Tesina Finale di
Programmazione di Interfacce Grafiche e Dispositivi Mobili
Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2024-2025
Dipartimento di Ingegneria

docente Prof. Luca Grilli

JHelifire

applicazione desktop JFC/SWING

Studenti:

342589	**Giovanni Ciocchetti**	giovanni.ciocchetti@studenti.unipg.it
357782	**Giulia Matracchi**	giulia.matracchi@studenti.unipg.it

Sommario

1 Descrizione del problema

L'obiettivo di questo progetto è lo sviluppo di un'applicazione desktop, denominata *JHeliFire*: una replica che rivisita il gameplay del videogioco arcade *HeliFire*. L'applicazione sarà implementata utilizzando la tecnologia JFC/Swing, scelta appositamente per garantire un'elevata portabilità e compatibilità con diversi sistemi operativi e piattaforme.

Di seguito, verrà presentata una breve descrizione del videogioco originale HeliFire, evidenziandone le caratteristiche principali e gli elementi di gameplay che si intendono semplificare. Successivamente, si illustrerà la versione *JHeliFire* che si intende realizzare, specificando in che modo verranno adattate e semplificate tali caratteristiche per il prototipo.

1.1 Il videogioco HeliFire

HeliFire è un classico videogioco arcade sviluppato da Nintendo R&D1 e pubblicato da Nintendo nel 1980. Il titolo si inserisce nel genere degli sparatutto arcade e si distingue per il suo design minimalista ma estremamente coinvolgente. Con una grafica raster colorata e un gameplay frenetico, HeliFire rappresenta un punto di riferimento dell'era arcade, in cui la semplicità dei controlli nasconde una sfida elevata e un ritmo incalzante.

Caratteristiche principali:

- Livelli veloci e difficili: i livelli di HeliFire sono progettati per essere estremamente dinamici. Con uno scorrimento rapido dello schermo, il gioco aumenta progressivamente la difficoltà, richiedendo al giocatore di compiere movimenti rapidi e precisi per schivare attacchi continui;
- Sincronizzazione musicale: la colonna sonora e gli effetti sonori sono strettamente integrati con l'azione di gioco. La musica si sincronizza con il ritmo frenetico delle ondate di nemici, contribuendo a creare un'atmosfera tesa e immersiva durante le fasi più critiche;
- Sfide ed obiettivi: il giocatore deve affrontare non solo l'attacco degli elicotteri nemici, ma anche altre minacce che provengono dalle profondità marine. Il gameplay prevede obiettivi specifici, come eliminare un certo numero di nemici o superare particolari schemi di attacco, premiando la precisione e il tempismo;
- Design arcade: la semplicità dei controlli e l'interfaccia immediata sono tipici dei giochi arcade dell'epoca, rendendo HeliFire accessibile e avvincente sin dal primo utilizzo. La sfida costante, combinata con un sistema di punteggio che premia la rapidità e l'accuratezza, contribuisce a mantenere alta la tensione durante tutta la partita.

Gameplay

Nel gameplay di HeliFire il giocatore assume il controllo di un sottomarino situato nella parte inferiore dello schermo.

- Movimento e schivata: il giocatore deve spostare il sottomarino in maniera fluida per evitare gli attacchi provenienti dall'alto, principalmente bombe e proiettili lanciati dagli

elicotteri nemici. La capacità di muoversi rapidamente in più direzioni è fondamentale per sopravvivere in ambienti in rapido movimento.
- Contrattacco: pur essendo in modalità difensiva, il sottomarino dispone di un'arma per contrattaccare. Il tempismo è essenziale: colpire i nemici in anticipo può impedire loro di generare ulteriori minacce, contribuendo a mantenere il ritmo e a incrementare il punteggio.
- Progressione e aumento della difficoltà: con il passare dei livelli, il gioco introduce una maggiore quantità di nemici e una velocità di scorrimento più elevata. Questa progressione costante non solo intensifica la sfida, ma richiede anche un continuo adattamento delle strategie da parte del giocatore, che deve gestire simultaneamente attacchi multipli e mantenere il controllo preciso del proprio veicolo.
- Esperienza Arcade completa: la combinazione di grafica semplice, effetti sonori diretti e una sfida costante fa di HeliFire un classico arcade.

1.2 L'applicazione *JHeliFire*

JHeliFire è un'applicazione desktop sviluppata in Java con la tecnologia JFC/Swing che propone una rivisitazione di un classico arcade. Ispirato al videogioco originale HeliFire, *JHeliFire* offre un'esperienza di gioco retrò, caratterizzata da un gameplay frenetico e accessibile.

Caratteristiche principali del gioco

- Player: il giocatore controlla un sottomarino agile, il cui design minimalista è studiato per garantire movimenti fluidi e reattivi lungo entrambi gli assi (orizzontale e verticale). I controlli sono semplificati per permettere una rapida apprendimento, consentendo al giocatore di schivare ostacoli e attacchi nemici con precisione e rapidità.
- Stile grafico: *JHeliFire* adotta uno stile grafico retrò in linea con l'estetica dei classici arcade. La grafica enfatizza la semplicità e la chiarezza visiva, garantendo una resa efficace su schermi moderni.
- Livelli: il gioco è organizzato in più livelli, ognuno caratterizzato da una progressione di difficoltà. Ad ogni livello la velocità di scorrimento e la quantità di nemici aumentano, imponendo al giocatore una maggiore concentrazione e abilità.
- Sistema di vite: in *JHeliFire* il giocatore dispone di un numero limitato di vite, che rappresentano il margine di errore consentito durante la partita. Ogni volta che il sottomarino subisce un colpo o collida con un nemico, viene sottratta una vita.

2 Specifica dei Requisiti

Di seguito viene presentata una lista dei requisiti del sistema software *JHeliFire*, i quali servono per riprodurre il gioco arcade HeliFire. I requisiti verranno suddivisi in due categorie: requisiti funzionali, che descrivono cosa deve fare il sistema, e requisiti non funzionali, che specificano le caratteristiche generali e le prestazioni del sistema.

2.1 Requisiti funzionali

I requisiti funzionali definiscono le funzionalità specifiche che il sistema deve fornire per soddisfare le esigenze del giocatore e riprodurre le dinamiche del gioco HeliFire.

1. Controllo del sottomarino:
    - Consentire al giocatore di muovere il sottomarino a sinistra, a destra, in basso e in alto entro una zona definita.
    - Permettere allo sparo tramite un tasto dedicato (es. la barra spaziatrice).
    - Visualizzare e gestire un numero predefinito di vite (3)

2. Generazione delle Ondate di Nemici:
    - Generare ondate di nemici suddivise in due tipologie:
        - Nemici d'aria: elicotteri di diverso tipo che sparano proiettili e bombe.
        - Nemici di mare: navi che possono andare a collidere con il player (il sottomarino).
    - Il sistema deve generare gradualmente queste ondate, introducendo progressivamente le due tipologie e aumentando la difficoltà (maggiore numero di nemici, velocità più elevata e frequenza di attacchi aumentata) man mano che il giocatore avanza nei livelli.

3. Rilevamento delle Collisioni:
    - Rilevare le collisioni tra i proiettili del giocatore e i nemici, tra i proiettili dei nemici e l'astronave del giocatore, e tra il giocatore e i nemici.
    - Gestire le collisioni.
    - In caso di collisione, aggiornare il punteggio o sottrarre una vita al giocatore, e attivare una fase di invulnerabilità temporanea per evitare danni immediati successivi.

4. Sistema di Punteggio e Classifica:
    - Assegnare punti al giocatore per ogni nemico distrutto, con valori differenti a seconda della tipologia di nemico
    - Visualizzare in tempo reale il punteggio
    - Salvare e visualizzare l'HighScore
    - Salvare i punteggi in un file di testo per la visualizzazione di una classifica dei migliori punteggi.

5. Gestione del Game Over:
    - La partita termina quando il giocatore esaurisce tutte le vite.
    - Al game over, mostrare una schermata che evidenzi il punteggio ottenuto e offra la possibilità di riprovare o tornare al menu.

6. Audio e grafica:
    • Integrare effetti sonori per spari, esplosioni, e altri eventi di gioco.
    • Includere una colonna sonora che accompagni l'esperienza di gioco.
    • Fornire un comando per attivare/disattivare l'audio (mute/unmute).
    • Visualizzare uno sfondo dinamico, con elementi in movimento, per migliorare l'immersività.

7. Pausa e ripresa:
    • Consentire al giocatore di mettere in pausa e riprendere il gioco tramite un apposito comando.

8. Grafica Bidimensionale Semplice
    • Il gioco deve presentare una grafica bidimensionale semplice, simile a quella del gioco arcade originale.

2.2 Requisiti non funzionali

I requisiti non funzionali descrivono le proprietà generali che il sistema deve rispettare per garantire un'esperienza di gioco piacevole e performante.

- Fluidità del gioco: il sistema deve garantire un'esperienza di gioco fluida e reattiva, mantenendo un frame rate costante di 60 FPS. Questa prestazione è fondamentale per evitare rallentamenti o scatti che possano compromettere la precisione dei controlli e l'immersione dell'utente.

- Compatibilità piattaforme: *JHeliFire* deve essere eseguibile su qualsiasi sistema dotato di una Java Virtual Machine (JVM), assicurando così la piena compatibilità con Windows, macOS e Linux. Ciò permette agli utenti di diverse piattaforme di accedere al gioco senza particolari limitazioni hardware.

- Espandibilità: il progetto deve essere strutturato in modo modulare, adottando un'architettura orientata agli oggetti che faciliti l'aggiunta di nuove funzionalità in futuro. L'implementazione di moduli separati (per la gestione grafica, audio, input, ecc.) consente di estendere il gioco senza necessità di una completa ristrutturazione del codice.

- Portabilità del codice: il codice sorgente deve essere scritto seguendo gli standard Java e le best practice di progettazione, per garantire che sia facilmente comprensibile, mantenibile e portabile. Questo approccio agevolerà il lavoro in team e favorirà eventuali modifiche o migrazioni su altre piattaforme.

- Sicurezza e robustezza: il sistema deve gestire in maniera appropriata eventuali errori e malfunzionamenti, implementando un efficace meccanismo di gestione delle eccezioni. In questo modo, il gioco eviterà arresti anomali e potrà continuare a funzionare in condizioni di errore, garantendo una maggiore affidabilità e una migliore esperienza utente.

3. Progetto

Viene ora presentata l'architettura dell'applicazione *JHeliFire*. Si illustra inizialmente la struttura delle cartelle e successivamente l'organizzazione del software, per poi scendere nella descrizione delle principali componenti che costituiscono il sistema.

3.1 Architettura del Sistema Software

L'applicazione *JHeliFire* è stata progettata con una suddivisione modulare in pacchetti, volta a organizzare in modo chiaro e funzionale le diverse componenti del software. Questa struttura facilita la comprensione del codice, la manutenzione e l'eventuale estensione del progetto. Ogni pacchetto è dedicato a un ambito specifico, garantendo un'organizzazione ordinata e una netta separazione delle responsabilità.

Il sistema si articola nei seguenti pacchetti principali:

- *controller/:* gestisce la logica del gioco e il coordinamento delle varie componenti. Le classi al suo interno si occupano di ricevere gli input dell'utente e di aggiornare lo stato del gioco;

- *model/:* rappresenta lo stato interno del gioco e le entità dinamiche che lo popolano, come il giocatore, i nemici, i proiettili, le esplosioni e gli oggetti bonus;

- *view/:* si occupa della rappresentazione grafica della partita e dell'interfaccia utente;

- *utility/:* fornisce servizi ausiliari come la gestione dell'audio e dei punteggi salvati, che supportano il funzionamento complessivo del gioco;

- *Main.java*: costituisce il punto di ingresso dell'applicazione e si occupa di inizializzare e collegare tra loro le principali componenti del sistema;

La struttura delle cartelle del progetto riflette questa organizzazione:

- *src/:* contiene il codice sorgente suddiviso nei pacchetti sopra descritti;
- *assets/:* contiene tutte le risorse multimediali utilizzate nel gioco, suddivise in tre sotto-cartelle:
    - figure/: sprite e immagini per le entità di gioco;
    - font/: font personalizzati per i testi a schermo;
    - sounds/: file audio per musica ed effetti sonori;

- *dist/: contiene il file* highscores.txt, file di salvataggio utilizzato per la memorizzazione dei punteggi ottenuti dai giocatori, ed il file eseguibile del gioco.

La figura sottostante mostra una rappresentazione grafica delle cartelle:

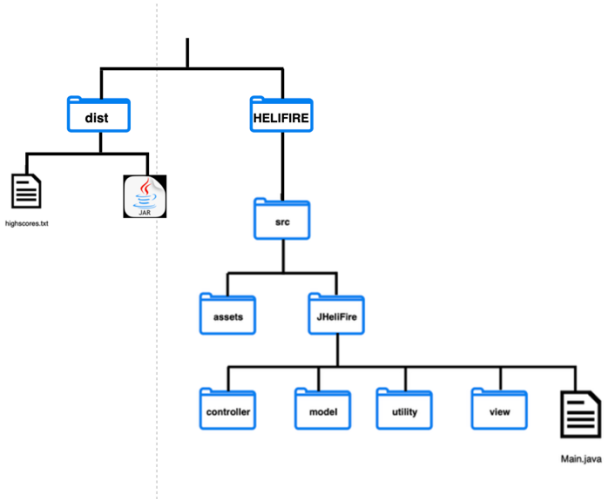


Fig. 1 Rappresentazione grafica delle cartelle

Le interazioni tra i pacchetti avvengono secondo un flusso ben definito: il controller riceve e interpreta gli input dell'utente, aggiorna il modello, e infine la vista provvede a rappresentare graficamente lo stato aggiornato del gioco. Le classi di utilità sono richiamate quando necessario per operazioni trasversali, come la riproduzione dei suoni o la gestione della classifica. Questa struttura ordinata contribuisce a garantire un buon livello di coesione interna ai pacchetti e un basso grado di dipendenza tra loro.

Di seguito, viene riportato il diagramma UML rappresentante l'interazione tra i pacchetti e le classi principali:

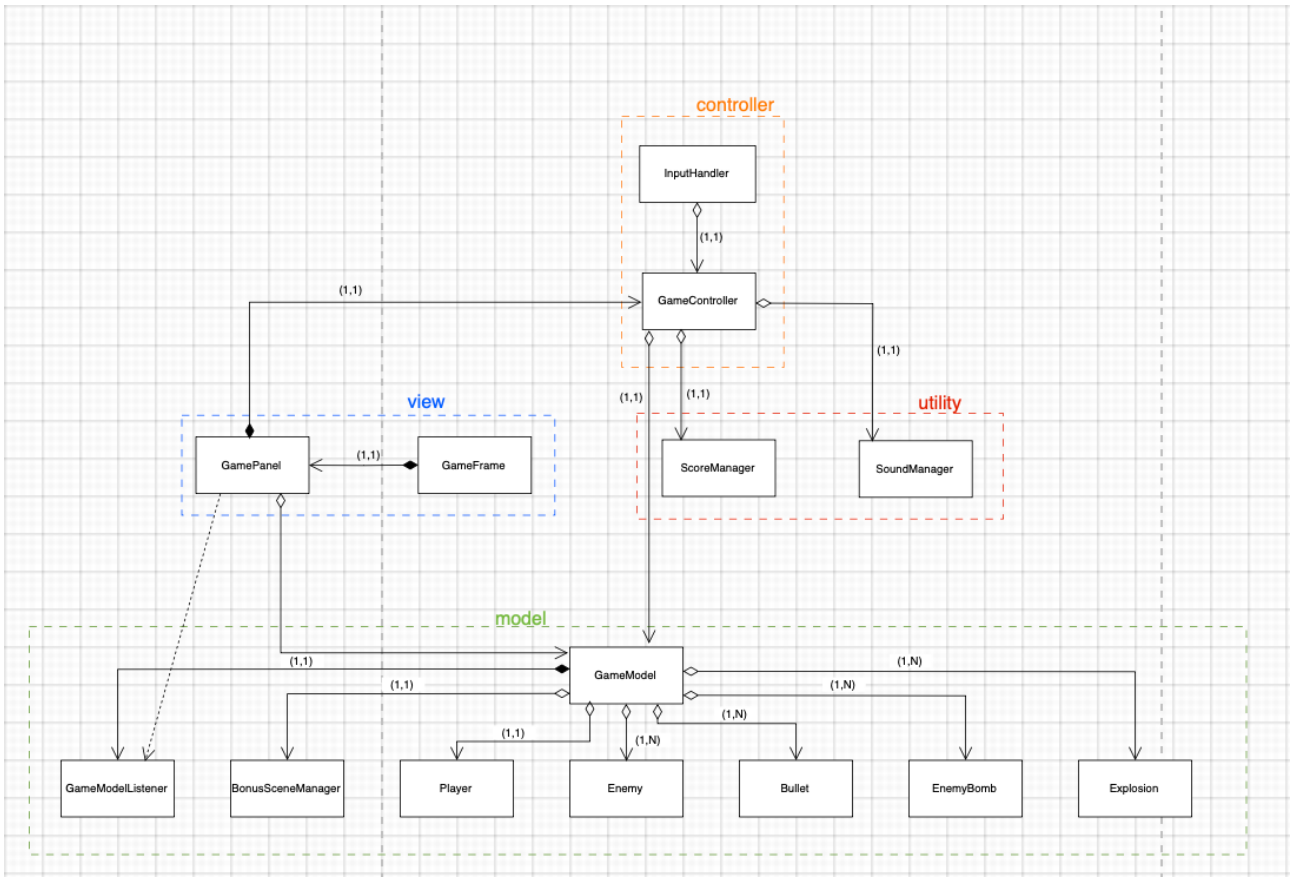


Fig. 2 Diagramma UML di alto livelli

3.2 Descrizione dei pacchetti

Nella seguente sezione vengono descritti i package che costituiscono il software e le loro classi.

3.2.1 Main

La classe Main rappresenta il punto di ingresso dell'applicazione JHeliFire. All'interno del metodo main(String[] args) vengono create e collegate le componenti principali del gioco. Di seguito si descrivono nel dettaglio le istruzioni presenti:

- **GameModel model = new GameModel(GamePanel.WIDTH, GamePanel.HEIGHT);**
  Viene creata un'istanza della classe GameModel, inizializzata con le dimensioni predefinite della finestra di gioco, rappresentate dalle costanti WIDTH e HEIGHT della classe GamePanel. GameModel gestisce lo stato del gioco, le entità dinamiche e la logica di avanzamento.

- **GamePanel panel = new GamePanel(model, null);**
  Viene creata un'istanza di GamePanel, il componente grafico principale che disegna il gioco e gestisce il rendering. In questa fase il controller viene passato come null, perché verrà assegnato successivamente.

- **GameController controller = new GameController(model, panel);**
  Viene creata un'istanza di GameController, che si occupa della gestione della logica di gioco e del controllo degli input da parte dell'utente. Il controller riceve come parametri il modello e il pannello del gioco, così da poter interagire con entrambi.

- **panel.setController(controller);**
  Viene associato il controller all'istanza di GamePanel, completando il collegamento tra le componenti principali.

- **panel.setEnabled(true); e panel.setFocusable(true);**
  Queste istruzioni abilitano il pannello e lo rendono pronto a ricevere il focus per la gestione degli input da tastiera.

- **new GameFrame(panel);**
  Viene creato un oggetto GameFrame, la finestra principale dell'applicazione, che riceve come parametro il pannello di gioco e lo contiene al suo interno. La gestione della visibilità e delle proprietà della finestra avviene all'interno della classe GameFrame.

- **panel.requestFocusInWindow();**
  Viene richiesto il focus per il pannello, così da permettere l'acquisizione degli input da tastiera fin dall'avvio del gioco.

Il metodo main si limita quindi a inizializzare e collegare tra loro le componenti fondamentali dell'applicazione, delegando la gestione degli aspetti grafici e della logica interna alle classi specifiche.

3.2.2 controller

Il package controller contiene le classi responsabili della gestione della logica di gioco e dell'elaborazione degli input da tastiera. In particolare, include GameController, che coordina l'avanzamento del gioco e gli eventi principali, e InputHandler, che intercetta e gestisce i comandi dell'utente durante la partita.

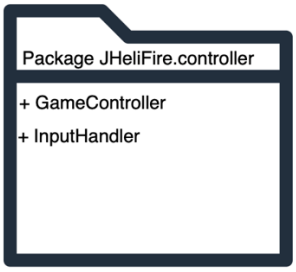


Fig. 3 Classi del package controller

GameController

La classe GameController gestisce il flusso e la logica del gioco JHeliFire, coordinando le interazioni tra il GameModel, che contiene lo stato e le regole del gioco, e il GamePanel, che si occupa del rendering e dell'interfaccia grafica. Il costruttore collega il controller al modello e alla vista, imposta lo stato iniziale su START_SCREEN e registra il controller come listener per gli eventi sonori generati dal modello.
Il metodo principale della classe è onTick(), richiamato a ogni ciclo del timer principale. Quando il gioco si trova nello stato GAME_PLAY, questo metodo aggiorna la logica chiamando model.update(), verifica lo stato del giocatore tramite checkPlayerState() e aggiorna lo sfondo con updateBackground(). In caso di stato BONUS_CUTSCENE, onTick() aggiorna la scena bonus richiamando il BonusManager dal GamePanel e gestisce la transizione al termine della sequenza bonus.
La gestione del bonus avviene in più fasi. Quando il modello segnala che un bonus è pronto (isBonusReady()), il controller disattiva il flag, interrompe la musica di sottofondo e, se l'audio non è silenziato, avvia il loop audio della scena bonus. Viene quindi avviato un timer che, dopo un secondo, cambia lo stato del gioco in BONUS_CUTSCENE. Durante questo stato, il BonusManager si occupa di aggiornare la scena bonus. Quando il bonus è completato (isComplete()), il controller richiama onBonusComplete(), che assegna il bonus al giocatore in base al numero di vite: se il giocatore ha tutte le vite, ottiene punti aggiuntivi; altrimenti, recupera una vita persa. Il controller ripristina poi la musica di fondo e lo stato torna a GAME_PLAY, pronto per il livello successivo.
Oltre alla gestione del bonus, il GameController coordina altri eventi chiave. Il metodo checkPlayerState() verifica se il giocatore è ancora vivo, e in caso contrario avvia handleGameOver(), che tramite un timer mostra la schermata di fine partita o la schermata per l'inserimento del nome se il punteggio rientra nella top 3. Se invece il modello segnala la condizione di vittoria (isInVictory()), viene avviato handleVictory() con una logica simile. Entrambe le situazioni aggiornano lo stato e richiedono il ridisegno della scena.
Il controller gestisce anche l'input utente. I metodi relativi al movimento e allo sparo (moveLeftPressed, shootPressed, ecc.) aggiornano direttamente i flag del giocatore, ottenuto

tramite model.getPlayer(). Il metodo onMousePressed(Point p) gestisce i click su pulsanti della GUI come "Play", "Retry", "Menu" o "Mute", verificando le coordinate rispetto alle aree interattive del GamePanel e aggiornando lo stato di gioco e l'audio. La gestione dell'inserimento del nome avviene con onKeyEvent(KeyEvent e), che raccoglie i caratteri digitati e, al termine, registra il punteggio tramite il ScoreManager accessibile dalla vista. Infine, il GameController implementa l'interfaccia GameSoundListener per reagire agli eventi del modello riproducendo i suoni appropriati tramite il SoundManager, come gli effetti di sparo, esplosioni o perdita di vita.
Attraverso questo insieme di metodi e interazioni, GameController mantiene sincronizzati logica, interfaccia grafica e audio, assicurando un'esperienza di gioco coerente e dinamica.

InputHandler

La classe InputHandler si occupa della gestione degli input da tastiera nel gioco *JHeliFire*. Essa estende KeyAdapter, una classe di utilità della libreria Java che semplifica la gestione degli eventi da tastiera, permettendo di sovrascrivere solo i metodi necessari. InputHandler funge da tramite tra gli eventi generati dall'utente e le azioni da eseguire sul gioco, delegando ogni comando al GameController.
Il costruttore della classe riceve come parametro un'istanza di GameController, che rappresenta il componente centrale responsabile della logica di gioco. Questo riferimento viene memorizzato in un campo privato e utilizzato nei metodi della classe per inoltrare gli input ricevuti. In questo modo, InputHandler resta indipendente dalla logica del gioco e si limita a trasmettere i comandi al controller.
La classe ridefinisce i metodi keyPressed(KeyEvent e) e keyReleased(KeyEvent e), che vengono chiamati automaticamente dal sistema quando l'utente preme o rilascia un tasto. All'interno di keyPressed, l'evento viene analizzato recuperando il codice del tasto premuto tramite getKeyCode(). Un'istruzione switch verifica quale tasto è stato premuto, e in base al valore del codice vengono richiamati i metodi corrispondenti del GameController. In particolare:

- la pressione delle frecce direzionali (VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN) attiva i metodi moveLeftPressed, moveRightPressed, moveUpPressed, moveDownPressed, che aggiornano i flag di movimento del Player;
- la pressione della barra spaziatrice (VK_SPACE) attiva il metodo shootPressed, che avvia l'azione di sparo.

In modo analogo, keyReleased gestisce il rilascio dei tasti. Anche in questo caso viene verificato quale tasto è stato rilasciato, e viene richiamato il metodo appropriato del GameController, come moveLeftReleased o shootReleased. Questo permette di disattivare il movimento o l'azione quando il tasto corrispondente non è più premuto.
Attraverso questa struttura, InputHandler assicura che ogni comando dell'utente venga trasmesso in tempo reale al GameController, che a sua volta agisce sul modello e aggiorna lo stato del gioco. La separazione dei compiti tra la classe di input e il controller contribuisce a mantenere il codice organizzato e facilmente modificabile.

Di seguito viene riportato il diagramma UML del package controller. Si osservi che vengono riportati solo gli attributi e i metodi principali di ciascuna classe.

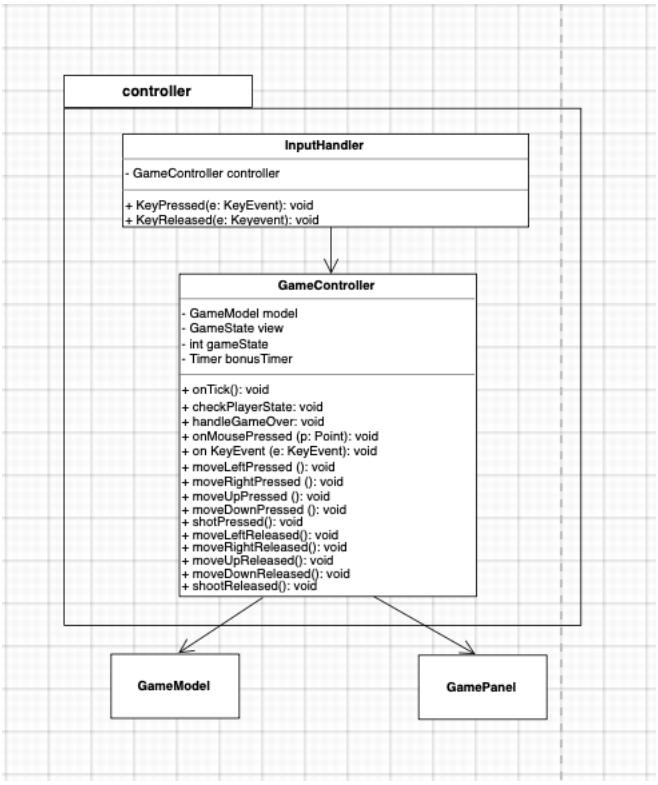


Fig. 4 Diagramma UML pacchetto delle classi del package controller

3.2.3 model

Il pacchetto model costituisce il nucleo logico dell'applicazione *JHeliFire* e raccoglie tutte le classi che rappresentano lo stato del gioco e le entità che lo popolano. In questo pacchetto sono presenti le definizioni delle entità dinamiche come il giocatore, i diversi tipi di nemici, i proiettili, le esplosioni e gli oggetti bonus, oltre alle componenti dedicate alla gestione degli eventi e del ciclo di gioco. Le classi contenute si occupano di modellare il comportamento e le proprietà degli oggetti presenti nella scena, aggiornandone lo stato a ogni ciclo e gestendone le interazioni, come i movimenti e le collisioni.
Il pacchetto include numerose classi, tra cui GameModel, che rappresenta il cuore della logica del gioco e coordina lo stato delle entità, e GameModelListener, che gestisce la notifica degli eventi sonori. Inoltre, sono presenti classi che modellano i nemici e i loro comportamenti: in particolare la superclasse Enemy, da cui derivano i diversi tipi di elicotteri (BlueHeli, GreenHeli, YellowHeli) e le navi nemiche. Altre classi presenti nel pacchetto sono dedicate alla gestione dei proiettili, delle esplosioni e delle scene bonus.
Di seguito verranno descritte in modo più approfondito le classi principali del pacchetto, evidenziando il ruolo e le interazioni di ciascuna all'interno del sistema.

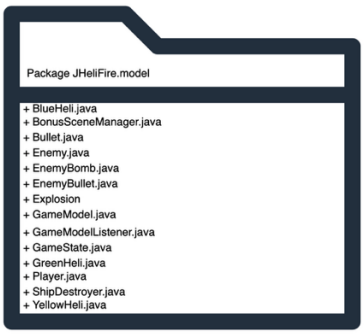


Fig. 5 Classi del package model

GameModel

La classe GameModel rappresenta il nucleo della logica del gioco in *JHeliFire* e gestisce lo stato complessivo della partita, coordinando il funzionamento del giocatore, dei nemici, dei proiettili, delle esplosioni e delle ondate. Essa mantiene al proprio interno i riferimenti a tutte le entità dinamiche del gioco: un oggetto Player, che rappresenta il giocatore e le sue proprietà; liste di oggetti come Enemy, Bullet, EnemyBullet, EnemyBomb ed Explosion, che rappresentano rispettivamente i nemici attivi, i proiettili del giocatore, i proiettili nemici, le bombe sganciate dagli elicotteri e le esplosioni. La classe tiene inoltre traccia del punteggio corrente, del record (high score), del livello, dello stato delle ondate, della disponibilità di un bonus e della condizione di vittoria.

L'istanza di GameModel viene inizializzata tramite il costruttore, che riceve la larghezza e l'altezza della finestra di gioco e richiama il metodo init(). Quest'ultimo si occupa di creare il giocatore, svuotare e reimpostare le liste delle entità dinamiche, azzerare il punteggio e preparare la prima ondata di nemici. Il metodo spawnWave() avvia una nuova ondata di nemici delegando la generazione a uno dei metodi specifici (spawnGreenWave, spawnBlueWave, spawnYellowWave) in base al tipo previsto. All'interno di questi metodi avviene la creazione e il posizionamento dei nemici, con coordinate casuali e parametri legati al livello.

Il metodo principale della classe è update(), che viene invocato a ogni ciclo dal GameController. All'interno di questo metodo si aggiornano il giocatore, i proiettili del giocatore e dei nemici, le bombe nemiche, le esplosioni e i nemici stessi. La gestione delle collisioni viene realizzata tramite i metodi handlePlayerBulletEnemyCollisions(), handleEnemyBulletPlayerCollisions(), handleShipDestroyerPlayerCollision() e handleEnemyBombPlayerCollisions(), tutti richiamati da update() per verificare e risolvere le interazioni tra le entità durante il gioco. Questi metodi controllano le intersezioni tra le hitbox degli oggetti e si occupano delle conseguenze delle collisioni, come la creazione delle esplosioni, l'aggiornamento del punteggio, la rimozione delle entità coinvolte e la notifica al listener sonoro per la riproduzione degli effetti audio. Le entità non più attive o visibili vengono rimosse dalle liste per liberare memoria e risorse. Il metodo update() gestisce inoltre l'avanzamento delle ondate e dei livelli tramite handleWaveAndLevelProgression(), che verifica quando tutte le ondate sono terminate e, se opportuno, avvia il livello successivo o imposta lo stato di vittoria. Quando è il momento, il metodo handleShipDestroyerSpawning() controlla lo spawn delle navi nemiche marine, rispettando i limiti stabiliti per livello.

Il GameModel offre diversi metodi di servizio e accesso: getScore(), getHighScore(), getLevel() e getPlayer() permettono al controller e al pannello grafico di ottenere informazioni sullo stato corrente della partita. I metodi come addScore(int points) aggiornano il punteggio e notificano i listener, mentre updateHighScore() verifica e aggiorna il record se il punteggio attuale lo

supera. Le liste delle entità (nemici, proiettili, bombe, esplosioni) possono essere recuperate tramite metodi come getEnemies() o getBullets(), così che il GamePanel possa disegnarle. Il metodo shoot() si occupa di creare un nuovo proiettile per il giocatore e notifica l'evento sonoro corrispondente. Per quanto riguarda il ciclo delle ondate, il modello gestisce direttamente l'avvio delle nuove ondate e l'aumento progressivo della difficoltà attraverso l'incremento del numero di nemici per tipo.
GameModel interagisce costantemente con numerose classi del pacchetto model, come Player, Enemy e le sue sottoclassi (GreenHeli, BlueHeli, YellowHeli, ShipDestroyer), Bullet, EnemyBullet, EnemyBomb ed Explosion. Inoltre, comunica con il GameController, che ne richiama i metodi per aggiornare lo stato e avviare o resettare la partita. Infine, attraverso l'interfaccia GameSoundListener, GameModel notifica al controller eventi come la distruzione di un nemico, la perdita di una vita o lo sparo del giocatore, così che vengano riprodotti i suoni appropriati tramite il SoundManager.
Grazie alla sua struttura e ai metodi offerti, GameModel si configura come la componente centrale che governa la logica e la progressione del gioco, assicurando un flusso ordinato delle entità e delle dinamiche che caratterizzano la partita.

Enemy

La classe Enemy rappresenta la superclasse astratta che definisce le proprietà e i comportamenti comuni a tutti i nemici presenti nel gioco JHeliFire. Essa costituisce il modello base da cui derivano le diverse tipologie di nemici, come GreenHeli, BlueHeli, YellowHeli e ShipDestroyer, ciascuna delle quali implementa il movimento e le caratteristiche specifiche. La realizzazione di Enemy come classe astratta consente di centralizzare le funzionalità comuni, garantendo un'architettura modulare e facilitando la manutenzione e l'estensione del codice.
La classe mantiene le informazioni fondamentali per la gestione del nemico, tra cui la posizione (x, y), le dimensioni (width, height), la velocità (speed) e il livello di gioco (level). Contiene inoltre un riferimento al GameModel, che le permette di interagire con il modello per aggiungere proiettili o bombe durante la partita. Il sistema di sparo del nemico è gestito tramite attributi come shootProbability, shootCooldown e maxShootCooldown, che regolano la probabilità e la frequenza con cui il nemico può sparare o sganciare una bomba.
Il metodo principale della classe è update(), marcato come final per impedire la sua ridefinizione da parte delle sottoclassi. Questo metodo rappresenta il ciclo di aggiornamento del nemico e coordina due operazioni: richiama il metodo astratto movement(), che deve essere implementato dalle sottoclassi per definire il movimento specifico del nemico, e successivamente invoca handleShooting(), che gestisce il sistema di sparo. Quest'ultimo metodo decrementa il cooldown dello sparo e, quando il cooldown è terminato, utilizza un controllo probabilistico per determinare se il nemico debba sparare un proiettile o sganciare una bomba. Se decide di sparare, invoca il metodo astratto shoot(), che viene implementato dalle sottoclassi con lo schema di sparo personalizzato. In alternativa, può richiamare il metodo shootBomb(), che crea una bomba nella posizione corretta e la aggiunge al modello.
La classe offre inoltre metodi di supporto per il sistema di sparo, come shootBasic(), che implementa un pattern base di proiettile verticale verso il basso, e shootAdvanced(), che può essere ridefinito dalle sottoclassi per realizzare schemi più complessi nei livelli avanzati. Il metodo shootBomb() è fornito direttamente dalla superclasse e standardizza il lancio delle bombe.

Per quanto riguarda la gestione delle collisioni, Enemy fornisce i metodi getBounds() e getHitBox(). Il primo restituisce un rettangolo che rappresenta l'ingombro complessivo del nemico, mentre il secondo fornisce una hitbox più piccola e centrata, utilizzata per rendere le collisioni più precise durante il gioco. Infine, il metodo astratto getScoreValue() obbliga le sottoclassi a definire il valore in punti assegnato alla distruzione del nemico.
La classe Enemy interagisce strettamente con il GameModel, che la aggiorna e ne gestisce la vita all'interno del ciclo di gioco. Le sottoclassi implementano il movimento e il comportamento specifico, sfruttando la struttura comune fornita dalla superclasse per integrare il proprio funzionamento nella logica complessiva del sistema.

YellowHeli

La classe YellowHeli rappresenta l'elicottero nemico più sofisticato dell'intero gioco e si distingue dagli altri per la complessità dei movimenti e per la gestione del fuoco, che evolve con il progredire della difficoltà. Fin dalla costruzione, tramite il costruttore YellowHeli(...), vengono inizializzate la posizione, le dimensioni, la velocità e i parametri di sparo, che vengono scalati in base al livello per rendere questo avversario sempre più impegnativo man mano che l'esperienza del giocatore avanza.
Il comportamento in partita è governato principalmente dal metodo movement(), che ha il compito di aggiornare sia lo spostamento sia l'animazione ad ogni ciclo di gioco. Al suo interno, la logica è delegata a handleMovement(), il quale, in base allo stato corrente, richiama metodi specifici per i diversi tipi di traiettoria. Se l'elicottero si muove orizzontalmente vengono utilizzati handleLeftMovement() e handleRightMovement(), che regolano lo spostamento rispettivamente verso sinistra e verso destra, controllando i limiti dello schermo e decidendo in certi casi di invertire direzione oppure di passare ad un movimento diagonale. In modo analogo, handleUpMovement() e handleDownMovement() gestiscono il movimento verticale entro i margini prefissati, con possibilità di cambi di direzione o di transizione verso una traiettoria obliqua. Quando l'elicottero deve spostarsi in diagonale interviene handleDiagonalMovement(), che permette al nemico di raggiungere un bersaglio interno allo schermo seguendo un percorso inclinato. La preparazione di queste manovre è affidata al metodo prepareDiagonalMovement(), che definisce il punto da raggiungere e lo stato successivo da assumere una volta completato il tragitto. Tutto ciò rende il movimento del YellowHeli molto meno prevedibile rispetto agli altri nemici, aumentando la difficoltà di intercettarlo. Parallelamente, updateAnimation() cura la parte grafica dell'animazione, gestendo il passaggio ciclico tra tre fotogrammi con un ritardo programmato che conferisce fluidità e realismo al movimento delle pale dell'elicottero.
Anche la gestione del fuoco è più complessa e varia a seconda del livello. Il metodo handleShooting() stabilisce il comportamento offensivo: fino al terzo livello viene mantenuta la logica classica, dove lo sparo è affidato al metodo shoot(), con tempi di ricarica e probabilità calcolati in base alla difficoltà. A partire dal quarto livello, però, l'elicottero cambia radicalmente approccio, introducendo il fuoco a raffica grazie a shootAdvanced(). In questa modalità non vengono più generati colpi singoli, ma sequenze di proiettili verticali sparati in rapida successione. L'implementazione prevede contatori interni che scandiscono il numero di proiettili restanti in una raffica e il tempo di attesa tra una raffica e l'altra: in questo modo il nemico alterna momenti di fuoco intenso a brevi pause, producendo un ritmo di attacco che risulta particolarmente pressante per il giocatore.

Oltre al movimento e al fuoco, la classe mette a disposizione alcuni metodi di supporto che ne completano il funzionamento. getScoreValue() assegna quaranta punti al giocatore che riesce ad abbattere l'elicottero, un valore superiore rispetto a quello degli altri nemici, a conferma della sua maggiore pericolosità. getAnimIndex() e getAnimFrames() servono invece a restituire le informazioni necessarie al ciclo di animazione, mentre isFacingLeft() indica l'orientamento orizzontale corrente, utile per determinare correttamente la resa grafica dell'elicottero sullo schermo.

Player

La classe Player rappresenta il giocatore controllato dall'utente all'interno del gioco JHeliFire. Essa modella le caratteristiche fondamentali del sottomarino, gestendo la posizione sullo schermo, i movimenti, le vite disponibili e lo stato di invulnerabilità temporanea dopo un colpo subito. Player offre inoltre i metodi necessari per aggiornare il proprio stato in risposta ai comandi impartiti dall'utente tramite la tastiera.
La classe mantiene come attributi principali le coordinate (x, y) che indicano la posizione corrente del giocatore, le dimensioni (width, height) utilizzate per il calcolo delle collisioni, e le velocità orizzontale e verticale (dx, dy). Sono presenti flag booleani (leftPressed, rightPressed, upPressed, downPressed) che rappresentano i comandi di movimento attivi. Player gestisce anche il numero di vite disponibili (lives), inizializzato a 3, un flag che indica se il giocatore è ancora vivo (alive) e un timer di invulnerabilità (invulnerabilityTimer), che protegge il giocatore per un certo numero di cicli dopo aver subito un colpo.
Il metodo principale della classe è update(), invocato dal GameModel a ogni ciclo di gioco. Questo metodo coordina l'aggiornamento del giocatore richiamando tre metodi interni: updateMovement(), che verifica i flag dei comandi e imposta le velocità orizzontale e verticale di conseguenza; updatePosition(), che aggiorna le coordinate del giocatore e le mantiene all'interno dei limiti dell'area di gioco (seaLeft, seaRight, seaTop, seaBottom); e updateInvulnerability(), che decrementa il timer di invulnerabilità quando attivo, rendendo il giocatore vulnerabile solo dopo che il tempo è scaduto.
La classe offre metodi specifici per la gestione delle vite: loseLife() decrementa il numero di vite se il giocatore non è invulnerabile, attiva il timer di invulnerabilità se ci sono ancora vite residue, oppure segna il giocatore come non più vivo se le vite si esauriscono. Il metodo addLife() permette di incrementare il numero di vite, fino a un massimo di 3. Per il rilevamento delle collisioni, Player fornisce il metodo getHitBox(), che restituisce un rettangolo più piccolo e centrato rispetto all'immagine del giocatore, utile per un rilevamento preciso durante il confronto con proiettili o nemici.
Player interagisce direttamente con il GameModel, che ne richiama i metodi per aggiornare lo stato e verificare le collisioni, e con il GameController, che modifica i flag di movimento in risposta agli input dell'utente. Il GamePanel, durante il disegno della scena, utilizza le informazioni di posizione e dimensione fornite dal Player per rappresentare graficamente il veicolo controllato dal giocatore. Grazie alla sua struttura, la classe Player incarna il modello del giocatore all'interno della logica di gioco e garantisce una gestione precisa e ordinata delle azioni e delle interazioni durante la partita.

GameModelListener

L'interfaccia GameModelListener rappresenta un meccanismo semplice ed efficace per la gestione degli eventi nel gioco JHeliFire. Essa definisce un meccanismo che consente alle classi interessate di essere avvisate ogni volta che lo stato del GameModel viene modificato in modo significativo. L'obiettivo di questa interfaccia è favorire un disaccoppiamento tra la logica del gioco e le componenti che devono reagire agli aggiornamenti, come ad esempio la vista o altre parti del sistema.
L'interfaccia definisce un unico metodo:

void onModelChanged();

che viene implementato dalle classi che si collegano al GameModel come listener per ricevere notifiche sui cambiamenti dello stato di gioco. Questo metodo viene chiamato dal GameModel tramite il metodo interno notifyModelChanged() ogni volta che si verifica un cambiamento nello stato del gioco che richiede un aggiornamento della vista o di altre componenti. Ad esempio, dopo l'avanzamento delle entità, la gestione delle collisioni o il passaggio a un nuovo livello, il GameModel invoca onModelChanged() sui listener registrati, permettendo alla vista di ridisegnare la scena o aggiornare eventuali interfacce utente.
L'implementazione di GameModelListener avviene tipicamente all'interno del GamePanel o di altre classi della vista, che hanno il compito di rappresentare graficamente lo stato aggiornato del gioco. In questo modo, l'interfaccia consente di mantenere una separazione chiara tra la logica e la visualizzazione, migliorando l'organizzazione e la manutenibilità del codice.

BonusSceneManager

La classe BonusSceneManager si occupa della gestione della scena bonus che viene attivata durante il gioco JHeliFire in seguito al completamento di determinati livelli o al raggiungimento di specifici obiettivi.
Durante questa sequenza speciale il gioco si interrompe e la normale azione di gioco viene sospesa: il sottomarino controllato dal giocatore inizialmente scompare dalla parte inferiore dello schermo, mentre sul lato dello schermo fa la sua comparsa un'isola che si muove lentamente verso il centro della scena. Sull'isola è visibile un personaggio femminile che danza in attesa del sottomarino. Una volta che l'isola raggiunge la posizione prevista, il sottomarino emerge dal basso e si dirige verso l'isola, avvicinandosi progressivamente alla figura della donna. Dopo un breve momento di attesa, il sottomarino raccoglie il bonus: il personaggio femminile scompare e il sottomarino inizia la manovra di ritorno verso la sua posizione di partenza.
Conclusa questa fase, viene mostrato il tipo di premio ottenuto (punti aggiuntivi o una vita extra), prima che la scena venga segnata come completa e il gioco riprenda normalmente.
La classe modella il progresso della scena bonus attraverso una macchina a stati, basata sull'enumerazione Phase, che definisce le diverse fasi della sequenza: ingresso dell'isola (ISLAND_ENTRANCE), salita del sottomarino (SUB_UP), avvicinamento all'isola (SUB_TO_ISLAND), raccolta del bonus (PICKUP), ritorno del sottomarino (RETURN), visualizzazione del premio (SHOW_POINTS) e completamento della scena (COMPLETE).

Gli attributi principali della classe includono le coordinate dell'isola (islandX, islandY) e del sottomarino (subX, subY), le velocità di movimento (islandSpeed, subSpeed) e variabili di supporto come timer, frameCounter e womanVisible, che indica se il personaggio del bonus è ancora visibile. Il campo isPointBonus stabilisce la natura del premio: se true, il bonus assegnerà punti al giocatore; se false, fornirà una vita aggiuntiva.
Il metodo principale della classe è update(), richiamato a ogni ciclo del gioco dal GameController quando lo stato del gioco è impostato sulla scena bonus. Questo metodo incrementa il contatore dei frame e gestisce la progressione della scena in base alla fase corrente. Ogni fase modifica progressivamente le coordinate del sottomarino e dell'isola o aggiorna lo stato degli elementi visivi. Ad esempio, durante la fase ISLAND_ENTRANCE, l'isola avanza verso lo schermo fino a raggiungere una determinata posizione, dopodiché il sottomarino inizia a salire (SUB_UP) e successivamente si sposta verso l'isola (SUB_TO_ISLAND). Nella fase PICKUP, il timer determina il momento in cui il personaggio del bonus scompare, e il sottomarino comincia la fase di ritorno (RETURN). Al termine, viene mostrato il premio nella fase SHOW_POINTS prima di segnare la scena come completa (COMPLETE).
La classe offre diversi metodi di accesso per ottenere lo stato corrente della scena e degli elementi coinvolti, come getIslandX(), getSubY(), isWomanVisible(), getPhase() e isComplete(). Il metodo setPointBonus(boolean isPointBonus) permette al controller di definire la tipologia di premio associata alla scena bonus. Il metodo reset() riporta la scena allo stato iniziale, reimpostando le coordinate e i parametri, così da renderla pronta per una nuova esecuzione.
BonusSceneManager interagisce con il GameController, che ne richiama i metodi per aggiornare la scena e verificare il completamento del bonus durante lo svolgimento del gioco. La classe fornisce al GamePanel le informazioni necessarie per il disegno degli elementi grafici della scena bonus, permettendo una rappresentazione visiva coerente con lo stato logico del gioco.

Di seguito viene riportato il diagramma UML del package model. Si osservi che vengono riportati solo gli attributi e i metodi principali di ciascuna classe.

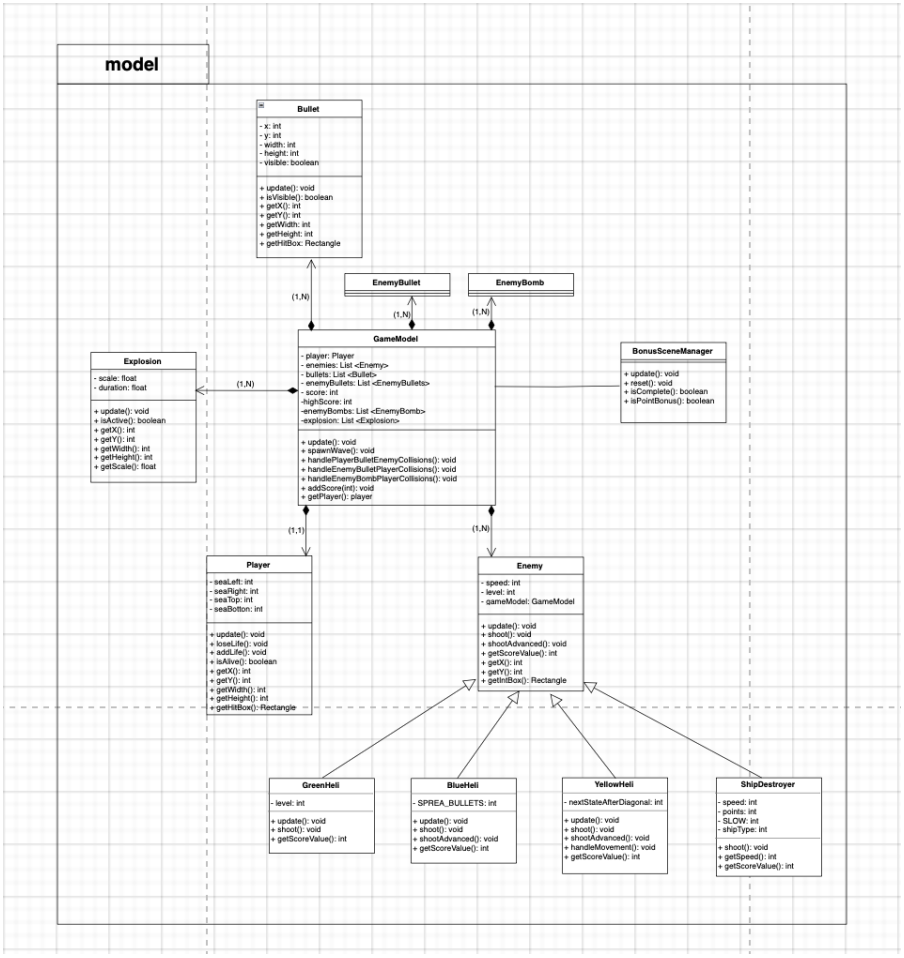


Fig. 6 Diagramma UML pacchetto delle classi del package model

3.2.4 utility

Il pacchetto utility raccoglie le classi di supporto che forniscono funzionalità complementari al gioco *JHeliFire*, fondamentali per arricchire l'esperienza complessiva. In particolare, contiene la classe SoundManager, responsabile della gestione degli effetti sonori e delle musiche di sottofondo, e la classe ScoreManager, che si occupa della memorizzazione e del recupero dei migliori punteggi ottenuti dai giocatori durante le partite.

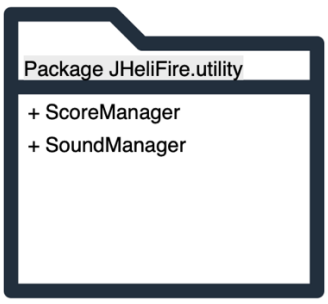


Fig. 7 Classi del package utility

Di seguito procediamo con la descrizione delle due classi del pacchetto.

ScoreManager

La classe ScoreManager si occupa della gestione della classifica dei migliori punteggi ottenuti dai giocatori durante le partite di *JHeliFire*. La classe conserva i punteggi in memoria e li memorizza su un file di testo esterno, denominato highscores.txt, per garantirne la persistenza tra una sessione e l'altra. Il suo scopo è fornire un sistema semplice ed efficace per il salvataggio, il caricamento e l'aggiornamento della top 3 dei migliori punteggi.
Tra gli attributi principali, ScoreManager mantiene una lista topScores che contiene oggetti ScoreEntry, una classe interna statica che rappresenta un singolo record della classifica, costituito da un nome e un punteggio. La lista viene inizializzata nel costruttore attraverso il metodo loadScores(), che legge il contenuto del file highscores.txt, carica i dati nella lista e li ordina in ordine decrescente di punteggio. Nel caso in cui il file non esista ancora (ad esempio alla prima esecuzione del gioco), il metodo cattura l'eccezione IOException e si limita a iniziare con una classifica vuota, che verrà popolata al primo salvataggio.
Il metodo addScore(String name, int score) consente di aggiungere un nuovo punteggio alla classifica. Dopo aver inserito il nuovo record, la lista viene ordinata nuovamente in ordine decrescente e ridotta alla top 3 eliminando eventuali punteggi in eccesso. Viene quindi richiamato il metodo saveScores(), che si occupa di scrivere la classifica aggiornata nel file di salvataggio. La scrittura avviene riga per riga, sfruttando il metodo toString() della classe ScoreEntry, che restituisce una rappresentazione del punteggio nel formato nome,score.
Il metodo isTop3(int score) verifica se un nuovo punteggio è sufficientemente alto da rientrare nella top 3. La verifica avviene confrontando il nuovo punteggio con l'ultimo elemento della lista ordinata, oppure restituendo true se la classifica attuale contiene meno di tre punteggi. Il metodo getTopScores() restituisce una copia della lista dei migliori punteggi per permettere ad altre classi, come la vista, di visualizzare la classifica senza modificare l'originale. Infine, getHighScore() fornisce il miglior punteggio attuale, restituendo un record fittizio con nome "---" e punteggio 0 se la classifica è ancora vuota.
ScoreManager interagisce con il GameController, che ne richiama i metodi per verificare se un nuovo punteggio rientra nella top 3, per aggiungere un nuovo record o per ottenere i dati da visualizzare a fine partita. In questo modo, la classe contribuisce alla gestione della classifica restando separata dalla logica principale di gioco, garantendo la persistenza dei dati e la semplicità di accesso alle informazioni sui punteggi.

SoundManager

La classe SoundManager si occupa della gestione dell'audio all'interno del gioco JHeliFire. Essa fornisce un insieme di metodi statici che permettono di riprodurre effetti sonori e musiche di sottofondo, centralizzando la gestione dei suoni e rendendola facilmente accessibile da qualsiasi parte del programma senza la necessità di creare istanze. Grazie alla sua struttura, SoundManager contribuisce a separare la logica audio dalla logica di gioco principale, migliorando l'organizzazione e la manutenibilità del codice.
La classe gestisce un attributo statico muted, che rappresenta lo stato globale dell'audio: quando questo flag è impostato a true, la riproduzione di qualsiasi suono viene disattivata. È presente anche un riferimento statico backgroundClip, che rappresenta il clip audio attualmente in esecuzione in loop, tipicamente usato per la musica di sottofondo. Per gli effetti sonori ripetuti più frequentemente, come lo sparo del giocatore o le esplosioni,

SoundManager utilizza due Clip precaricati (shootClip e explosionClip), riducendo così il ritardo nella riproduzione di questi suoni durante il gioco.
Il metodo preloadSounds() carica in anticipo questi clip audio più usati, tramite il metodo privato loadClip(String soundPath), che si occupa di ottenere il file audio dalla risorsa specificata, aprirlo e prepararlo per la riproduzione. Questo approccio migliora le prestazioni, poiché i suoni sono già pronti all'uso quando richiesti.
Per la riproduzione degli effetti sonori, la classe offre metodi come playShoot() e playExplosion(), che riproducono rispettivamente il suono dello sparo e dell'esplosione utilizzando i clip precaricati. Questi metodi si occupano di fermare eventuali riproduzioni precedenti dello stesso suono e di riavviare il clip dall'inizio, così da garantire che l'effetto audio sia sempre udibile al momento giusto. Il metodo playSound(String soundPath) consente di riprodurre un effetto sonoro singolo caricandolo al momento, utilizzato per suoni meno frequenti o specifici. Questo metodo aggiunge un LineListener al clip per chiudere automaticamente la risorsa al termine della riproduzione, evitando sprechi di memoria.
Per la musica di sottofondo o altri suoni da riprodurre in loop, SoundManager mette a disposizione il metodo playLoop(String soundPath). Questo metodo ferma e chiude eventuali loop già attivi, carica e avvia il nuovo suono in ripetizione continua e aggiorna il riferimento a backgroundClip. Per interrompere la musica di sottofondo, il metodo stopBackgroundLoop() verifica se un loop è attivo e, in caso positivo, lo ferma e libera la risorsa.
Infine, SoundManager offre i metodi setMuted(boolean m) e isMuted() per modificare o verificare lo stato dell'audio globale. Quando l'audio viene disattivato tramite setMuted(true), il metodo provvede anche a fermare la musica di sottofondo in esecuzione. La classe interagisce strettamente con il GameController, che richiama i metodi di SoundManager per riprodurre suoni in risposta agli eventi principali del gioco, come la vittoria, la sconfitta, gli spari o le esplosioni. Anche il GameModel, attraverso il GameSoundListener, si appoggia a SoundManager per notificare gli eventi sonori durante le collisioni o altre azioni di gioco.

Di seguito viene riportato il diagramma UML del package utility. Si osservi che vengono riportati solo gli attributi e i metodi principali di ciascuna classe.

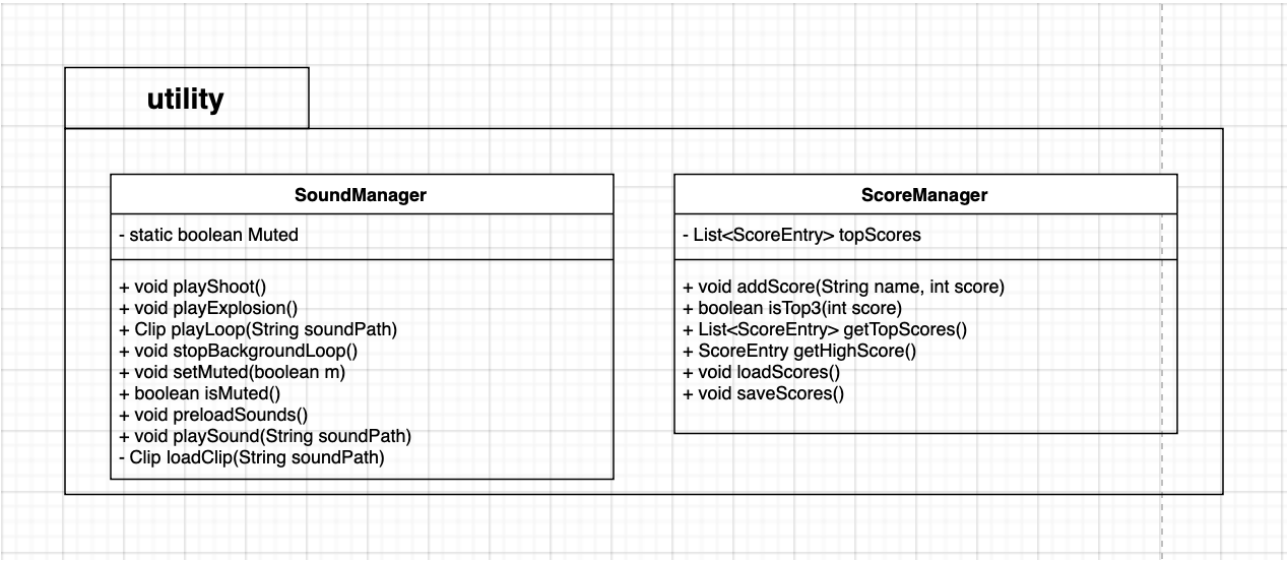


Fig. 8 Diagramma UML pacchetto delle classi del package utility

3.2.5 view

Il pacchetto view contiene le classi responsabili della gestione grafica del gioco *JHeliFire*. In particolare, include GameFrame, che rappresenta la finestra principale dell'applicazione, e GamePanel, che si occupa del disegno della scena di gioco, delle schermate e degli elementi dell'interfaccia. Queste classi interagiscono con le classi delle cartelle model e controller per visualizzare lo stato aggiornato della partita e raccogliere gli input dell'utente.

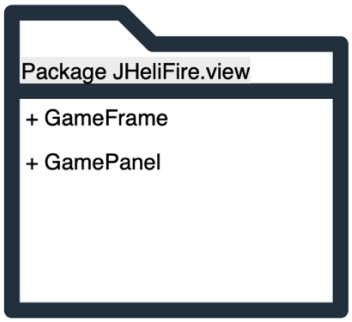


Fig. 9 Classi del package view

Di seguito procediamo con la descrizione delle due classi del pacchetto.

GameFrame

La classe GameFrame rappresenta la finestra principale del gioco *JHeliFire* e si estende da JFrame, una classe della libreria Swing di Java utilizzata per creare finestre con interfaccia grafica nelle applicazioni desktop. JFrame fornisce tutte le funzionalità di base necessarie per costruire e gestire una finestra, come il titolo, la gestione della chiusura, il contenuto visibile e le dimensioni.
Il costruttore di GameFrame riceve come parametro un oggetto GamePanel, che rappresenta il pannello su cui viene disegnata la scena di gioco. All'interno del costruttore, la finestra viene configurata impostando il titolo a "HeliFire", definendo che l'applicazione deve terminare quando la finestra viene chiusa (EXIT_ON_CLOSE) e disabilitando la possibilità di ridimensionarla (setResizable(false)) per mantenere la coerenza della grafica del gioco. Il GamePanel viene aggiunto alla finestra con il metodo add(panel), e pack() regola automaticamente le dimensioni della finestra per adattarle al contenuto. Infine, la finestra viene centrata sullo schermo con setLocationRelativeTo(null) e resa visibile con setVisible(true)
GameFrame interagisce direttamente con il GamePanel, fornendo il contenitore in cui il pannello viene visualizzato e aggiornato durante il gioco. La classe si limita alla configurazione della finestra e delega al pannello la gestione del disegno e dell'interazione con l'utente.

GamePanel

La classe GamePanel rappresenta il pannello grafico principale del gioco JHeliFire, responsabile del disegno della scena, della gestione dell'interfaccia utente e della raccolta degli input da tastiera e mouse. Estende JPanel della libreria Swing e implementa ActionListener, MouseListener e GameModelListener per integrare al meglio la logica di aggiornamento e l'interazione con il giocatore.
Il pannello definisce una dimensione fissa di 800x600 pixel e mantiene riferimenti ai componenti principali: un oggetto GameModel che rappresenta lo stato del gioco, un GameController che gestisce la logica, e un ScoreManager per la classifica dei punteggi. Include inoltre elementi grafici come l'immagine di sfondo, le icone per il mute/unmute, e le immagini per disegnare le entità di gioco. Il pannello gestisce lo scorrimento dello sfondo tramite la variabile backgroundX e il timer Swing per aggiornare e ridisegnare la scena a intervalli regolari (circa 60 frame al secondo).

Il costruttore inizializza il pannello configurandone le proprietà (dimensione, sfondo, layout, doppio buffering) e carica le risorse grafiche e audio necessarie. Viene avviata la musica di sottofondo e precaricati i suoni più usati grazie a SoundManager.preloadSounds(). Il GamePanel registra sé stesso come listener del modello per ricevere notifiche sui cambiamenti dello stato di gioco, avvia il timer e prepara un pulsante grafico per l'accesso alle opzioni.
Il metodo paintComponent(Graphics g) gestisce il disegno di tutti gli elementi grafici a seconda dello stato del gioco (menu iniziale, partita attiva, opzioni, game over, vittoria, inserimento nome, scena bonus). Invoca metodi specifici come drawEntities() per disegnare il giocatore, i nemici, i proiettili e le esplosioni, oppure drawHUD() per il punteggio, le vite e le informazioni di gioco. Metodi come drawMenu(), drawGameOver(), drawVictory() e drawOptions() si occupano della visualizzazione delle varie schermate del gioco e dell'interfaccia utente, inclusi i pulsanti e le istruzioni.
Il pannello raccoglie gli input del giocatore tramite mousePressed() e processKeyEvent(), inoltrandoli al GameController per la gestione della logica. Integra un pulsante grafico per l'accesso rapido alle opzioni e un sistema per attivare o disattivare l'audio con un clic sull'icona mute/unmute.
GamePanel collabora strettamente con il GameModel e il GameController per riflettere graficamente lo stato aggiornato del gioco e raccogliere gli input. Si interfaccia inoltre con il SoundManager per la gestione dell'audio e con il ScoreManager per mostrare la classifica dei migliori punteggi. Grazie alla sua struttura, la classe rappresenta il collegamento diretto tra la logica del gioco e la sua rappresentazione visiva, coordinando in modo integrato grafica, interazione e interfaccia.

Di seguito viene riportato il diagramma UML del package view. Si osservi che vengono riportati solo gli attributi e i metodi principali di ciascuna classe.

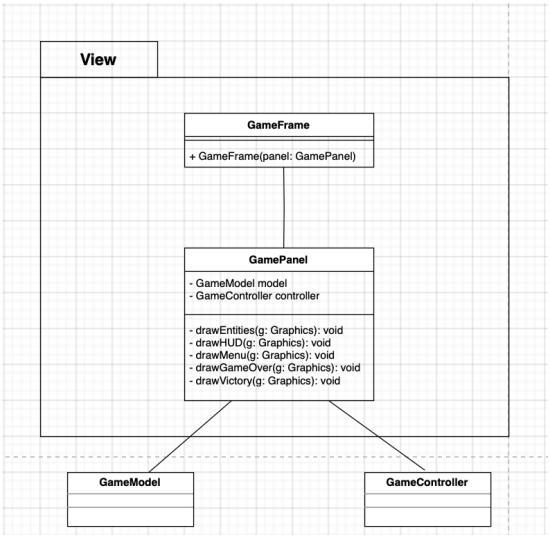


Fig. 10 Diagramma UML pacchetto delle classi del package view

3.3 Realizzazione delle risorse grafiche

Le immagini e gli sprite presenti nel videogioco sono stati disegnati direttamente da noi, utilizzando l'editor online *Pixilart*. Questa scelta ci ha permesso di replicare al meglio le immagini e lo stile grafico del gioco originale *HeliFire*, aggiungendo alcuni dettagli personalizzati per adattarli al nostro progetto e alle dimensioni dello schermo. In questo modo abbiamo potuto ottenere una resa visiva coerente con lo spirito retrò del gioco, pur introducendo elementi che riflettessero le nostre preferenze estetiche e le esigenze tecniche del prototipo.

Fig. 11 YellowHeli

Fig. 12 ShipDestroyer

Fig. 13 Player

Fig. 14

3.4 Problemi riscontrati

Durante lo sviluppo dell'applicazione *JHeliFire*, sono emerse alcune problematiche legate sia alla gestione di specifiche funzionalità di gioco, sia all'organizzazione del codice e alla sua strutturazione in pacchetti.

Inizialmente, il codice era sviluppato in maniera più monolitica, senza una chiara separazione delle responsabilità tra le varie componenti. Questo approccio ha presto mostrato dei limiti, soprattutto nel momento in cui è stato necessario ampliare le funzionalità e gestire elementi complessi come il sistema di punteggio, l'audio, la scena bonus e la progressione dei livelli. Per migliorare la manutenibilità del progetto, si è deciso di ristrutturare il codice suddividendolo in pacchetti distinti: *model*, dedicato alla logica del gioco e alla gestione delle entità dinamiche*; view,* per la visualizzazione grafica e l'interfaccia utente; *controller*,

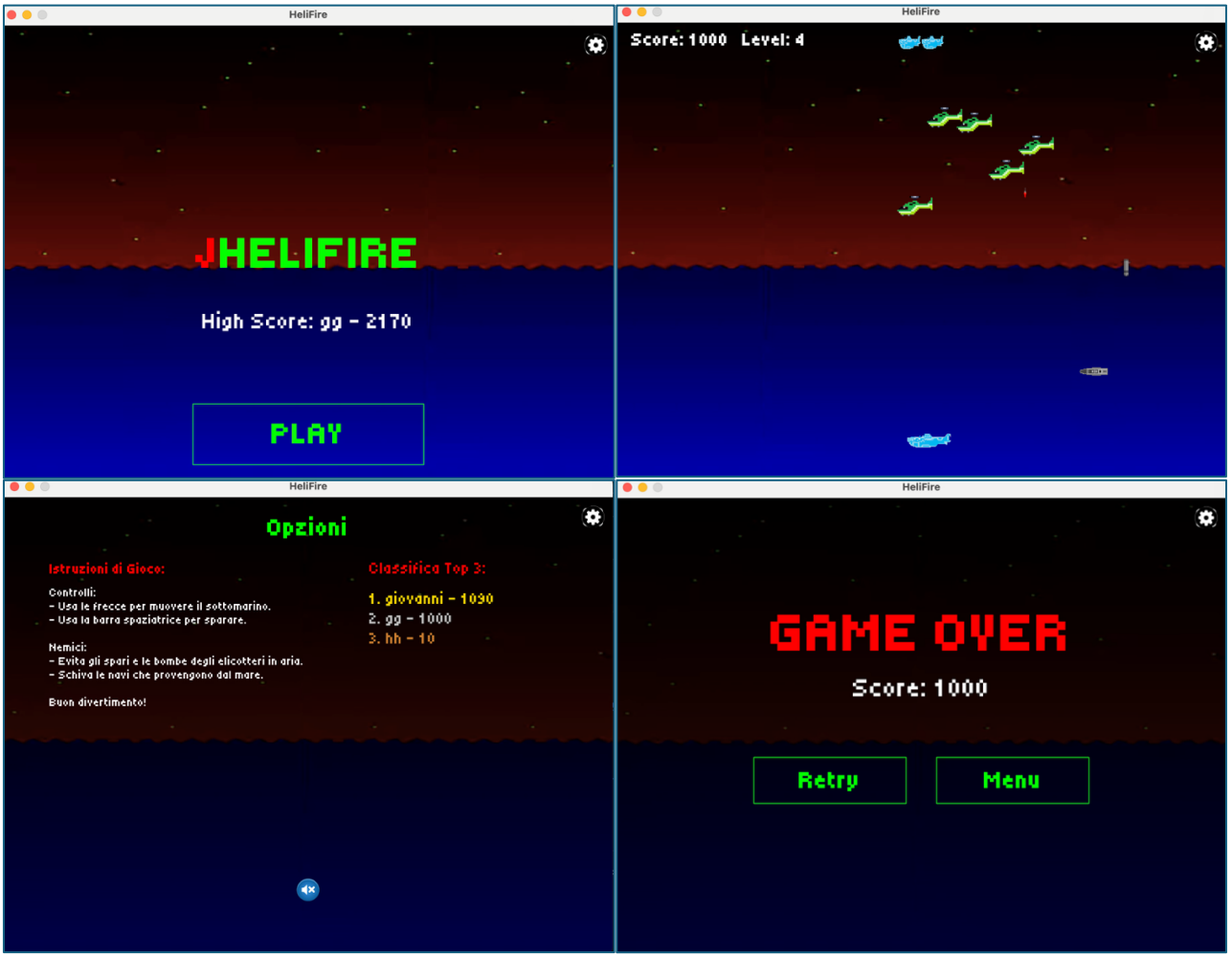
responsabile della logica applicativa e del coordinamento degli eventi; *utility*, per i servizi ausiliari come la gestione dell'audio e dei punteggi; e *main*, come punto di ingresso dell'applicazione.
Un'altra sfida significativa è stata la gestione corretta delle collisioni tra entità, in particolare quando erano presenti molte istanze contemporaneamente (proiettili, nemici, esplosioni, bombe). Si è resa necessaria un'attenta progettazione delle hitbox e una gestione sicura delle rimozioni, per evitare errori logici o rallentamenti nel ciclo di aggiornamento del gioco.
Anche l'integrazione dell'audio ha richiesto attenzione: la riproduzione di effetti sonori in tempo reale ha mostrato ritardi nelle prime versioni, risolti precaricando i suoni più frequenti (spari, esplosioni) attraverso la classe SoundManager.

4 Schermate di gioco

Nel progetto, il gioco è strutturato in diverse schermate, ognuna delle quali rappresenta uno stato specifico del gameplay. Gli stati sono gestiti centralmente attraverso la classe GameState, che permette di organizzare e disaccoppiare la logica del gioco. Le schermate principali includono:

- **Start Screen**: La schermata iniziale, dove il giocatore può avviare il gioco o accedere alle opzioni.
- **Game Play**: La schermata principale del gioco, dove si svolge l'azione.
- **Options**: Una schermata che permette di visualizzare la classifica, attivare/disattivare l'audio (mute/unmute) e consultare le istruzioni di gioco.
- **Game Over**: La schermata che appare quando il giocatore perde.
- **Victory**: La schermata di vittoria, mostrata al completamento del livello.
- **Enter Name Screen**: La schermata per inserire il nome del giocatore, utile per salvare i punteggi.
- **Bonus Cutscene**: Una scena speciale che premia il giocatore con punti bonus o vite extra.

Vengono illustrate sotto le schermate di gioco che rappresentano gli stati del gioco principali:




HeliFire
CLASSIFICA TOP 3!
Inserisci il tuo nome: _
(Premi INVIO per confermare)


HeliFire
YOU WIN!
Score: 1230
Replay
Menu

5 Conclusioni e sviluppi futuri

Lo sviluppo di *JHeliFire* ha rappresentato un'esperienza formativa significativa, permettendoci di affrontare concretamente tutte le fasi del ciclo di vita di un'applicazione interattiva: dall'analisi dei requisiti, alla progettazione strutturata, fino alla realizzazione di un sistema funzionante, completo di interfaccia grafica, audio e logiche di gioco avanzate.
Il progetto ci ha permesso di approfondire la programmazione ad oggetti in Java e l'utilizzo della libreria Swing per la creazione di applicazioni desktop, nonché di sperimentare tecniche di gestione degli eventi, rendering grafico e sincronizzazione audio. Particolare attenzione è stata dedicata all'organizzazione del codice in pacchetti funzionali (model, view, controller, utility, main): una scelta che si è rivelata efficace per garantire modularità, leggibilità e manutenibilità del progetto.

Sebbene l'applicazione soddisfi pienamente i requisiti iniziali, vi sono diverse possibilità per estenderne ulteriormente le funzionalità in futuro. Tra i principali sviluppi possibili si riportano:

- **Introduzione di nuovi nemici** con pattern di attacco e movimenti differenti;
- **Sistema di power-up** (es. velocità aumentata, sparo potenziato, scudi temporanei);
- **Modalità multiplayer locale**, con due giocatori sullo stesso schermo;
- **Porting su piattaforme mobili** (Android), con adattamento dell'interfaccia e dei controlli touch.

In conclusione, JHeliFire rappresenta una solida base su cui costruire ulteriori evoluzioni, e si configura come un progetto didattico completo che unisce aspetti teorici e pratici della programmazione, della progettazione del software e della realizzazione di videogiochi 2D.

6 Bibliografia

- Java Swing (JFC), https://docs.oracle.com/en/java/javase/24/
- Java Sound API, https://docs.oracle.com/javase/tutorial/sound/index.html
- Documentazione ufficiale Java™ Platform, Standard Edition 8 API Specification, https://docs.oracle.com/javase/8/docs/api/
- Pixilart, http://www.pixilart.com
- Heli Fire (Nintendo, 1980), https://en.wikipedia.org/wiki/Heli_Fire