosestTarget`). La ricerca è ottimizzata per controllare solo le entità sulla stessa riga
- **Gestire i Cooldown**: Verifica che l'attaccante non sia in fase di cooldown
- **Creare dei Proiettili**: Se tutte le condizioni sono soddisfatte, utilizza l'`EntityFactory` per creare un'entità proiettile nella posizione dell'attaccante e aggiunge un `CooldownComponent` all'attaccante per prevenire attacchi troppo ravvicinati

```
// In CombatSystem.scala
private def spawnProjectileAndSetCooldown(
      world: World,
      entity: EntityId,
      position: Position,
      projectileType: ProjectileType,
      cooldown: Long
  ): World =
    val (world1, _) = EntityFactory.createProjectile(world, position, projectileType)
    world1.addComponent(entity, CooldownComponent(cooldown))
```

CollisionSystem

Questo sistema risolve le collisioni, prendendo il `World` modificato dal `CombatSystem` e restituendo un nuovo `World`.

- **Iterazione Funzionale**: Come `CombatSystem`, usa `foldLeft` per processare proiettili e troll in mischia (`processProjectileList`, `processMeleeList`).
- **Gestione dell'Assenza (Monadi)**: La logica per processare una singola collisione (`processProjectileCollision`, `processMeleeCollision`) è incapsulata in for-comprehension su `Option` per estrarre i componenti necessari (posizione, tipo, danno). Se un componente manca, il for-comprehension fallisce e restituisce il `World` invariato tramite `.getOrElse(world)`. La ricerca dell'entità collidente (`findCollidingEntity`) usa anch'essa `Option` e `flatMap`.
  - **Collisioni dei Proiettili**: Il sistema itera su tutti i proiettili attivi e controlla se la loro posizione (cella della griglia) coincide con quella di un'entità bersaglio. Se viene rilevata una collisione:
    * Il proiettile viene distrutto
    * Un `CollisionComponent`, contenente l'ammontare del danno, viene aggiunto all'entità bersaglio
    * Se il proiettile è di tipo "ghiaccio", viene aggiunto anche un `FreezedComponent` per rallentare il bersaglio
  - **Collisioni in Mischia**: Successivamente, il sistema gestisce gli attacchi in mischia dei troll. Se un troll si trova nella stessa cella di un mago, il troll viene bloccato (aggiungendo un `BlockedComponent`) e, se non è in cooldown, infligge danno al mago aggiungendo un `CollisionComponent`

Questa separazione di responsabilità permette di gestire in modo pulito e modulare i diversi tipi di interazione offensiva nel gioco. Il danno vero e proprio viene poi applicato dall'`HealthSystem` in una fase successiva del ciclo di gioco.

Utility e Sistemi di Supporto

GridMapper

Il `GridMapper` è un utility object fondamentale che funge da "traduttore" tra il sistema di coordinate logiche della griglia (righe e colonne) e il sistema di coordinate fisiche dello schermo (pixel x, y).

Fornisce metodi essenziali come:

- `logicalToPhysical`: Converte una coppia `(riga, colonna)` nella posizione centrale in pixel di quella cella
- `physicalToLogical`: Converte coordinate `(x, y)` in pixel nella coppia `(riga, colonna)` corrispondente

Questo disaccoppia completamente la logica di gioco (che ragiona in termini di griglia) dalla rappresentazione grafica (che lavora con i pixel), rendendo il codice più pulito e manutenibile.

Movimento dei Proiettili

Il movimento di tutte le entità, inclusi i proiettili, è gestito dal `MovementSystem`. Per i proiettili, la logica è semplice e lineare:

- I proiettili dei maghi si muovono da sinistra verso destra (`projectileRightMovement`)
- I proiettili dei troll si muovono da destra verso sinistra (`linearLeftMovement`)

Il sistema aggiorna la `PositionComponent` di ogni proiettile in base alla sua velocità e al tempo trascorso (`deltaTime`). Inoltre, il `MovementSystem` è anche responsabile di rimuovere i proiettili che

escono dai confini dello schermo, evitando l'accumulo di entità inutili.

RenderSystem

Il `RenderSystem` orchestra il processo di visualizzazione delle entità. Ad ogni ciclo, non ridisegna ciecamente tutto, ma implementa un'ottimizzazione per migliorare le performance:

- **Raccolta delle Entità**: Colleziona tutte le entità che possiedono sia un `PositionComponent` che un `ImageComponent`
- **Creazione di un Hash di Stato**: `generateStateHash` è una funzione pura che prende le sequenze immutabili di entità e barre vita e produce una stringa hash deterministica, usando `map` e `mkString`.
- **Confronto**: Confronta l'hash corrente con quello dell'ultimo frame renderizzato (`lastRenderedState`).

```
// In RenderSystem.scala
private def shouldRender(currentState: String): Boolean =
    !lastRenderedState.contains(currentState)
```

La decisione di renderizzare (`shouldRender`) è una semplice comparazione tra l'hash corrente e quello precedente. Se gli hash non corrispondono, avviene il rendering tramite il metodo `update`, che restituisce una nuova istanza del `RenderSystem` con `lastRenderedState` aggiornato, mantenendo l'immutabilità del sistema stesso. Questo approccio evita side effect (il rendering sulla UI) se non strettamente necessari.

Implementazione - Giovanni Pisoni

Panoramica dei contributi

Il mio contributo al progetto si è focalizzato sulle seguenti aree:

- **Architettura del `GameEngine`**: Definizione del `GameEngine` e del `GameState` per la gestione dello stato di gioco
- **Implementazione del `GameLoop`**: Creazione di un game loop a timestep fisso per garantire aggiornamenti consistenti
- **Implementazione del `GameController` e gestione degli eventi**: Sviluppo di un sistema di gestione degli eventi per disaccoppiare i componenti del sistema e implementazione del `GameController` per coordinare le interazioni tra i vari sistemi di gioco
- **Logica delle entità**: Implementazione del `MovementSystem` e dello `SpawnSystem` per il comportamento dei nemici, e dell'`HealthBarRenderSystem` per il rendering delle barre della salute delle entità di gioco
- **Interfaccia utente**: Sviluppo dei menu di pausa e delle schermate di vittoria/sconfitta
- **Testing**: Scrittura di test per i sistemi implementati, come `MovementSystemTest` e `SpawnSystemTest`

Principali sfide implementative

Durante lo sviluppo del progetto, ho affrontato diverse sfide tecniche significative che hanno richiesto un'attenta analisi e soluzioni innovative.

1. **Gestione della Concorrenza e Thread Safety**

   La prima sfida importante è stata garantire la thread-safety tra il game loop, che opera su un thread dedicato, e l'interfaccia ScalaFX, che richiede aggiornamenti sul JavaFX Application Thread. Ho risolto questa problematica utilizzando `AtomicReference` per lo stato dell'engine e del game loop, e una `ConcurrentLinkedQueue` per le azioni pendenti nel `GameController`. Questo approccio ha permesso di mantenere la separazione tra la logica di gioco e il rendering, evitando race conditions e deadlock.

2. **Transizione da movimento a celle a movimento continuo**

   Inizialmente, il `MovementSystem` utilizzava un approccio basato su celle discrete, dove le entità si teletrasportavano istantaneamente tra posizioni della griglia. Questo metodo, sebbene semplice da implementare, produceva un'esperienza visiva poco fluida. La transizione a un sistema di movimento basato su pixel ha richiesto una completa riprogettazione: ho ristrutturato la struttura `Position` con coordinate `Double`, implementato l'interpolazione del movimento basata sul `deltaTime`, e gestito le transizioni graduali per il movimento zigzag dei Troll Assassini. Questa modifica ha migliorato significativamente la qualità visiva del gioco, ma ha anche introdotto nuove complessità nella gestione delle collisioni e nel mapping tra coordinate logiche e fisiche.

3. **Complessità del sistema di spawn procedurale**

   Lo `SpawnSystem` ha presentato sfide nella sincronizzazione tra il timing di gioco e il tempo reale. La gestione della pausa richiedeva che gli eventi di spawn schedulati venissero "spostati nel tempo" di una durata pari alla pausa stessa. Ho implementato un meccanismo che traccia il momento della pausa (`pausedAt`) e, alla ripresa, ricalcola tutti i timestamp degli spawn pendenti.

Implementazione - GameEngine e GameLoop

Il cuore del gioco è rappresentato dal `GameEngine` e dal `GameLoop`, componenti che ho sviluppato per orchestrare l'intero flusso di gioco.

GameEngine

Il `GameEngine` è stato progettato come una macchina a stati finiti che gestisce le fasi principali del gioco (`MainMenu`, `Playing`, `Paused`, `GameOver`). L'engine è responsabile di:

- **Inizializzare e arrestare il gioco**: Avvia e ferma il `GameLoop` e gestisce il ciclo di vita del `GameController`
- **Gestire lo stato del gioco**: Mantiene il `GameState` corrente, che include la fase di gioco, il tempo trascorso e lo stato di pausa
- **Coordinare gli aggiornamenti**: Durante la fase `Playing`, invoca il metodo `update` del `GameController` per far avanzare la logica di gioco

L'immutabilità è un principio chiave: ogni modifica dello stato non altera l'oggetto corrente, ma ne crea una nuova istanza. Questo approccio funzionale previene effetti collaterali e semplifica la gestione dello stato.

```
case class GameState(
    phase: GamePhase = GamePhase.MainMenu,
    isPaused: Boolean = false,
    elapsedTime: Long = 0L,
    fps: Int = 0
):
  def transitionTo(newPhase: GamePhase): GameState = newPhase match
    case Paused  => copy(phase = newPhase, isPaused = true)
    case Playing => copy(phase = newPhase, isPaused = false)
    case other   => copy(phase = other)
```

Per garantire la thread-safety nelle operazioni concorrenti, l'aggiornamento dello stato del `GameEngine` utilizza un pattern atomico con compare-and-set e tail recursion.

GameLoop

Per garantire un'esperienza di gioco fluida e un comportamento deterministico, ho implementato un **game loop a timestep fisso**.Questo approccio disaccoppia la logica di gioco dalla velocità di rendering, assicurando che il gioco si comporti allo stesso modo su hardware diversi.

Il game loop a timestep fisso garantisce:

- **Determinismo**: il gioco si comporta identicamente su hardware diverso
- **Stabilità fisica**: la simulazione rimane coerente indipendentemente dal frame rate
- **Prevedibilità**: facilita il testing e il debugging del comportamento di gioco

Il `GameLoop` utilizza un `accumulator` per gestire il tempo trascorso tra i fotogrammi. La logica di gioco viene aggiornata in passi discreti di tempo fisso (`FRAME_TIME_MILLIS`), garantendo che, anche in caso di cali di frame rate, la simulazione di gioco progredisca correttamente.

La gestione del ciclo di aggiornamenti avviene attraverso il metodo `processAccumulatedFrames`. Questo metodo verifica se il tempo accumulato è sufficiente per eseguire un passo di aggiornamento della logica di gioco. In caso affermativo, invoca il metodo `update` del `GameEngine` e sottrae il tempo fisso dall'accumulatore.

```
@tailrec
private def processAccumulatedFrames(): Unit =
  val state = readState
  if state.hasAccumulatedTime(fixedTimeStep) && !engine.isPaused then
    engine.update(fixedTimeStep)
    updateState(_.consumeTimeStep(fixedTimeStep))
    processAccumulatedFrames()
```

Implementazione - GameController e gestione degli eventi

GameController e gestione degli Stati

Il `GameController` agisce come il principale orchestratore del gioco, facendo da ponte tra l'input dell'utente, la logica di gioco (i sistemi ECS) e il `GameEngine`. La sua responsabilità è quella di

tradurre le azioni del giocatore e gli eventi di sistema in aggiornamenti dello stato del mondo di gioco.

Per gestire la complessità dei numerosi sistemi che compongono la logica del gioco (movimento, combattimento, generazione di elisir, ecc.), ho introdotto la case class `GameSystemsState`. Questa classe incapsula lo stato di tutti i sistemi, garantendo che vengano aggiornati in un ordine predicibile e coerente.

Il metodo `updateAll` all'interno di `GameSystemsState` è cruciale: definisce la pipeline di esecuzione dei sistemi ad ogni ciclo di gioco. L'ordine di esecuzione è fondamentale: ad esempio, il `MovementSystem` viene eseguito prima del `CollisionSystem` per garantire che le collisioni vengano rilevate sulle nuove posizioni, e l'`HealthSystem` viene eseguito dopo, per applicare i danni risultanti. Questa struttura garantisce che le interdipendenze tra i sistemi siano gestite correttamente.

```
// in GameSystemsState.scala
def updateAll(world: World): (World, GameSystemsState) =
    val (world1, updatedElixir)    = elixir.update(world)
    val (world2, updatedMovement)  = movement.update(world1)
    val (world3, updatedCombat)    = combat.update(world2)
    val (world4, updatedCollision) = collision.update(world3)
    // ... e così via per gli altri sistemi
```

Il `GameController` mantiene un'istanza di `GameSystemsState` e la utilizza per evolvere lo stato del gioco. Inoltre, gestisce le azioni del giocatore, come il posizionamento dei maghi, verificando le condizioni necessarie (es. elisir sufficiente) e aggiornando lo stato di conseguenza in modo asincrono tramite una coda di azioni (`pendingActions`), per evitare problemi di concorrenza con il game loop.

Gestione degli Eventi

Per disaccoppiare i vari componenti del gioco e gestire le transizioni di stato in modo pulito e centralizzato, ho implementato un sistema di eventi. Questo sistema si basa su due componenti principali: `EventSystem` e `EventHandler`.

`EventSystem` definisce la gerarchia degli eventi di gioco (`GameEvent`) e una `EventQueue` immutabile. Gli eventi sono stati suddivisi per priorità, per garantire che le operazioni critiche (come `ExitGame`) vengano processate prima di altre.

```
// in EventSystem.scala
trait GameEvent:
  def priority: Int

object GameEvent:
  // System events
  case object ExitGame extends GameEvent:
    override def priority: Int = 0
  // Game State events
  case object GameWon extends GameEvent:
    override def priority: Int = 1
  // Menu events
  case object ShowMainMenu extends GameEvent:
    override def priority: Int = 2
  // Input events
  case class GridClicked(logicalPos: LogicalCoords, screenX: Int, screenY: Int) ↩
    extends GameEvent:
```

```
    override def priority: Int = 3
```

L'`EventHandler` è il cuore del sistema di eventi. È responsabile della gestione della coda di eventi e del dispatching degli stessi ai gestori appropriati. Utilizza un `AtomicReference` per gestire il suo stato (`EventHandlerState`) in modo thread-safe, un aspetto cruciale dato che gli eventi possono essere generati da thread diversi (es. il thread del game loop e il thread della UI).

Il metodo processEvent dell'`EventHandler` funge da macchina a stati finiti, gestendo le transizioni tra le varie fasi del gioco (`GamePhase`) in risposta a eventi specifici. Ad esempio, quando riceve un evento `ShowGameView`, non solo cambia la vista, ma avvia anche il GameEngine se non è già in esecuzione.

```
// in EventHandler.scala
  private def handleEvent(event: GameEvent): Unit =
    val state = stateRef.get()

    event match
      case ShowMainMenu =>
        handleMenuTransition(MainMenu, Some(ViewState.MainMenu))
        Option.when(isGameActive)(stopEngine())
      case ShowGameView =>
        handleMenuTransition(Playing, Some(ViewState.GameView))
        Option.when(!engine.isRunning)(startEngine())
      case Pause if state.currentPhase == Playing =>
        pauseEngine()
        handleMenuTransition(Paused, Some(ViewState.PauseMenu))
      // ... altri casi di eventi
```

Inoltre, la `EventQueue` è stata progettata per supportare operazioni composizionali. L'implementazione di metodi come `map` e `flatMap` permette di trasformare il flusso di eventi in modo dichiarativo, preservando l'immutabilità della coda e migliorando l'espressività del codice. Questo approccio funzionale si manifesta anche in metodi come `prioritize`, che ordina gli eventi in base alla loro criticità. Anziché iterare e riordinare manualmente la coda, questa operazione viene espressa come una singola trasformazione funzionale sulla collezione sottostante, garantendo che le operazioni più urgenti vengano eseguite per prime.

```
// Operazioni monadiche sulla coda di eventi
def map(f: GameEvent => GameEvent): EventQueue =
  copy(queue = queue.map(f))

def flatMap(f: GameEvent => Queue[GameEvent]): EventQueue =
  copy(queue = queue.flatMap(f))

def fold[B](initial: B)(f: (B, GameEvent) => B): B =
  queue.foldLeft(initial)(f)

// Prioritizzazione degli eventi
def prioritize(): EventQueue =
  copy(queue = Queue.from(queue.toList.sortBy(_.priority)))
```

Queste operazioni permettono di trasformare e comporre eventi in modo dichiarativo, mantenendo l'immutabilità della coda.

Questa architettura a eventi permette di avere un controllo centralizzato e prevedibile sul flusso del gioco, rendendo il sistema più robusto e facile da estendere con nuove funzionalità e interazioni.

Implementazione - Logica delle entità

La logica comportamentale delle entità nemiche, i troll, è stata implementata attraverso due sistemi dedicati all'interno dell'architettura Entity-Component-System (ECS): lo `SpawnSystem` e il `MovementSystem`. Questi moduli sono responsabili, rispettivamente, della generazione procedurale delle ondate di nemici e della gestione del loro comportamento di movimento sulla plancia di gioco.

SpawnSystem: Generazione Procedurale delle Ordate

Lo `SpawnSystem` orchestra la comparsa dei troll, introducendo una curva di difficoltà progressiva e un elemento di imprevedibilità. Una scelta progettuale chiave è stata quella di attivare il sistema solo dopo il posizionamento del primo mago da parte del giocatore. Questa decisione conferisce al giocatore il controllo sull'inizio effettivo della partita, permettendogli di stabilire una difesa iniziale prima di affrontare la prima ondata.

La generazione dei nemici è un processo dinamico e parametrico, governato da diverse logiche:

- **Difficoltà progressiva**: La sfida si intensifica con l'avanzare delle ondate. Lo `SpawnSystem` si interfaccia con il modulo di configurazione `WaveLevel` per applicare moltiplicatori alle statistiche base dei troll (salute, velocità, danno). Questo scaling assicura che la difficoltà aumenti in modo controllato e predicibile
- **Distribuzione dinamica dei nemici**: Per evitare la monotonia, la composizione delle ondate varia nel tempo. Le ondate iniziali sono dominate da troll di base, ma con il progredire della partita, il sistema introduce gradualmente tipologie di nemici più specializzate e complesse, come i `Warrior` o gli `Assassin`, seguendo una distribuzione di probabilità che si evolve a ogni nuova ondata
- **Generazione a "batch"**: Anziché generare i troll a intervalli perfettamente regolari, è stata implementata una logica di "batch". I nemici vengono generati in piccoli gruppi con intervalli temporali leggermente randomizzati. Questo approccio crea un flusso di avversari più organico e meno prevedibile, costringendo il giocatore ad adattare costantemente le proprie strategie difensive

```
private def generateSpawnBatch(currentTime: Long, firstRow: Option[Int], ↩
  numOfSpawns: Int): List[SpawnEvent] =
  val isFirstBatch = pendingSpawns.isEmpty && firstRow.isDefined
  List.tabulate(numOfSpawns): index =>
    val useFirstRow = isFirstBatch && index == 0
    generateSingleSpawn(currentTime + index * BATCH_INTERVAL, useFirstRow, firstRow)
```

Come si evince dal codice, il primo troll di un'ondata viene sempre generato sulla stessa riga del primo mago posizionato, una scelta implementativa per focalizzare l'azione iniziale nel punto in cui il giocatore ha deciso di stabilire la sua prima linea di difesa.

MovementSystem: Strategie di movimento dei Troll

Una volta che un'entità è stata generata, il suo comportamento spaziale è governato dal `MovementSystem`. Durante lo sviluppo, questo sistema ha subito un'importante evoluzione.

Inizialmente, il movimento era basato su una logica a celle, dove le entità si spostavano istantaneamente da una cella della griglia all'altra. Questo approccio, sebbene semplice da implementare, risultava visivamente "scattoso" e poco realistico.

Per migliorare la fluidità e la qualità visiva del gioco, si è deciso di passare a un sistema di movimento basato su pixel. Questa transizione ha richiesto la definizione di una struttura dati `Position` che rappresenta le coordinate `(x, y)` nello spazio di gioco con valori `Double`, consentendo spostamenti frazionari e quindi animazioni più fluide.

Il `MovementSystem` è stato quindi riprogettato per aggiornare la `PositionComponent` di ogni entità mobile a ogni ciclo del game loop, calcolando lo spostamento in base alla velocità dell'entità e al `deltaTime` (il tempo trascorso dall'ultimo frame). Questo sistema applica diverse strategie di movimento in base alla tipologia dell'entità, secondo un'implementazione del **Strategy Pattern**:

1. **Movimento lineare**: La maggior parte dei troll implementa una strategia di movimento lineare, avanzando da destra verso sinistra con una velocità definita nel loro `MovementComponent`. Questo comportamento costituisce il fondamento della sfida tattica del gioco, richiedendo un posizionamento strategico delle entità difensive per intercettare l'avanzata nemica

```
private val linearLeftMovement: MovementStrategy = (pos, movement, _, _, dt) =>
    val pixelsPerSecond = movement.speed * CELL_WIDTH
    val minY            = GRID_OFFSET_Y
    val maxY            = GRID_OFFSET_Y + GRID_ROWS * CELL_HEIGHT - CELL_HEIGHT / 2
    Position(
      pos.x - pixelsPerSecond * dt,
      pos.y
    )
```

2. **Movimento a zigzag**: Per introdurre una maggiore complessità tattica, è stata implementata una strategia di movimento non lineare per il `Troll Assassino`. Questa entità alterna il proprio percorso tra la corsia di generazione e una corsia adiacente, scelta in modo pseudocasuale. Questo comportamento a zigzag lo rende un bersaglio più elusivo, obbligando il giocatore a considerare un posizionamento difensivo più flessibile. Per implementare questo comportamento `stateful` (il troll deve "ricordare" in quale fase del movimento si trova e da quanto tempo) mantenendo il `MovementSystem` completamente `stateless` è stato introdotto lo `ZigZagStateComponent`. Questo componente agisce come una piccola macchina a stati associata a ogni singolo Troll Assassino, contenendo informazioni come la riga di spawn, la riga alternativa, la fase corrente del movimento (`OnSpawnRow` o `OnAlternateRow`) e il timestamp di inizio della fase. In questo modo, il `MovementSystem` non deve mantenere alcuno stato interno; ad ogni update, legge semplicemente lo `ZigZagStateComponent` dell'entità per calcolarne la nuova posizione, garantendo che ogni assassino gestisca il proprio ciclo di zigzag in modo indipendente e che la logica di movimento rimanga pura e disaccoppiata

```
private val zigzagMovement: MovementStrategy = (pos, movement, entity, world, dt) =>
    world.getComponent[ZigZagStateComponent](entity) match
      case Some(state) =>
        val currentTime = System.currentTimeMillis()
        val updatedPos  = calculateZigZagPosition(pos, movement.speed, dt, state, ↩
  currentTime)

        updatedPos
      case None =>
        pos
```

Il `MovementSystem` gestisce anche l'interazione con altri sistemi attraverso il sistema a componenti. Ad esempio, la presenza di un `FreezedComponent` su un troll, applicato dal `CollisionSystem` a seguito di un attacco di ghiaccio, viene rilevata dal MovementSystem per modificare dinamicamente la velocità dell'entità. Questo disaccoppiamento tra la logica del movimento e gli effetti di stato, facilitato dall'architettura ECS, ha permesso di implementare interazioni complesse tra entità in modo modulare e manutenibile.

HealthBarRenderSystem: Feedback visivo sullo stato di salute delle entità

Per fornire al giocatore un feedback visivo immediato sullo stato di salute delle entità in gioco, ho implementato l'`HealthBarRenderSystem`. Questo sistema si integra nel ciclo di rendering principale e ha la responsabilità di disegnare le barre della vita sopra le entità che hanno subito danni.

L'implementazione segue un approccio orientato all'efficienza e alla separazione delle responsabilità, operando in diverse fasi all'interno del suo metodo `update`:

1. **Raccolta dati**: Il sistema per prima cosa interroga il `World` per identificare tutte le entità che possiedono un `HealthComponent`. Per ciascuna di queste entità, raccoglie le informazioni necessarie per il rendering: la posizione, la percentuale di salute corrente e il `HealthBarComponent` associato. Una scelta implementativa importante è stata quella di fornire un comportamento di default: se un'entità con salute non ha un `HealthBarComponent` esplicito, il sistema ne crea uno al volo, differenziando il colore della barra in base al tipo di entità (verde per i maghi, rosso per i troll). Questo garantisce la coerenza visiva e semplifica la creazione delle entità, che non devono necessariamente essere definite con un componente per la barra della vita
2. **Calcolo dei parametri di Rendering**: Successivamente, i dati raccolti vengono elaborati per calcolare i parametri esatti per il rendering. Questo include l'aggiornamento del colore della barra in base alla percentuale di salute (verde per salute alta, giallo per media, rosso per bassa), una logica incapsulata all'interno del `HealthBarComponent` stesso per mantenere il componente coeso e responsabile del proprio stato visivo
3. **Filtro di visibilità**: Una delle ottimizzazioni chiave del sistema è il filtraggio delle barre della vita. Per evitare di disegnare elementi non necessari e ridurre l'overhead di rendering, vengono renderizzate solo le barre delle entità la cui salute è compresa tra 0% e 100% (esclusi). Le entità con salute piena o quelle sconfitte non mostrano la barra della vita, mantenendo l'interfaccia pulita e focalizzata sulle informazioni rilevanti per il giocatore

```
// in HealthBarRenderSystem.scala
private def filterVisibleBars(bars: Map[EntityId, RenderableHealthBar]): ↩
Map[EntityId, RenderableHealthBar] =
  bars.filter { case (_, (_, percentage, _, _, _, _)) => percentage < 1.0 && ↩
percentage > 0.0 }
```

4. **Caching e Rendering**: Infine, i dati delle barre visibili vengono memorizzati in una cache (`healthBarCache`) all'interno dello stato del sistema. Questa cache viene poi passata al `RenderSystem` principale, che si occupa del disegno effettivo degli elementi a schermo. L'uso di una cache interna permette di disaccoppiare la logica di calcolo delle barre dalla loro effettiva visualizzazione

Questo approccio garantisce che il feedback visivo sullo stato di salute sia non solo informativo, ma anche performante, contribuendo a un'esperienza di gioco fluida anche in presenza di un numero

elevato di entità a schermo.

Interfaccia Utente

Oltre alla logica di gioco, il mio contributo si è esteso all'implementazione di componenti cruciali dell'interfaccia utente, in particolare i menu di overlay che gestiscono le interruzioni del flusso di gioco. Questi elementi sono stati sviluppati utilizzando **ScalaFX**, adottando un approccio funzionale e dichiarativo per la costruzione della UI.

Menu di Pausa e Schermate di Vittoria/Sconfitta

Ho sviluppato i pannelli `PauseMenu` e `GameResultPanel` per gestire, rispettivamente, la messa in pausa del gioco da parte dell'utente e la conclusione di un'ondata o della partita.

La progettazione di questi componenti si è basata su alcuni principi chiave:

- **Componibilità e riuso**: Entrambi i pannelli condividono una struttura simile, basata su uno `StackPane` che sovrappone un layout di controlli a un'immagine di sfondo. La creazione dei bottoni e la gestione delle loro azioni sono state delegate a un `ButtonFactory` centralizzato, che traduce le interazioni dell'utente in `GameEvent` specifici (es. `ResumeGame`, `ContinueBattle`, `NewGame`). Questo approccio ha permesso di ridurre la duplicazione del codice e di mantenere una netta separazione tra la vista e la logica di controllo
- **Gestione dichiarativa degli stati**: Per il `GameResultPanel`, ho utilizzato un **Algebraic Data Type (ADT)**, definito tramite una `sealed trait`, per modellare i due possibili esiti della partita: `Victory` e `Defeat`. Questa scelta progettuale permette di rappresentare gli stati in modo type-safe ed estensibile. Ogni `case object` incapsula le informazioni specifiche per quel determinato stato, come l'immagine del titolo da mostrare e l'azione da associare al pulsante di continuazione

```
sealed trait ResultType:
  def titleImagePath: String
  def continueButtonText: String
  def continueAction: ButtonAction

case object Victory extends ResultType:
  val titleImagePath               = "/victory.png"
  val continueButtonText           = "Next wave"
  val continueAction: ButtonAction = ContinueBattle

case object Defeat extends ResultType:
  val titleImagePath               = "/defeat.png"
  val continueButtonText           = "New game"
  val continueAction: ButtonAction = NewGame
```

Questo pattern non solo rende il codice più leggibile e manutenibile, ma garantisce anche che il pannello si adatti correttamente al contesto, offrendo azioni pertinenti all'utente (ad esempio, "Prossima ondata" dopo una vittoria e "Nuova partita" dopo una sconfitta)

Per migliorare la robustezza dell'interfaccia in queste schermate, è stata implementata una logica di debouncing per i pulsanti "Next wave" e "New game". Questa modifica previene comportamenti anomali o race condition che potrebbero verificarsi a causa di click ripetuti e rapidi da parte dell'utente. La soluzione, implementata nel `GameResultPanel` attraverso il metodo

`createDebouncedButton`, utilizza un `AtomicBoolean` per garantire che l'azione associata al pulsante venga eseguita una sola volta. Una volta premuto, il pulsante viene temporaneamente disabilitato per una breve durata (definita dalla costante DEBOUNCE_MS), impedendo l'invio di eventi `ContinueBattle` o `NewGame` duplicati all'`EventHandler`. Questo accorgimento assicura una transizione di stato pulita e controllata, gestita dal `GameController`.

Per ottimizzare le performance, il caricamento delle immagini di sfondo e dei titoli è stato implementato utilizzando `lazy val`. In questo modo, le risorse grafiche vengono caricate dal disco solo al momento del loro primo utilizzo effettivo, riducendo il tempo di avvio e il consumo di memoria dell'applicazione. La logica di caricamento e caching è stata incapsulata nell'`ImageFactory`, promuovendo ulteriormente il riuso del codice.

In sintesi, l'implementazione di questi componenti dell'interfaccia utente ha seguito i principi della programmazione funzionale e della separazione delle responsabilità, portando a un codice modulare, efficiente e facilmente estensibile.

Testing e Validazione

La validazione della correttezza e della robustezza del software è stata una componente integrante del processo di sviluppo. Sebbene non sia stata adottata una metodologia strettamente **Test-Driven Development (TDD)**, i test sono stati scritti in modo sistematico parallelamente o immediatamente dopo l'implementazione di ogni funzionalità. Questo approccio ha permesso di garantire la stabilità del codice, facilitare le fasi di refactoring e prevenire l'introduzione di regressioni.

Per la stesura e l'esecuzione dei test è stato utilizzato **ScalaTest**, un framework ampiamente diffuso nell'ecosistema Scala, che ha permesso di scrivere test chiari e leggibili.

Domain-Specific Language (DSL) per Scenari di Test

Una delle sfide principali nel testare un'applicazione complessa come un videogioco, specialmente uno basato sull'architettura ECS, è la configurazione dello stato iniziale per ogni scenario di test. La creazione manuale di entità, l'aggiunta di componenti e l'impostazione dei parametri di gioco possono risultare verbose, ripetitive e difficili da leggere, oscurando l'intento effettivo del test.

Per superare questa difficoltà, è stato progettato e implementato un **Domain-Specific Language (DSL)** interno, specifico per la creazione di scenari di gioco.

Il DSL si basa su un `ScenarioBuilder` che offre una serie di metodi concatenabili per definire lo stato del gioco in modo fluido:

```
// in GameScenarioDSL.scala
def scenario(setup: ScenarioBuilder => Unit): (World, GameSystemsState) =
  val builder = ScenarioBuilder()
  setup(builder)
  builder.build()

class ScenarioBuilder:
  // ...
  def withWizard(wizardType: WizardType): WizardPlacer = ...
  def withTroll(trollType: TrollType): TrollPlacer = ...
  def withElixir(amount: Int): this.type = ...
  def atWave(waveNumber: Int): this.type = ...
```

```
  // ...
```

Questo permette di scrivere test estremamente concisi e focalizzati sul comportamento da verificare, come dimostrato nell'esempio seguente:

Senza DSL (verboso e poco leggibile):

```
val world = World.empty
val (world1, wizard) = world.createEntity()
val world2 = world1.addComponent(wizard, WizardTypeComponent(WizardType.Fire))
val world3 = world2.addComponent(wizard, PositionComponent(pos))
val world4 = world3.addComponent(wizard, HealthComponent(100, 100))
// ... 10+ linee simili
```

Con DSL (dichiarativo e chiaro):

```
val (testWorld, _) = scenario: builder =>
  builder
    .withWizard(WizardType.Fire).at(GRID_ROW_MID, GRID_COL_START)
    .withTroll(TrollType.Basic).at(GRID_ROW_MID, GRID_COL_END)
    .withElixir(ELIXIR_START)
```

Copertura dei Test

Sono state create suite di test per tutti i principali moduli logici del gioco, garantendo una solida copertura delle funzionalità critiche.

I test coprono circa il 75% del codice implementato, con una copertura del 90% per i componenti core del `GameEngine` e dei sistemi ECS. I componenti UI non sono stati testati in quanto basati su ScalaFX, framework che richiede test di integrazione complessi. In particolare, sono stati testati:

- **`GameEngineTest` e `GameLoopTest`**: Verificano la corretta gestione del ciclo di vita del gioco (avvio, arresto, pausa, ripresa) e la stabilità del ciclo di aggiornamento a timestep fisso
- **`MovementSystemTest`**: Assicura che le diverse strategie di movimento (lineare, zigzag) vengano applicate correttamente e che gli effetti di stato (come il rallentamento) modifichino il comportamento delle entità come previsto
- **`SpawnSystemTest`**: Valida la logica di generazione dei nemici, controllando il rispetto dei tempi, il numero massimo di troll per ondata e l'applicazione corretta dello scaling di difficoltà
- **`GameSystemsStateTest`**: Verifica le transizioni di stato del gioco e la corretta rilevazione delle condizioni di vittoria e sconfitta

L'approccio al testing adottato si è dimostrato efficace nel garantire la qualità e la robustezza del codice, costituendo una rete di sicurezza indispensabile durante l'intero ciclo di sviluppo del progetto. Anche se riflettendo sul processo, un'adozione del Test-Driven Development (TDD) avrebbe potuto portare ulteriori benefici. Questo avrebbe potuto ridurre alcune delle sessioni di refactoring e portare a un design ancora più pulito e disaccoppiato. Inoltre, avrebbe fornito una guida più strutturata per l'implementazione, trasformando i requisiti in casi di test eseguibili che avrebbero definito in modo inequivocabile il comportamento atteso di ogni modulo

Implementazione - Giovanni Rinchiuso

Panoramica dei Contributi

Il mio contributo al progetto si è focalizzato sulle seguenti aree:

- **Sistemi di gioco**: `ElixirSystem`, `HealthSystem`, gestione economia e salute delle entità
- **Configurazione e bilanciamento**: `WaveLevel`, calcolo parametri ondate e distribuzione troll
- **Sistema di input**: `InputProcessor`, `InputSystem`, `InputTypes` con validazione
- **Interfaccia utente**: `InfoMenu`, `ShopPanel`, `WavePanel` con gestione stato reattiva
- **Testing**: DSL per `ElixirSystemTest`, `HealthSystemTest`, `InputProcessorTest`, `InputSystemTest`

Gestione dell'Economia: ElixirSystem

L'elisir è la risorsa centrale del gioco, necessaria per acquistare maghi e difendersi dai troll. Ho implementato `ElixirSystem` come sistema immutabile che gestisce la generazione periodica di elisir, la produzione dai maghi generatori, e le transazioni di spesa.

Il sistema è implementato come case class immutabile che estende il trait `System`, integrandosi così nell'architettura ECS del gioco. Ogni operazione restituisce una nuova istanza del sistema, garantendo che lo stato sia sempre consistente.

```
case class ElixirSystem(
    totalElixir: Int = INITIAL_ELIXIR,
    lastPeriodicGeneration: Long = 0L,
    firstWizardPlaced: Boolean = false,
    activationTime: Long = 0L
) extends System
```

L'aggiunta di elisir è una funzione pura che restituisce un nuovo sistema senza modificare quello esistente:

```
def addElixir(amount: Int): ElixirSystem =
  copy(totalElixir = Math.min(totalElixir + amount, MAX_ELIXIR))
```

La spesa di elisir utilizza `Option.when` per validare la transazione, restituendo sia il nuovo stato che un booleano di successo:

```
def spendElixir(amount: Int): (ElixirSystem, Boolean) =
  Option.when(totalElixir >= amount):
    copy(totalElixir = totalElixir - amount)
  .map((_, true))
    .getOrElse((this, false))
```

Questo approccio rende lo stato del gioco consistente.

L'aggiornamento del system utilizza `Option.when` per gestire la logica condizionale:

```
override def update(world: World): (World, System) =
  Option.when(firstWizardPlaced):
    val periodicSystem = updatePeriodicElixirGeneration()
    periodicSystem.updateGeneratorWizardElixir(world)
  .getOrElse((world, this))
```

Se il primo mago non è stato ancora piazzato, il sistema semplicemente restituisce lo stato attuale senza eseguire alcuna elaborazione. Questo pattern elimina la necessità di statement `if-else` espliciti, rendendo il codice più dichiarativo.

La generazione periodica utilizza `Option` per gestire l'inizializzazione e i controlli temporali:

```
private def updatePeriodicElixirGeneration(): ElixirSystem =
  val currentTime = System.currentTimeMillis()
  Option.when(lastPeriodicGeneration == 0L):
    copy(
      lastPeriodicGeneration = currentTime,
      activationTime = Option.when(activationTime == ↩
    0L)(currentTime).getOrElse(activationTime)
    )
  .orElse:
    checkAndGenerateElixir(currentTime)
  .getOrElse(this)
```

Il pattern `orElse` permette di concatenare logiche alternative: se è la prima generazione, inizializza i timestamp; altrimenti, controlla se è il momento di generare elisir.

Per l'elaborazione dei maghi generatori, ho utilizzato for-comprehension per validare le condizioni in sequenza, fermandosi alla prima che fallisce:

```
private def processGeneratorEntity(
    world: World,
    entityId: EntityId,
    currentTime: Long,
    system: ElixirSystem
): (World, ElixirSystem) =
  (for
    wizardType      <- world.getComponent[WizardTypeComponent](entityId)
    _               <- Option.when(wizardType.wizardType == WizardType.Generator)(())
    elixirGenerator <- world.getComponent[ElixirGeneratorComponent](entityId)
  yield processGeneratorCooldown(world, entityId, currentTime, elixirGenerator, system))
    .getOrElse((world, system))
```

Questa implementazione verifica che:

1. L'entità abbia un componente `WizardTypeComponent`
2. Il tipo di mago sia effettivamente `Generator`
3. L'entità abbia un componente `ElixirGeneratorComponent`

Se una qualsiasi di queste verifiche fallisce, il for-comprehension termina e restituisce lo stato originale tramite `getOrElse`. Questo approccio è molto più sicuro e leggibile rispetto a una serie di statement `if` annidati.

Per processare tutti i maghi generatori nel mondo, utilizzo `foldLeft` per accumulare i cambiamenti attraverso tutte le entità:

```
private def updateGeneratorWizardElixir(world: World): (World, ElixirSystem) =
  val currentTime       = System.currentTimeMillis()
  val generatorEntities = world.getEntitiesWithTwoComponents[WizardTypeComponent, ↩
    ElixirGeneratorComponent].toList
```

```
  generatorEntities.foldLeft((world, this)): (acc, entityId) =>
    val (currentWorld, currentSystem) = acc
    processGeneratorEntity(currentWorld, entityId, currentTime, currentSystem)
```

Il `foldLeft` accumula sia il `World` aggiornato che l'`ElixirSystem` aggiornato, propagando lo stato attraverso l'elaborazione di ogni entità. Questo pattern è utile nella programmazione funzionale per gestire sequenze di trasformazioni mantenendo l'immutabilità.

Gestione della Salute: HealthSystem

`HealthSystem` è responsabile della gestione delle collisioni, dei danni, della morte delle entità e delle ricompense.

L'update del sistema segue un pattern di pipeline funzionale, dove ogni fase trasforma lo stato e lo passa alla successiva:

```
override def update(world: World): (World, System) =
  val (world1, system1) = processCollisionComponents(world)
  val (world2, system2) = system1.processDeaths(world1)
  val (world3, system3) = system2.removeDeadEntities(world2)
  (world3, system3)
```

Ogni funzione nella pipeline:

1. Riceve il mondo e il sistema correnti
2. Esegue una trasformazione specifica
3. Restituisce il nuovo mondo e sistema

Questo approccio garantisce che ogni fase sia isolata e testabile indipendentemente, seguendo il principio di Single Responsibility (SRP).

Per processare tutte le entità con componenti di collisione, utilizzo `foldLeft` per accumulare i cambiamenti:

```
private def processCollisionComponents(world: World): (World, HealthSystem) =
  world.getEntitiesWithComponent[CollisionComponent]
    .foldLeft((world, this)): (acc, entityId) =>
      val (currentWorld, currentSystem) = acc
      currentWorld.getComponent[CollisionComponent](entityId)
        .map: collision =>
          val worldWithoutCollision = ↩
    currentWorld.removeComponent[CollisionComponent](entityId)
          currentSystem.applyCollisionToEntity(worldWithoutCollision, entityId, ↩
    collision)
        .getOrElse(acc)
```

Il pattern utilizzato qui combina `foldLeft` con `map` su `Option`: per ogni entità, tentiamo di ottenere il componente di collisione. Se presente, applichiamo il danno e rimuoviamo il componente; altrimenti, manteniamo lo stato corrente. Questo evita la necessità di controlli null o eccezioni.

L'applicazione del danno utilizza `Option` e `filter` per validare lo stato dell'entità prima di applicare modifiche:

```
private def applyCollisionToEntity(
    world: World,
```

```
    entityId: EntityId,
    collisionComp: CollisionComponent
): (World, HealthSystem) =
  world.getComponent[HealthComponent](entityId)
    .filter(_.isAlive)
    .map: healthComp =>
      val newHealth     = math.max(0, healthComp.currentHealth - collisionComp.amount)
      val newHealthComp = healthComp.copy(currentHealth = newHealth)
      val updatedWorld  = updateHealth(world, entityId, newHealthComp)
      handlePossibleDeath(updatedWorld, entityId, newHealthComp)
    .getOrElse((world, this))
```

Il `filter(_.isAlive)` garantisce che il danno venga applicato solo alle entità vive, mentre il pattern `map-getOrElse` gestisce l'assenza del componente senza eccezioni.

Il calcolo delle ricompense utilizza pattern matching per mappare i tipi di troll alle ricompense appropriate. Questo approccio è più sicuro e leggibile rispetto a una serie di if-else:

```
private def calculateElixirReward(world: World, entityId: EntityId): Int =
  world.getComponent[TrollTypeComponent](entityId)
    .map(_.trollType)
    .map:
      case TrollType.Base     => BASE_TROLL_REWARD
      case TrollType.Warrior  => WARRIOR_TROLL_REWARD
      case TrollType.Assassin => ASSASSIN_TROLL_REWARD
      case TrollType.Thrower  => THROWER_TROLL_REWARD
    .getOrElse(0)
```

Se l'entità non è un troll (non ha `TrollTypeComponent`), restituisce 0. Il compilatore Scala verifica che tutti i casi siano gestiti, prevenendo bug a runtime.

Per identificare le entità morte, utilizzo for-comprehension con filtri multipli:

```
private def getNewlyDeadEntities(world: World): List[EntityId] =
  for
    entityId <- world.getEntitiesWithComponent[HealthComponent].toList
    if !entitiesToRemove.contains(entityId)
    health <- world.getComponent[HealthComponent](entityId).toList
    if !health.isAlive
  yield entityId
```

In questo modo:

1. Itera su tutte le entità con `HealthComponent`
2. Filtra quelle non già marcate per rimozione
3. Estrae il componente salute
4. Filtra quelle non vive

Il risultato è una lista di entità che sono morte ma non ancora rimosse. La sintassi for-comprehension rende la logica molto più chiara rispetto a una catena di `filter` e `flatMap`.

Configurazione e Bilanciamento: WaveLevel

`WaveLevel` è l'oggetto che gestisce la progressione della difficoltà attraverso le ondate di troll. Determina quindi, come il gioco diventa progressivamente più sfidante mantenendo un equilibrio

tra sfida e giocabilità.

`WaveLevel` deve risolvere diverse problematiche:

- **Varietà progressiva**: nelle prime ondate appaiono solo troll base, mentre ondate successive introducono gradualmente nemici più specializzati e pericolosi
- **Distribuzione probabilistica**: ogni ondata ha una specifica composizione di tipi di troll, definita tramite probabilità che determinano la frequenza di apparizione di ciascun tipo
- **Scalabilità**: i parametri dei troll (salute, velocità, danno) aumentano con le ondate per rendere il gioco sempre più sfidante
- **Bilanciamento**: gli intervalli di spawn diminuiscono progressivamente, aumentando la pressione sul giocatore

Pattern Matching per Distribuzione Troll

La distribuzione dei tipi di troll cambia progressivamente con le ondate. Ho utilizzato pattern matching con guards per definire le distribuzioni di probabilità:

```
def calculateTrollDistribution(wave: Int): Map[TrollType, Double] =
  wave match
    case w if w <= 1 =>
      Map(
        TrollType.Base     -> 1.0,
        TrollType.Warrior  -> 0.0,
        TrollType.Assassin -> 0.0,
        TrollType.Thrower  -> 0.0
      )
    case w if w <= 2 =>
      Map(
        TrollType.Base     -> 0.7,
        TrollType.Warrior  -> 0.3,
        TrollType.Assassin -> 0.0,
        TrollType.Thrower  -> 0.0
      )
    case w if w <= 3 =>
      Map(
        TrollType.Base     -> 0.5,
        TrollType.Warrior  -> 0.3,
        TrollType.Assassin -> 0.2,
        TrollType.Thrower  -> 0.0
      )
    case w if w <= 4 =>
      Map(
        TrollType.Base     -> 0.4,
        TrollType.Warrior  -> 0.3,
        TrollType.Assassin -> 0.2,
        TrollType.Thrower  -> 0.1
      )
    case _ =>
      Map(
        TrollType.Base     -> 0.3,
        TrollType.Warrior  -> 0.3,
        TrollType.Assassin -> 0.25,
        TrollType.Thrower  -> 0.15
```

```
)
```

Ogni pattern definisce una distribuzione di probabilità che determina quali tipi di troll appaiono in quella fase del gioco. Questo approccio offre numerosi vantaggi in termini di leggibilità: osservando i pattern, la progressione della difficoltà emerge naturalmente, mostrando come nelle prime ondate dominino i troll base per poi introdurre gradualmente le varianti più pericolose. L'estensibilità è altrettanto semplice: se volessimo aggiungere nuovi livelli di difficoltà, basterebbe inserire ulteriori case senza toccare la logica esistente.

Il compilatore verifica automaticamente che tutti i casi siano gestiti, e il case `_` finale garantisce un fallback sicuro per tutte le ondate oltre la quarta. Inoltre, ogni Map restituita è immutabile e la funzione è completamente pura, senza side-effects.

L'uso di guards nel pattern matching (`if w <= 1`, `if w <= 2`, etc.) Permette di definire range di ondate piuttosto che valori singoli, rendendo la configurazione più flessibile rispetto a un approccio basato su uguaglianza esatta. Ad esempio, tutte le ondate dalla quinta in poi usano la stessa distribuzione finale, che rappresenta il massimo livello di difficoltà del gioco.

Selezione Pesata con FoldLeft

Per selezionare un tipo di troll random basato sulla distribuzione di probabilità, ho implementato un algoritmo di selezione pesata utilizzando `foldLeft`:

```
def selectRandomTrollType(distribution: Map[TrollType, Double]): TrollType =
  val random = scala.util.Random.nextDouble()
  distribution
    .toSeq
    .sortBy(_._2)
    .foldLeft((0.0, Option.empty[TrollType])) {
      case ((cumulative, Some(selected)), _) =>
        (cumulative, Some(selected))
      case ((cumulative, None), (trollType, probability)) =>
        val newCumulative = cumulative + probability
        if random <= newCumulative then (newCumulative, Some(trollType))
        else (newCumulative, None)
    }
    ._2
    .getOrElse(TrollType.Base)
```

L'algoritmo implementa la tecnica della "roulette wheel selection":

1. Genera un numero random tra 0 e 1
2. Accumula le probabilità usando `foldLeft`, creando segmenti cumulativi
3. Quando la somma cumulativa supera il valore random, seleziona quel tipo
4. Restituisce il tipo selezionato, con fallback a `TrollType.Base`

Il pattern matching nei case del `foldLeft` implementa un "early exit" funzionale:

```
case ((cumulative, Some(selected)), _) => (cumulative, Some(selected))
```

Una volta che un tipo è stato selezionato (l'Option diventa `Some`), questo pattern mantiene la selezione ignorando tutte le iterazioni successive.

Il caso alternativo:

```
case ((cumulative, None), (trollType, probability)) =>
  val newCumulative = cumulative + probability
  if random <= newCumulative then (newCumulative, Some(trollType))
  else (newCumulative, None)
```

aggiorna la somma cumulativa e verifica se il valore random cade in questo "segmento" della roulette. Se sì, avviene la selezione; altrimenti, continua ad accumulare. Infine, `._2.getOrElse(TrollType.Base)` estrae il tipo selezionato dall'Option, fornendo un fallback sicuro nel caso improbabile che nessun tipo venga selezionato (ad esempio, se tutte le probabilità fossero 0).

Sistema di Input: InputProcessor, InputSystem e InputTypes

Ho implementato un'architettura a tre livelli che separa le responsabilità nella gestione degli input dell'utente. Questa struttura garantisce una chiara separazione delle responsabilità, facilita il testing e rende il sistema facilmente estendibile.

1. InputTypes

Il livello più basso dell'architettura definisce i tipi di dato fondamentali utilizzati nel sistema di input. `MouseClick` rappresenta un evento di click del mouse con coordinate `(x, y)`, mentre `ClickResult` incapsula il risultato della validazione di un click, contenendo la posizione, un flag di validità e un messaggio di errore opzionale.

2. InputProcessor

Il livello intermedio è responsabile della logica di validazione vera e propria. Implementa metodi per verificare se un click cade all'interno dei bounds della griglia, convertire coordinate dello schermo in posizioni di gioco e validare le posizioni risultanti. La separazione tra processamento e validazione permette di testare facilmente la logica di validazione in isolamento.

3. InputSystem

Il livello più alto fornisce un'interfaccia per l'utilizzo del sistema. Utilizza `InputProcessor` internamente nascondendo i dettagli implementativi e offre metodi come `handleMouseClick` che accettano coordinate dello schermo e restituiscono un `ClickResult`. Inoltre, fornisce metodi utility come `processClicks` per elaborare batch di click, `validPositions` per filtrare solo le posizioni valide, e `partitionClicks` per separare click validi e invalidi.

ClickResult

Un elemento centrale di questa implementazione è `ClickResult`, che ho implementato seguendo il pattern delle monadi per comporre validazioni. Questo approccio permette di concatenare multiple validazioni.

`ClickResult` incapsula il risultato di un click del mouse, memorizzando la posizione, un flag di validità e un messaggio di errore opzionale. Ho implementato le tre operazioni monadiche fondamentali (`map`, `flatMap` e `filter`).

L'operazione `map` permette di trasformare la posizione contenuta se il risultato è valido, lasciando inalterati i risultati invalidi. `flatMap` consente di concatenare validazioni che a loro volta producono

`ClickResult`, implementando così il pattern della "railway-oriented programming" dove un errore in qualsiasi punto della catena cortocircuita le operazioni successive. `filter` aggiunge la capacità di validare predicati sulla posizione, convertendo un risultato valido in invalido se il predicato fallisce.

```
result
  .map(pos => pos.normalize())
  .filter(_.isInBounds, "Out of bounds")
  .flatMap(pos => validateCell(pos))
```

Ogni operazione nella catena viene eseguita solo se quella precedente ha avuto successo, e il primo fallimento propaga automaticamente attraverso tutta la catena senza bisogno di controlli espliciti.

Composizione di Predicati di Validazione

Per semplificare l'applicazione di multiple validazioni, ho implementato un metodo `validate` nel companion object di `ClickResult` che accetta un numero variabile di predicati con i loro messaggi di errore associati:

```
def validate(pos: Position)(validations: (Position => Boolean, String)*): ClickResult =
  validations.foldLeft(valid(pos)): (result, validation) =>
    result.filter(validation._1, validation._2)
```

Questo metodo utilizza `foldLeft` per applicare sequenzialmente tutte le validazioni fornite. Ogni validazione è una tupla contenente un predicato (una funzione `Position => Boolean`) e un messaggio di errore. Il risultato iniziale è un `ClickResult` valido contenente la posizione, che viene poi trasformato applicando ogni validazione in sequenza tramite `filter`.

Un esempio di utilizzo all'interno di `InputProcessor`:

```
def processClickWithValidation(click: MouseClick): ClickResult =
  val position = click.toPosition
  ClickResult.validate(position)(
    (_.isValid, "Position is not valid"),
    (_ => isInGridArea(click.x, click.y), "Click outside grid area")
  )
```

In questo esempio, la posizione viene validata contro due predicati: prima si verifica che la posizione sia valida in sé, poi si controlla che cada all'interno dell'area della griglia. Se una qualsiasi validazione fallisce, il `ClickResult` diventa invalido con il messaggio di errore appropriato, e le validazioni successive vengono comunque eseguite (anche se il loro risultato viene ignorato) per completare il fold.

Extension Methods in MouseClick

Ho utilizzato le extension methods di Scala 3 per arricchire il tipo `MouseClick` con metodi di validazione, rendendo l'API più fluente e intuitiva:

```
extension (click: MouseClick)
  def validate(processor: InputProcessor): ClickResult =
    processor.processClick(click)

  def isInGrid(processor: InputProcessor): Boolean =
    processor.isInGridArea(click.x, click.y)
```

```
  def validateWith(processor: InputProcessor)(errorMsg: String): ClickResult =
    processor.processClick(click) match
      case result if result.isValid => result
      case _                        => ClickResult.invalid(errorMsg)
```

Il metodo `validate` delega al `InputProcessor` per eseguire la validazione standard. `isInGrid` fornisce un controllo booleano per verificare se il click cade nell'area della griglia. `validateWith` permette di personalizzare il messaggio di errore, usando pattern matching per sostituire eventuali errori di default con un messaggio custom.

L'utilizzo permette di concatenare operazioni in modo naturale:

```
val click = MouseClick(x, y)
click.validate(processor)
  .filter(_.isInCell, "Not in cell")
  .map(_.toGridCoordinates)
```

Questa catena di operazioni valida il click, filtra per verificare che sia in una cella, e trasforma le coordinate. Se qualsiasi passo fallisce, l'errore si propaga automaticamente e il risultato finale sarà un `ClickResult` invalido con il messaggio di errore appropriato.

Interfaccia Utente: InfoMenu, ShopPanel e WavePanel

Ho sviluppato diversi componenti dell'interfaccia utente che costituiscono l'esperienza visiva e interattiva del gioco.

InfoMenu

L'`InfoMenu` fornisce al giocatore informazioni dettagliate sulle meccaniche di gioco, sui diversi tipi di maghi e sui vari tipi di troll. La sua implementazione si basa su una struttura a tab che permette di navigare tra diverse sezioni informative: regole del gioco, caratteristiche dei maghi e caratteristiche dei troll.

La gestione dello stato della navigazione è implementata attraverso una closure che mantiene riferimenti ai bottoni di navigazione e aggiorna dinamicamente la loro opacità per indicare quale sezione è attualmente attiva. Quando l'utente clicca su un tab, il contenuto dell'area centrale viene sostituito con la vista appropriata e l'opacità dei bottoni viene aggiornata per riflettere lo stato corrente.

Per maghi e troll vengono mostrate delle card informative contenenti le statistiche (salute, danno, costo/ricompensa) e l'immagine rappresentativa di ogni tipologia.

ShopPanel

Lo `ShopPanel` è il componente dell'interfaccia che permette al giocatore di acquistare i maghi durante la partita. Questo pannello mostra tutte le tipologie di maghi disponibili, con le loro icone e i rispettivi costi in elisir.

La caratteristica principale dello `ShopPanel` è la sua capacità di aggiornarsi dinamicamente in base alla quantità di elisir posseduta dal giocatore. Quando l'elisir aumenta o diminuisce, il pannello ricalcola automaticamente quali maghi sono acquistabili e aggiorna il loro aspetto visivo di

conseguenza: i maghi acquistabili vengono resi interattivi con effetti hover e cursor a mano, mentre quelli non acquistabili vengono disabilitati visivamente con opacità ridotta e bordi grigi.

Lo stato del pannello è modellato attraverso una struttura dati immutabile che mantiene l'ammontare corrente di elisir, lo stato di apertura/chiusura del pannello e una mappa che associa ogni tipo di mago al suo stato di disponibilità. Questa architettura garantisce che il pannello rimanga sempre sincronizzato con lo stato del gioco, fornendo al giocatore un feedback visivo immediato sulle opzioni di acquisto disponibili.

WavePanel

Il `WavePanel` mostra informazioni sull'ondata corrente e si aggiorna automaticamente quando il gioco progredisce.

Lo stato del pannello mantiene l'ultimo numero di ondata renderizzato (per evitare aggiornamenti ridondanti), un riferimento opzionale al componente `Text` che mostra il numero e un riferimento opzionale al pannello stesso. Il metodo `updateWaveNumber` garantisce che l'interfaccia venga aggiornata solo quando il numero dell'ondata effettivamente cambia, evitando rendering inutili e migliorando le performance.

Testing

Ho sviluppato test per i principali sistemi di cui mi sono occupato: ElixirSystem, HealthSystem, InputProcessor e InputSystem. Anche se non ho seguito rigorosamente il Test-Driven Development, ho scritto i test in modo sistematico parallelamente o immediatamente dopo l'implementazione di ogni funzionalità. Per semplificare la scrittura dei test e renderli più leggibili, ho sviluppato quattro DSL specializzati per testare i sistemi implementati, utilizzando pattern funzionali per garantire immutabilità e type-safety.

ElixirSystemTest

L'`ElixirSystem` richiede la gestione di timing, generazione periodica e interazioni con il mondo di gioco. Ho quindi, progettato un DSL che mantiene lo stato attraverso `Option`, permettendo di memorizzare valori tra le diverse fasi del test senza ricorrere a variabili mutabili.

Questo esempio di test mostra come il DSL renda espressivo il testing temporale:

```
"ElixirSystem" should "generate elixir from generator wizards" in {
  givenAnElixirSystem
    .activated
    .withWorld
    .andGeneratorWizardAt(Position(2, 3))
    .rememberingInitialElixir
    .afterWaiting(GENERATOR_WIZARD_COOLDOWN + ELIXIR_WAIT_MARGIN)
    .whenUpdated
    .shouldHaveAtLeast(PERIODIC_ELIXIR).moreElixirThanInitial
}
```

Ho implementato una enum `ComparisonType` e una case class `ElixirAmountComparison` che permettono di esprimere asserzioni come `shouldHaveAtLeast(50).moreElixirThanInitial` o `shouldHaveExactly(100).moreElixirThanInitial`:

```
enum ComparisonType:
  case AtLeast, Exactly

case class ElixirAmountComparison(dsl: ElixirSystemDSL, amount: Int, comparisonType: ↩
    ComparisonType):
  def moreElixirThanInitial: ElixirSystemDSL =
    dsl.initialElixir.foreach: initial =>
      val diff = dsl.system.getCurrentElixir - initial
      comparisonType match
        case ComparisonType.AtLeast => diff should be >= amount
        case ComparisonType.Exactly => diff shouldBe amount
    dsl
```

HealthSystemTest

In `HealthSystem` i test devono creare entità, applicare danni e verificare sia lo stato di salute che le ricompense. Ho modellato queste operazioni attraverso tre case class che rappresentano diverse fasi del test.

Il flusso tipico di un test si presenta così:

```
"HealthSystem" should "kill entity and reward elixir" in {
  aHealthSystem
    .withTroll(TrollType.Base)
    .havingHealth(50, 100)
    .takingDamage(60)
    .done
    .whenUpdated
    .entity(0).shouldBeDead
    .systemShouldHaveElixir(INITIAL_ELIXIR + BASE_TROLL_REWARD)
}
```

Le tre case class che compongono il DSL sono `HealthSystemDSL` per il contesto principale, `EntityBuilder` per la configurazione delle entità, e `EntityAssertions` per le verifiche. Le transizioni avvengono attraverso i tipi di ritorno: chiamare `withEntity` restituisce un `EntityBuilder` che permette di configurare l'entità, `done` riporta al contesto principale, e `entity(n)` fornisce un `EntityAssertions` per le verifiche.

Questa struttura sfrutta il sistema di tipi di Scala per prevenire errori a compile-time. Ad esempio, non è possibile verificare lo stato di un'entità prima di averla configurata, perché il compilatore non permetterebbe di chiamare `entity(0)` prima di aver chiamato `done`.

L'accumulo delle entità avviene in modo immutabile: ogni operazione restituisce una nuova istanza del DSL con il world aggiornato e l'entità aggiunta alla lista attraverso `entities :+ entity`:

```
def withTroll(trollType: TrollType): EntityBuilder =
  val (updatedWorld, entity) = world.createEntity()
  val worldWithComponent = updatedWorld.addComponent(entity, ↩
    TrollTypeComponent(trollType))
  EntityBuilder(this.copy(world = worldWithComponent, entities = entities :+ entity), ↩
    entity)
```

InputProcessorTest e InputSystemTest

Per i sistemi di input ho progettato DSL che separano la fase di setup delle coordinate dalla fase di verifica dei risultati.

Un esempio di test:

```
"InputProcessor" should "validate grid coordinates" in {
  aClick
    .atOffset(10, 10)
    .whenProcessed
    .shouldBeValid
    .andShouldBeInCell
}
```

La struttura si basa su due case class: `ClickBuilder` accumula le coordinate, mentre `ClickResultAssertions` gestisce le verifiche. Il metodo `whenProcessed` fa da ponte tra le due fasi, eseguendo la validazione e restituendo le asserzioni.

Testing

Approccio

Considerata la natura del gioco — basato su un'architettura ECS (Entity-Component-System) e su logiche di aggiornamento continue — è stato fondamentale garantire l'affidabilità del codice attraverso test unitari e di integrazione.

L'obiettivo è stato quello di mantenere un'elevata qualità del codice sin dalle prime fasi di sviluppo, scrivendo i test **in parallelo all'implementazione** delle funzionalità e assicurando che ogni parte del sistema fosse verificabile in modo indipendente.

Tecnologie utilizzate

Per la scrittura e l'esecuzione dei test è stato utilizzato **ScalaTest**.
Le principali caratteristiche sfruttate includono:

- **Suite modulari**: una suite di test per ogni componente logico (Engine, ECS, Game Logic, Rendering)
- **Matchers espressivi**: per una sintassi più leggibile e semantica rispetto ai semplici `assert`
- **Test isolati**: grazie alla progettazione funzionale e immutabile, ogni test può essere eseguito senza dipendenze da stato globale

Grado di copertura

Tutte le principali funzionalità del gioco sono coperte da test automatici.
In particolare, sono stati testati:

- **Game engine core**: gestione dello stato, aggiornamenti e game loop
- **Entity & Component System**: creazione, rimozione e interazioni tra entità
- **Logica di gioco**: movimento e comportamento dei troll, attacchi e abilità dei maghi, generazione dell'elisir, condizioni di vittoria/sconfitta
- **Sistema dell'elisir e progressione**: bilanciamento risorse, costi e progressione delle ondate
- **Gestione eventi e collisioni**: eventi interni e interazioni tra entità

I test sono stati eseguiti regolarmente durante tutto il ciclo di sviluppo, assicurando:

- **Correttezza logica**
- **Robustezza rispetto a input non validi**
- **Stabilità e coerenza tra moduli** dopo ogni refactoring

Retrospettiva

Analisi del processo di sviluppo e dello stato attuale

Il processo di sviluppo adottato ha garantito una buona organizzazione e coordinazione tra i membri del team. Complessivamente, siamo soddisfatti del processo adottato in quanto le scadenze settimanali sono state per lo più soddisfatte.

La criticità maggiore riscontrata è stata la suddivisione dei task negli sprint, in modo da garantire un carico di lavoro equo e permettere lo sviluppo indipendente tra i membri, soprattutto nelle fasi iniziali del progetto. L'architettura ECS (Entity Component System) scelta ha richiesto un periodo di adattamento iniziale per comprendere appieno le interazioni tra entità, componenti e sistemi. Tuttavia, una volta acquisita familiarità con questo pattern, lo sviluppo è proceduto in modo più fluido ed efficiente.

Questa metodologia ci ha permesso di gestire le tempistiche in modo accurato e i frequenti confronti hanno permesso di evitare incongruenze o ambiguità. L'adozione di un approccio funzionale con Scala ha facilitato la gestione dello stato immutabile del gioco, riducendo significativamente i bug legati alla concorrenza e agli effetti collaterali.

Migliorie e lavori futuri

Le funzionalità principali previste sono state tutte realizzate: cinque tipi di wizard e quattro tipi di troll con comportamenti differenziati, oltre a un'interfaccia utente completa con menu, shop e indicatori di gioco. Alcune migliorie future potrebbero riguardare l'interfaccia grafica, che sebbene funzionale potrebbe essere ulteriormente arricchita, e le funzionalità opzionali rimaste in sospeso: l'inserimento di ostacoli nella mappa, l'aggiunta di colpi speciali o potenziamenti per i wizard, e l'implementazione di nuove mappe con layout diversi.

Data la struttura modulare del progetto, potrebbe essere molto semplice inserire in futuro ulteriori difficoltà di gioco e strategie, in modo da rendere l'esperienza più stimolante. L'aggiunta di nuovi tipi di entità, boss fight a fine wave, o un sistema di upgrade persistente tra le partite potrebbero aumentare significativamente la rigiocabilità del gioco.

Conclusioni

In conclusione, il progetto Wizards vs Trolls ha rappresentato un'ottima occasione per sperimentare concretamente tecniche e processi di sviluppo studiati durante il corso. Inoltre, ha permesso di affrontare la progettazione del software con un approccio differente, a partire dalle prime fasi fino alla conclusione, ponendo l'attenzione più sulla metodologia e sulla qualità del codice che sulla realizzazione di grandi funzionalità. L'utilizzo di Scala e del paradigma funzionale ci ha spinto a ragionare in termini di immutabilità e composizione, portando a un codice più robusto e manutenibile.

Durante lo sviluppo, abbiamo integrato la scrittura dei test in parallelo all'implementazione del codice. Questo approccio, sebbene non seguisse il ciclo TDD in modo rigoroso, si è rivelato prezioso: ci ha spinto a chiarire i requisiti in anticipo e ha garantito una maggiore correttezza del software. In particolare, la creazione di una DSL custom per i test di scenario ha migliorato notevolmente la leggibilità e l'efficacia delle verifiche sulle meccaniche di gioco.

Tuttavia, questo metodo ha presentato delle sfide. Poiché i test venivano scritti insieme al codice, a volte risultavano strettamente legati ai dettagli implementativi, richiedendo ristrutturazioni quando il codice veniva sottoposto a refactoring.

Con il senno di poi, un'applicazione più formale del TDD avrebbe probabilmente ridotto la necessità di riscrivere i test, guidando un design più stabile fin dall'inizio. Nonostante le difficoltà, l'esperienza è stata un'importante lezione sul valore di una metodologia di test rigorosa e ci ha fornito una maggiore consapevolezza per i progetti futuri.