PizzaTime Project Report

Paradigmi di Programmazione e Sviluppo

Lorenzo Chiana
Meshua Galassi
Giada Gibertoni
Giacomo Pasini

Anni accademici 2018–2019 e 2020–2021

Indice

Capitolo 1

Processo di sviluppo

Nel seguente capitolo si discute delle metodologie di sviluppo adottate dal team e delle scelte tecnologiche fatte per il compimento del progetto.

1.1 Metodologia di sviluppo

Come metodologia di sviluppo abbiamo scelto di applicare uno sviluppo agile con scrum. Abbiamo suddiviso il lavoro in sei sprint totali, di cui: cinque, relativi all'implementazione di nuovi componenti, della durata di una settimana ciascuna. Una di circa due settimane in cui ci siamo occupati della rifattorizzazione del codice e della risoluzione di bug. Quest'ultima non è riportata all'interno dello sprint planning in quanto non aggiungeva funzionalità.

Prima di partire con l'implementazione si è tenuta una riunione preliminare in cui è stata fatta una sessione di scoping e planning in modo tale da definire i requisiti principali, redigere il product backlog e definire le stime dei tempi di sviluppo.

Inoltre, si è svolta una piccola fase di launching in cui è stata concordata da tutti i membri, l'architettura di base dell'applicazione.

Product backlog

Priority	Item	Details	Initial Size Estimate
1	Utente naviga nel menù iniziale	Inizio partita, settings, credits, classifica	S
2	Utente vede arena	Creazione componenti arena (muri e pavimento)	M
3	Utente muove il player nell'arena	Creazione player e funzioni di movimento	L
4	Utente sceglie che difficoltà deve avere il gioco	Livelli di difficoltà	M
5	Utente vede ostacoli		S
6	Utente può raccogliere bonus partita		M
7	Utente vede i nemici nell'arena	Interazione con i nemici, movimento semplice casuale	L
8	Utente spara		M
9	Utente accumala score e perde vite		XS
10	Utente si muove tra i livelli		XL
11	Utente visualizza la classifica e ha la possibilità di salvare il suo score	Classifica generica con indicazione del livello di difficoltà	S
12	La partita è caratterizzata da suoni		M
13	Utente gioca una partita con comportamento dei nemici variabili, dipendente dalla difficoltà.	Prolog - movimento nella direzione del giocatore + vari tipi di movimento caratteristici per diversi tipi di nemici	XXL

Figura 1.1: Product Backlog

1.2 Strumenti utilizzati

1.2.1 Versioning

Il sistema di *versioning* ha ricorperto un ruolo centrale nella realizzazione del progetto e per tale scopo abbiamo scelto di utilizzare **Git**. Per quanto riguarda l'hosting del repository abbiamo optato per la piattaforma **GitHub**.

Il flusso di lavoro è stato definito secondo le linee guida del metodo di branching **git-flow**:

- Il branch *master* contiene il codice relativo a ciascuna release.
- Il branch *development* ospita il codice realizzato durante uno sprint. Tale codice è testato e stabile, ma non ancora giudicato completamente utilizzabile.
- I vari branch *task-** corrispondono a ciascun task individuato durante i vari sprint. Tale codice è in fase di implementazione e può contenere codice non completamente funzionante, in quanto ancora in sviluppo.

- è stato necessario creare un branch di *hotfix* per risolvere un bug presente nella release v1.0.

1.2.2 Dependency management & buildscript

Per la gestione delle dipendenze, quali ad esempio librerie o plugin, e la compilazione del codice, come strumento di automazione dello sviluppo si è utilizzato **sbt**. In particolare, sono stati scritti buildscript per automatizzare:

- la gestione delle *dipendenze* provenienti da repository Maven;
- il processo di *testing* tramite ScalaTest e controllo di *qualità del codice* tramite Scalastyle;
- generazione dei *Jar* eseguibili.

1.2.3 Continuous Integration

Per la verifica del codice prodotto e dei reativi test si è optato per *Travis CI*. Questo servizio offre la possibilità di registrare un web-hook al repository GitHub su cui è istanziato il progetto, così da provocare l'esecuzione di Travis CI ad ogni commit effettuato sul repository. La configurazione di Travis è stata definita utilizzando il formato YAML e in particolare sono stati specificati:

- il linguaggio (Scala);
- la versione di Scala;
- la JDK da utilizzare.

Capitolo 2

Analisi del dominio

2.1 Descrizione del gioco

Il progetto consiste nello sviluppo di un gioco di tipo Battle Arena con grafica 2D a visuale fissa. Il giocatore interpreta un personaggio che si muove all'interno di una mappa generata casualmente, sconfiggendo nemici e completando livelli.

Il giocatore si sposta da una stanza di dimensioni fisse ad un'altra, che rappresentano i diversi livelli del gioco. Una stanza può contenere un numero arbitrario di nemici, oggetti collezionabili e ostacoli; sconfiggere tutti i nemici garantisce l'accesso al livello successivo. Solo una stanza è visibile a schermo in un dato momento.

L'obiettivo del gioco potrebbe essere quello di raggiungere una stanza contenente un oggetto speciale che determina la fine della partita, o, in alternativa, il completamento di un certo numero di stanze. Nel primo caso, anche la probabilità di trovare l'oggetto speciale è casuale, ma comunque dipendente da alcune condizioni di gioco (come, ad esempio, il raggiungimento di una soglia di punteggio). Tuttavia, il gioco potrebbe essere giocato in una modalità che ha come scopo la sopravvivenza, solo per ottenere un punteggio personale più alto, che viene mantenuto salvato e mostrato nella sezione della classifica del menu. La natura casuale del gioco implica una curva di difficoltà facilmente variabile, facendo dipendere il gameplay anche dalla fortuna, che può renderlo divertente ma a volte anche impegnativo.

L'arma principale a disposizione del giocatore sono proiettili (pomodori) da lanciare ai nemici; gli oggetti collezionabili garantiscono un bonus di punteggio o di vita.

Il gioco consente di registrare diversi profili utente per salvare le statistiche di gioco e confrontarle in una classifica.

Capitolo 3

Requisiti

3.1 Requisiti di business

Sono stati individuati i seguenti requisiti necessari per la realizzazione del progetto:

- possibilità di avviare una partita tramite un menu di gioco
- possibilità di comandare un personaggio all'interno di una mappa di gioco
- possibilità di salvare i dati del giocatore

3.2 Requisiti utente

Sono stati poi individuati i requisiti che rappresentano ciò che l'utente deve poter fare con l'applicazione. Un utente potrà:

- scegliere la difficoltà di gioco;
- avviare una partita attraverso un menu, che dovrà contenere almeno le seguenti voci:
  - iniziare una nuova partita;
  - visualizzare i punteggi;
  - modificare le impostazioni.
- sconfiggere nemici, che devono essere:
  - dotati di un movimento indipendente;
  - capaci di danneggiare il giocatore;
  - dipendenti dalla difficoltà;
  - necessari per il completamento del livello.
- raccogliere oggetti ed evitare ostacoli;
- accumulare un punteggio quando progredisce nel gioco, che si ottiene in caso di:

  - uccisione di un nemico;
  - raccoglimento di un oggetto collezionabile.
- salvare automaticamente il proprio punteggio;
- visualizzare la classifica di tutti i profili registrati.

3.3 Requisiti funzionali

L'applicazione deve implementare le seguenti funzionalità:

- interfaccia di gioco 2D a visuale fissa;
- gestione di una mappa di gioco, in particolare:
  - definizione di un sistema di coordinate adeguato a gestire sia entità statiche che dinamiche in una mappa 2D;
  - operazioni di creazione e aggiornamento della mappa per poter gestire il popolamento dell'area di gioco con le varie entità.
- gestione di tutte le entità facenti parte del gioco, tra cui:
  - personaggio giocabile;
  - componenti della mappa come muri, pavimento e porta;
  - entità di gioco statiche come oggetti collezionabili e ostacoli;
  - entità di gioco dinamiche come nemici e proiettili.
- sistema di difficoltà che permetta di controllare tutti gli aspetti relativi alla logica della partita, tra cui:
  - dimensione della mappa;
  - un range per il numero di ogni entità presente in un livello;
  - punteggio ottenuto dal giocatore.
- possibilità di giocare una partita composta da diversi livelli;
- possibilità di salvare e visualizzare i dati del giocatore, in particolare:
  - poter salvare sul dispositivo il punteggio ottenuto, associato al nome utente registrato;
  - poter visualizzare una classifica dei punteggi comprendente tutti i profili registrati.

3.4 Requisiti non funzionali

L'applicazione realizzata dovrà rispettare i seguenti vincoli relativi alla qualità del sistema:

- *Consistenza*: vista la natura del gioco, comprendente diverse entità dinamiche e in continuo movimento sulla mappa, lo stato del gioco deve essere mantenuto coerente nel tempo, in modo da poter ottenere una fotografia dello stato di ogni entità presente in un dato step logico del gioco.
- *Reattività*: la natura dinamica del gioco impone anche una reattività elevata, sia per quanto riguarda l'aggiornamento dello stato di visualizzazione della mappa, che deve aggiornarsi secondo un ritmo periodico, ma anche per garantire una risposta adeguata ai comandi dell'utente, che deve poter avvenire in una finestra di tempo ragionevole.

3.5 Requisiti implementativi

I requisiti implementativi comprendono:

- la necessità di definire una gerarchia di entità che sia sufficientemente rappresentativa del dominio del gioco in questione; è quindi opportuno scegliere la giusta separazione di concetti per il tipo di *gameplay* che si vuole implementare;
- la necessità di garantire che il gioco si svolga ad un ritmo costante, ovvero di definire una struttura logica coerente per rappresentare il flusso logico del gioco, che rispetti la visualizzazione a schermo, che gestisca per tempo la reazione ai comandi inseriti dall'utente e che garantisca la coerenza delle strutture dati utilizzate, nell'ambito degli step logici che dovranno essere definiti per controllare lo svolgimento della partita;
- In modo da ragionare sul problema da un punto di vista in parte funzionale, il codice creato dovrà fare uso delle proprietà che caratterizzano Scala, che comprendono:
  - unione di programmazione ad oggetti e funzionale;
  - utilizzo di strutture dati immutabili per evitare side-effect;
  - possibilità di scrivere codice più coinciso;
  - possibilità di alzare ulteriormente il livello di astrazione.

  Queste proprietà permettono di creare del codice più facilmente scalabile. Inoltre, il codice dovrà rispettare tutti i requisiti di stile del caso, ed ogni funzionalità del gioco dovrà essere adeguatamente testata con Scalatest.

Capitolo 4

Design architetturale

4.1 Architettura complessiva

Di seguito è riportato un diagramma che illustra le classi principali per la comunicazione tra Model-View-Controller.

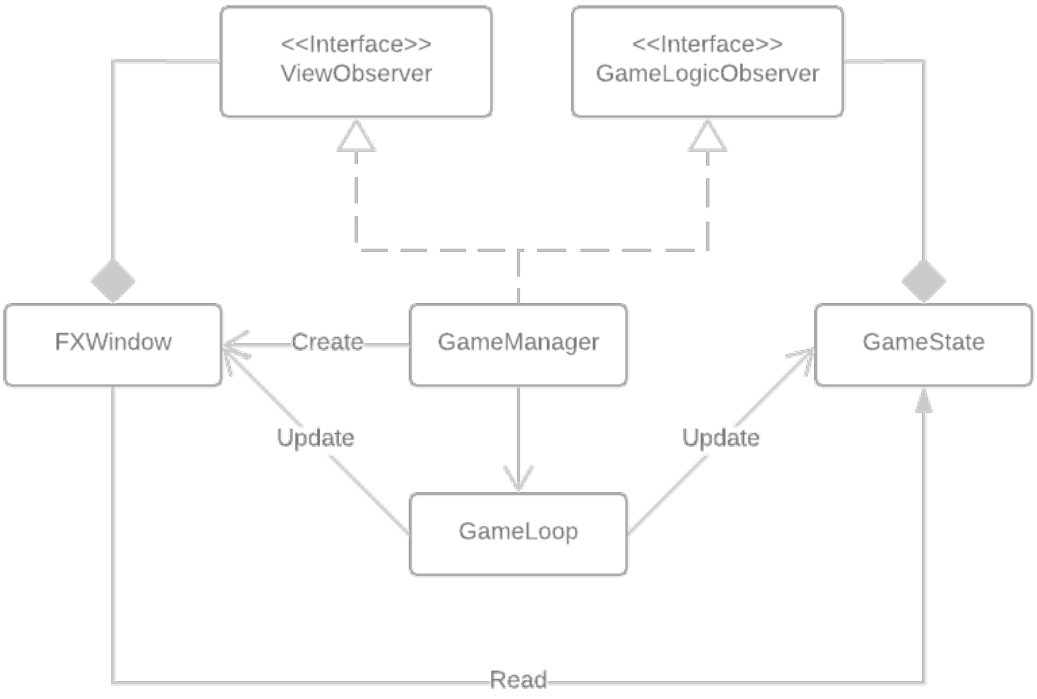

Figura 4.1: Architettura

4.1.1 MVC Pattern

Per lo sviluppo del gioco si è scelto di utilizzare il pattern MVC in quanto si vuole avere una separazione pulita tra la parte logica e la parte di visualizzazione. In questo modo si lascia aperta la possibilità di avere diversi tipi di interfaccia grafica.

Model

Contiene la logica di gioco.

View

Ha il compito di mostrare i dati forniti dal Model. Sarà costituita da due parti: il menù di gioco e l'arena di gioco.

Controller

Fa da ponte tra Model e View, cattura gli eventi generati dalla view utilizzando il pattern Observer. Si occupa della lettura da file e della gestione dell'Event-Loop.

4.1.2 Event-Loop

Si è scelto di utilizzare un event-loop per gestire la ricezione di input utente, l'update della parte logica e l'update della parte di interfaccia grafica. Si è scelto di utilizzare un approccio ad event-loop single-thread, poiché, si adatta bene al tipo di gioco sviluppato. Inoltre l'approccio a event-loop, essendo in esecuzione su un singolo thread e processando un task alla volta, risulta essere thread safe, esulando da problemi implementativi in cui si poteva incorrere.

Capitolo 5

Design di dettaglio

5.1 Organizzazione del codice

Partendo dal design architetturale, il codice è stato diviso nei tre package *gamelogic*, *gamemanager* e *gameview* per rispecchiare il pattern MVC; un quarto package chiamato *utilities* contiene invece funzionalità utili a tutti e tre. Di seguito la descrizione dei principali elementi di ogni package e delle loro funzionalità.

5.1.1 Package gamelogic

Il package *gamelogic* rappresenta il model, ovvero la logica del gioco, e si compone di:

- **Arena**: Questa classe contiene le entità che popolano la mappa in un dato momento della partita, nonchè tutti i metodi necessari per interagire con esse e l'area di gioco. Volendo accedere allo stato della partita in corso, infatti, basterà controllare lo stato delle strutture dati per le entità. Una funzione *generateMap()* viene richiamata, attraverso il controller, nella fase di popolazione iniziale, ovvero all'inizio di ogni livello, e una funzione *updateMap()* invece si occupa dell'aggiornamento dello stato di ogni entità e viene richiamata ad ogni step logico, come si può vedere dal codice in figura 5.1.

```scala
class Arena(val playerName: String, val mapGen: MapGenerator) {
  var player: Player = Player(playerName)
  var hero: Hero = Hero(Position(center, Some(Down)))
  var enemies: Set[EnemyCharacter] = Set()
  var bullets: Set[Bullet] = Set()
  var collectibles: Set[Collectible] = Set()
  var obstacles: Set[Obstacle] = Set()
  var walls: Set[Wall] = for (p <- bounds) yield Wall(Position(p, None))
  val floor: Set[Floor] = for (p <- tiles) yield Floor(Position(p, None))
  var door: Option[Door] = None
  var endedLevel: Boolean = false
  private var lastInjury: Option[EnemyCharacter] = None

  /** Generates a new level. */
  def generateMap(): Unit = {
    mapGen.generateLevel()
    movePlayerOnDoor()
  }

  /** Updates the [[Arena]] for the new logical step. */
  def updateMap(movement: Option[Direction], shoot: Option[Direction]): Unit = {
    checkShoot(shoot)
    checkMovement(movement)
    checkBullets()
    checkEnemies()
    checkDoor()
  }
```

Figura 5.1: Esempio di codice di *Arena*

MapGenerator (figura 5.2), facente parte dei parametri di inizializzazione della classe, incapsula la logica di generazione a partire dalla difficoltà scelta. Implementa il legame tra la difficoltà del gioco (rappresentata da un insieme di valori e range) ed il modo randomico con cui vengono generate le entità.

```scala
/** Encapsulates the logic for generating a new level.
 *  Used by [[Arena]].
 *
 *  @param difficulty the [[Difficulty]] chosen by the user
 */
case class MapGenerator(difficulty: Difficulty.Value) {
  var currentLevel: Int = 0

  /** Generates a new level, populating the [[Arena]] with the resulting [[Entity]]s. */
  def generateLevel(): Unit = {
    incLevel()
    initializeDoor()
    generateEnemies()
    generateCollectibles()
    generateObstacles()
  }
```

Figura 5.2: Esempio di codice di *MapGenerator*

- **GameState**: Un object che incapsula lo stato attuale della partita. La classe *Arena* viene istanziata al suo interno, rendendolo il punto di riferimento per avere accesso al contesto del gioco. Infatti, il controller ne richiama i metodi che incapsula per la gestione della partita, come ad esempio l'avvio del gioco, il richiamo dello step da eseguire, la generazione di un nuovo livello e la gestione dei record.
- **Entity**: Come riportato in figura 5.3, tutti gli elementi del gioco estendono da questa interfaccia. Il trait *Entity* rappresenta una qualsiasi entità sulla mappa di gioco, dotata di una posizione propria. Viene poi esteso per distinguere tra le entità che occupano una posizione fissa (*Collectible*, *Obstacle*, componenti dell'arena) e quelle che invece possono muoversi (*MovableEntity*): queste ultime possiedono metodi per avanzare di posizione all'interno dell'arena, data una direzione di movimento. Il trait *EnemyCharacter* rappresenta i nemici, mentre il trait *LivingEntity* viene applicato alle entità dotate di un proprio punteggio che ne rappresenta la vita a disposizione (*Hero*, *EnemyCharacter*).

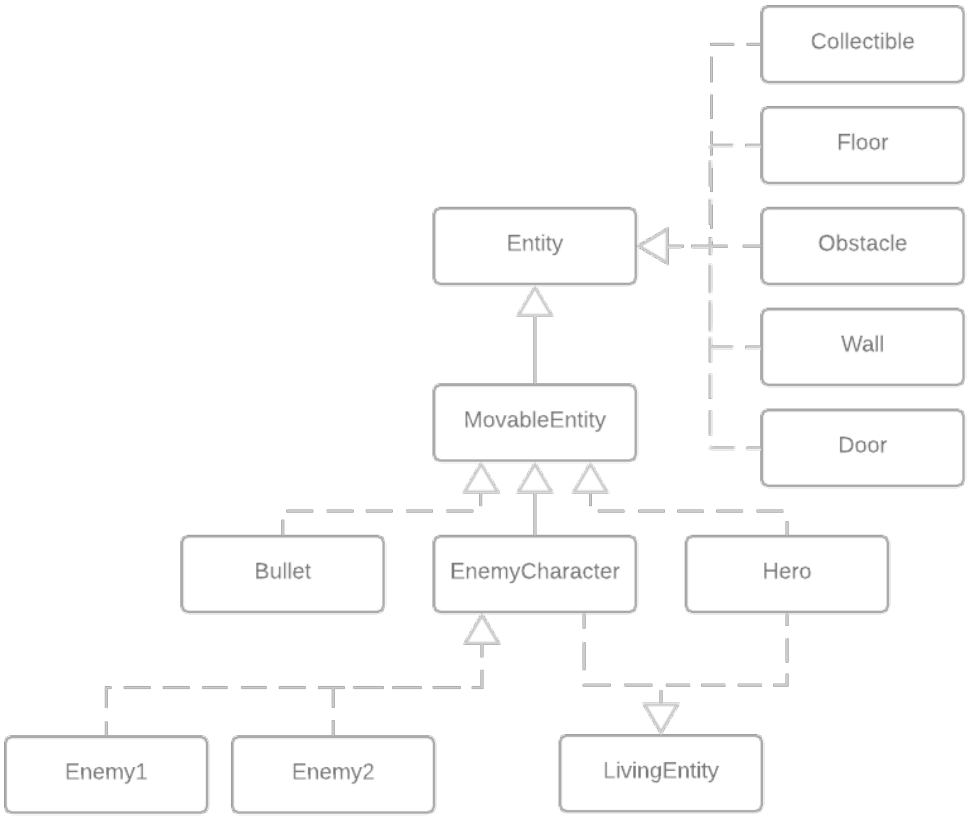

Figura 5.3: Organizzazione della logica di gioco

5.1.2 Package gameview

Il package *gameview* rappresenta la view, responsabile di visualizzare i dati forniti dal model. É composto da un insieme di scene che estendono FXView che provvede al caricamento di eventuali layout. Inoltre, estendono la trait Scene.

Le scene principali sono:

- FXMainScene: rappresenta l'implementazione della scena principale che appare all'utente all'avvio dell'applicazione. Permette di avviare la partita, modificare le impostazioni, guardare i credits o uscire dal gioco.
- FXGameScene: rappresenta l'arena di gioco e comprende tutti gli elementi forniti da *gamelogic*. I vari elementi sono rappresentati da case class di supporto a *FXGameScene* ed estendono tutti da una trait GameElements relativa a una serie di elementi generici.
- FXSettingsScene: rappresenta le impostazioni, tramite essa si può cambiare il nome del player e la difficoltà del gioco.
- FXPlayerRankingScene: rappresenta la classifica del gioco.

- FXCreditsScene: rappresenta la i credits.

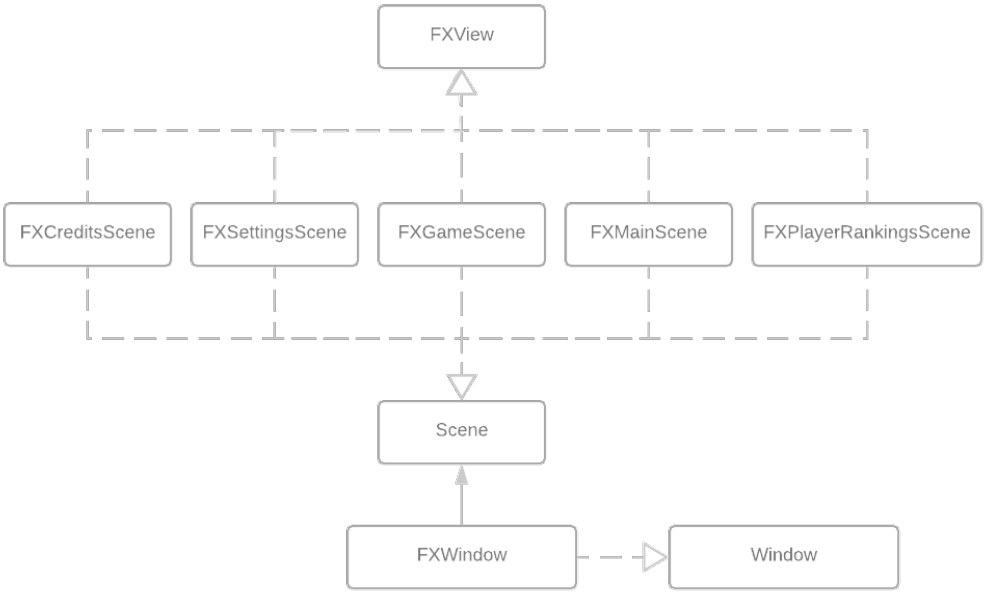

Figura 5.4: Organizzazione del gameview

5.1.3 Package gamemanager

Il package *gamemanager* rappresenta il controller, responsabile dell'interazione tra model e view, e si compone di:

- **GameManager**: Questo object è l'elemento principale del package e svolge alcune tra le funzioni più importanti relative al controllo logico della partita (come nell'esempio in figura 5.5). Viene usato come tramite per la comunicazione tra le classi di model (*GameLogicObserver*) e view (*ViewObserver*), e attraverso esso viene gestita l'inizializzazione della view e il caricamento delle immagini necessarie.

  La concorrenza del programma è gestita tramite un approccio basato su task: il loop principale del gioco, *GameLoop*, viene eseguito su un thread separato in uno specifico *ExecutionContext*, mentre altre funzioni che potrebbero richiedere un tempo elevato (caricamento di elementi della view, immagini e suoni) sono delegate a dei thread tramite il meccanismo delle *Future*, che in questo caso vengono eseguite in un *ExecutionContext* fornito da Scala (global). Dovendo gestire proprio questo aspetto, *GameManager* incapsula un *ExecutionContext*, usato per lanciare *GameLoop*, e il valore di *TimeSlice* necessario per la temporizzazione degli step logici del gioco.

```scala
/** Call to main to initialize the game. */
def initializeGame(primaryStage: Stage) : Future[Unit] = Future{
  val view: Window = FXWindow(primaryStage)
  addObserver( obs = this)
  loadImage().onComplete({
    case Success(_) =>  view.scene_(new Intent(MainScene)); view.showView()
    case Failure(t) => log(t.getMessage)
  })
}

/** Notifies that the game has started. */
def notifyStartGame(): Unit = {
  endGame = false
  GameState.addObserver( obs = this)
  GameState.startGame(playerName, MapGenerator(difficulty))
  ThreadPool.execute(new GameLoop())
  loadWorldRecord()
}

/** Notifies that the game has ended. */
def notifyEndGame(): Unit = {
  endGame = true
  GameState.endGame()
  savePlayerRankings()
  finishGame()
}
```

Figura 5.5: Esempio di codice di *GameManager*

- **GameLoop**: In linea con il design architetturale, questa classe implementa il pattern event-loop per la gestione delle dinamiche di gioco (figura 5.6). Come specificato in precedenza, l'event-loop viene lanciato su un thread separato e svolge un ciclo nel quale esegue uno step logico del programma: questo comprende la gestione dei nuovi eventi ed il conseguente aggiornamento della view. Gli eventi vengono inizialmente organizzati in due code, rispettivamente per i movimenti e per gli spari (*playerMoves* e *playerShoots*), per poi essere letti dall'event-loop ad ogni nuovo step.

```
def run(): Unit = {
  while (!endGame) {
    val startTime: Long = currentTimeMillis()
    gameStep()
    val deltaTime: Long = currentTimeMillis() - startTime
    if (deltaTime < TimeSliceMillis) sleep(TimeSliceMillis - deltaTime)
  }
  finishGame()
}

def gameStep(): Unit = {
  nextStep(checkNewMovement(), checkNewShoot())
  numCycle += 1

  /** Updates the view. */
  if (view.nonEmpty) {
    view.get match {
      case scene: FXGameScene =>
        if (arena.get.endedLevel) {scene.endLevel(); arena.get.endedLevel = false}
        else {scene.updateView()}
      case _ =>
    }
  }
  if (checkNewWorldRecord().isDefined) worldRecord = checkNewWorldRecord().get
  if (arena.get.hero.isDead) notifyEndGame()
}
```

Figura 5.6: Esempio di codice di *GameLoop*

- **ViewObserver**: Interfaccia che rappresenta l'observer della view nell'implementazione del pattern Observer. Viene utilizzata per la comunicazione da view a controller e si occupa di notificare le transizioni tra le varie scene; inoltre, notifica i *KeyEvent* generati dall'utente e catturati dalla *GameView*. Quando un evento viene notificato, viene aggiunto alla relativa coda per essere successivamente consumato dal *GameLoop*, come riportato in figura 6.13.

```
def notifyAction(action: Action): Unit = action.actionType match {
  case Movement => playerMoves = playerMoves :+ action.direction
  case Shoot => playerShoots = playerShoots :+ action.direction
}
```

Figura 5.7: Aggiunge l'azione notificata alla relativa coda

5.2 Pattern utilizzati

Durante l'intero progetto di sviluppo sono stati inclusi diversi pattern e aspetti di degin avanzato.

Primo fra tutti il **Singleton** che è stato ampiamente utilizzato, in quanto Scala tutti gli object sono Singleton. Poi il pattern **Observer**, si vedano le classi *ViewObserver* e *GameLogicObserver*, il pattern **Builder** come la classe *StaticArena* per la generazione di *Arena* in maniera deterministica. È stato utilizzato, inoltre, il pattern **Decorator**, vedasi *LivingEntity*.

5.3 Scelte rilevanti

Prima e durante la fase di sviluppo è stato necessario valutare delle scelte di implementazione, principalmente per cercare di sfruttare al meglio le caratteristiche di Scala, in modo da scrivere del codice intuitivo, compatto e scalabile, cercando anche di trarre il meglio dal paradigma funzionale e di garantire il più possibile immutablità e assenza di side-effect; un altro motivo di scelta principale ha riguardato anche la pesantezza del codice, avendo a che fare con un'applicazione particolarmente dinamica e reattiva. Di seguito sono riportate le scelte più significative che sono state adottate in fase di sviluppo:

- **Case class**: Una scelta fondamentale, in quanto legata all'immutabilità del codice, ha riguardato l'utilizzo delle case class, in particolare per le entità. Queste sono caratterizzate da una posizione che per molte di esse cambia ad ogni step, ma volendo mantenerne lo stato immutabile, si è puntato a dichiararle come case class, in modo da poterle facilmente istanziare nuovamente a seguito di una modifica dello stato, nonchè per la facilità d'uso all'interno dei costrutti match-case. La gestione dello stato delle entità in questo modo è stata anche oggetto di refactoring durante lo sviluppo.

  Altra questione non meno importante riguarda la dichiarazione delle strutture dati nella classe *Arena* (ma anche nel resto del programma), che possono potenzialmente rappresentare il collo di bottiglia per la velocità di esecuzione complessiva dell'applicazione. Seguendo la stessa logica utilizzata per le case class, sono sempre state usate strutture immutabili, dichiarate come var per garantirne la sostituzione in seguito ad un loro aggiornamento. L'immutabilità in questo caso assicura la coerenza nello stato del model.

- **GameLoop**: Il loop di controllo è stato oggetto di rifattorizzazione per quanto riguarda la struttura del ciclo e l'implementazione del suo thread. Il ciclo inizialmente seguiva una logica ricorsiva ed è stato poi convertito in un più semplice while-loop, per motivi meramente legati alla velocità di esecuzione, essendo un punto cruciale per il flusso logico del programma; la classe, invece, è passata dall'estendere *Thread* all'implementare *Runnable*, volendo avere una concorrenza basata sui task.

Capitolo 6

Implementazione

6.1 Lorenzo Chiana

6.1.1 Gamelogic

La parte di progettazione dei vari componenti di base non è spettata a me, ma ho contribuito lo stesso, dal secondo sprint in poi, con diverse modifiche e rifattorizzazioni a classi e metodi del model, a volte anche in pair programming con Meshua Galassi (come per il task "Creazione nemici lato model" nel terzo sprint). Inoltre, contributo significativo all'interno della parte di model sono stati diversi bugfixing nel corso degli sprint. Le varie classi che ho toccato maggiormente sono: *Arena*, *Enemy*, *MapGenerator*, *MovableEntity*, *Obstacle*, *ObstacleTypes* e *Player*.

Arena

In Arena mi sono occuppato dei vari metodi che vanno a controllare se un punto contiene una certa entità come *containsCollectible()*, *containsObstacle()* e così via. Inoltre in vari task sono andato a toccare e rifattorizzare il metodo *checkMovement()* per introdurre la collezione dei bonus e controllo della collisione con nemici e conseguente pardita di vita dell'eroe.

Enemy e MovableEntity

Durante il terzo sprint mi sono occupato insieme a Meshua Galassi della generazione dei nemici, in particolare mi sono occupato della rifattorizzazione del vecchio metodo *canMove()*.

6.1.2 MapGenerator

Mi sono occupato della rifattorizzazione dei metodi *generateEnemies()*, *generateCollectibles()* e *generateObstacles()*.

Obstacle e ObstacleTypes

Inizialmente gli ostacoli non avevano un tipo e non era possibile distirguerli per associargli sprite diverse. Per ovviare a questa problematica ho implementato la possibilità di avere tre diverse tipologie di ostacoli in maniera casuale rifattorizzando il tutto.

Player

Rifattorizzato introducendo la possibilità di avere un record personale.

Dopo le varie rifattorizzazioni

Dopo varie rifattorizzazioni susseguite nei vari sprint del mio contributo più significativo rimane:

- la rifattorizzazione ai metodi *generateEnemies()*, *generateCollectibles()* e *generateObstacles()* all'interno di *MapGenerator*;
- la gestione dei record del giocatore ed aggiornamento della classifica in *Player* e *GameState*;
- la creazione e rifattorizzazione di metodi come *containsCollectible()*, *containsObstacle()*, *containsWall()*, *containsEnemy()* e *containsBullet()* di *Arena*;
- la creazione di *ObstacleTypes*.

6.1.3 Gameview

Durante il primo sprint mi sono occupato della progettazione e realizzazione dei mockup del menù di gioco, andando a creare le principali voci del menù che, ancora nell'ultima versione, sono presenti, quali "*Settings*" e "*Credits*". Inoltre, per quanto riguarda l'interfaccia di gioco, ho realizzato un prima versione base dell'arena di gioco accessibile attraverso la voce del menù "*Start game*". Dal quarto sprint ho introdotto anche la voce "*Player rankings*" con relativa scena. Le varie interfacce dei singoli componenti della view sono stati progettati e realizzati ponendo attenzione a renderli indipendenti dalla piattaforma anche grazie all'utilizzo di file *CSS* e *FXML*. Quest'ultima tipologia di file è stata prodotta grazie all'utilizzo del tool *Scene Builder*.

Inoltre mi sono occupato della generazione grafica dell'arena, andando a creare le sprite delle varie entità dapprima in modo statico e poi, dal secondo sprint, in modo dinamico a seconda delle informazioni contenute nel model. In un secondo momento questa generazione grafica è stata rifattorizzata da Giada Gibertoni, andando a snellire la classe *FXGameScene* e portandola alla struttura che ha oggi 6.8.

Scelta della libreria

Prima della progettazione e implementazione dell'interfaccia mi sono concentrato insieme alla collega Giada Gibertoni sulla ricerca di una libreria grafica da utilizzare. Il primo risultato di questa ricerca è ricaduto su *ScalaFX* che ricalca lo stile di *JavaFX* però in termini molto più Scala-like. D'altro canto ci si è accorti quasi subito che è molto scarna di esempi pratici e per questo motivo alla fine abbiamo abbandonato l'idea di utilizzare questa libreria. La ricerca è proseguita su altre librerie tutte scartate o per il medesimo motivo o perché non davano la possibilità di definire l'interfaccia grafica tramite file esterni, limitando il tutto a dover disegnare la GUI tramite classi. Alla fine si è optato per JavaFX, soprattutto perché offre tanti esempi reperibili online grazie ad una community molto attiva e anche perché offre un'integrazione con file esterni come FXML.

Finestra generica e implementazione in JavaFX

Data questa incertezza iniziale sulla scelta della libreria grafica, sono andato a creare una finestra generica grazie alla quale un eventuale futuro cambio di libreria non avrebbe portato comportamenti inattesi nel comportamento di base della finestra. Una volta definite le funzionalità di base nell'interfaccia *Window*, questa sarà implementata da *FXWindow* che va a creare una finestra conforme agli standard di JavaFX.

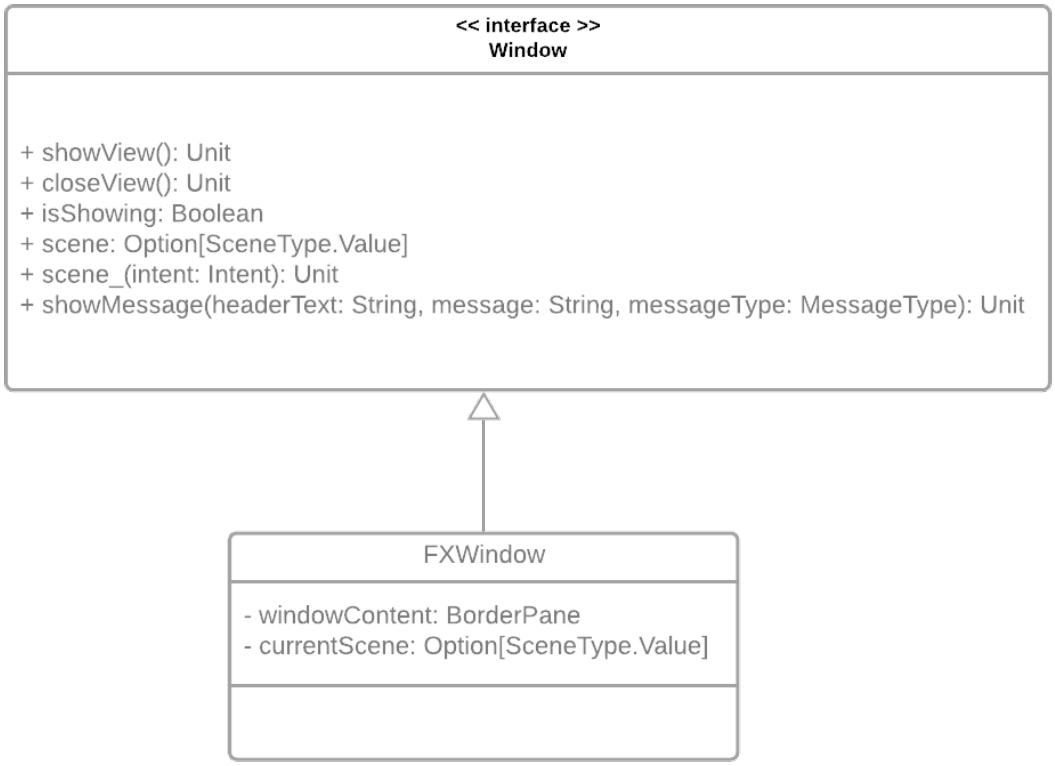

Figura 6.1: Interfaccia Window e FXWindow che la implementa.

Navigazione e scene

La navigazione tra le schermate viene permesso grazie al metodo *scene_()* che va a creare la nuova scena e a notificare all'observer (*GameManager*) che deve cambiare la scena con la nuova. La decisione su che scena creare viene presa sul tipo di scena (*SceneType*), un'*Enumeration* passata come parametro all'*Intent* (classe che rappresenta l'intento di cambiare la scena corrente con un'altra).

Arena di gioco

Mi sono inoltre occupato della creazione dell'interfaccia di gioco, andando a creare visualmente l'arena che il giocatore vedrà mentre gioca, all'interno di *FXGameScene*. Qui ogni punto dell'arena logica viene tradotto in una piastrella alla quale viene assegnata una certa sprite a seconda

della tipologia dell'entità che è presente su tale punto (metodo *createArena()*). Queste piastrelle vengono regolarmente aggiornate ed allineate con la parte logica del programma tramite il metodo *nextStep()* di *GameLoop*. La realizzazione di FXGameScene è stata incrementale durante tutto lo sviluppo del gioco, tant'è che ha subito diverse modifiche e aggiunte da diversi componenti del gruppo.

6.1.4 Gamemanager

La parte di gamemanager, come per la parte di gamelogic, la sono andato a toccare diverse volte, ma solo con piccole modifiche e rifattorizzazioni. I contributi più significativi che ho dato in questa parte del progetto si possono suddividere in: *PreferencesHandler* e *caricamento e salvataggio della classifica di gioco*.

PreferencesHandler

Viene utilizzata per salvare le preferenze dell'utente come il nome e la difficoltà di gioco attraverso l'uso di *Java Preferences API*. Il salvataggio e la serializzazione di queste impostazioni vengono fatte attraverso l'estensione *lift-json-ext* che verrà poi utilizzata anche per salvare la classifica di gioco.

Classifica di gioco

La classifica di gioco viene salvata in un file JSON tramite la *PrintWriter* di *java.io* con la seguente struttura: 6.2

Figura 6.2: Struttura del file rank.json

Questo file viene letto tramite il metodo *loadPlayerRankings()* di *GameManager* andando a creare una mappa che come chiave presenta il nome della difficoltà e come valore un'altra mappa che, a sua volta, come chiave ha il nome del giocatore e come valore il suo record. Questa mappa viene salvata in *GameState* (all'interno di gamelogic) e aggiornata dinamicamente durante il gioco. Alla fine di ogni partita la mappa viene salvata nel file JSON attraverso il metodo *savePlayerRankings()*.

Durante il task relativo alla creazione della classifica ho scelto di adottare la libreria lift-json del framework Lift, perché una delle più utilizzate e consigliate per i progetti Scala.

6.1.5 Utilities

Nei vari sprint a seconda dell'esigenza sono andato a creare:

- Animations: utilizzata per creare un animazione di fade-in o fade-out tra le varie scene;
- Intent: classe che rappresenta l'intento di cambiare la scena corrente con un'altra;
- MessageTypes: un sailed trait che rappresenta i vari tipi di messaggi di una finestra di dialogo;
- ViewLoader: permette il caricamento, e quindi l'utilizzo, di layout in formato *.fxml*;
- WindowSize: contiene le dimensioni della finestra del menù principale e di quella di gioco.

6.1.6 Test

Per quanto riguarda la parte di testing mi sono occupato di andare a testare le classi *Player*, *Hero*, *Enemy* e *Bullet* con *AnyFlatSpec* come tipologia di sintassi.

Per ogni entità sono andato a testare le caratteristiche principali come ad esempio la direzione del movimento, le collisioni, eventuali decrementi o incrementi di vita o di punteggio e così via. L'unico inconveniente di questo approccio era dato dalla casualità di creazione dell'arena di gioco che poteva far fallire alcuni test come, ad esempio, testare il movimento in una certa direzione. In questo caso si poteva presentare il caso in cui, nel punto in cui si doveva muovere l'entità, era presente un'altra entità finendo per colliderci e risultando di consenguenza non essersi mossa. Per ovviare a questo problema ho optato nella creazione di una classe builder *StaticArena* nella quale l'arena generata casualmente viene svuotata e impostata con delle entità in determinati punti passatagli come parametri, permettendo così di creare una scena ad hoc per il test da eseguire.

6.1.7 Environment

Dell'ambiente di sviluppo mi sono occupato di impostare in sbt i plugin *ScalaStyle*, per il controllo dello stile, e *sbt-assembly*, per il deploy del file jar. Inoltre mi sono occupato della configurazione di Travis CI, andando ad impostare la versione di Scala, la directory del progetto, e la versione della JDK da utilizzare.

6.2 Meshua Galassi

Mi sono occupata principalmente dell'implementazione delle funzionalità lato model ma, in alcuni casi, ho collaborato con Giada Gibertoni per definire parti del package gamemanager e gameview. Una parte consistente del mio lavoro ha riguardato soprattutto l'esecuzione di refactor e la risoluzione di bug.

6.2.1 Sviluppo

GameState

Mi sono occupata di definire *GameState*, ovvero un object che rappresenta il mezzo con cui i componenti esterni possono interfacciarsi con gli elementi che costituiscono la logica di gioco, in modo da esporre all'esterno solamente le informazioni che effettivamente sono necessarie all'interazione con essa.

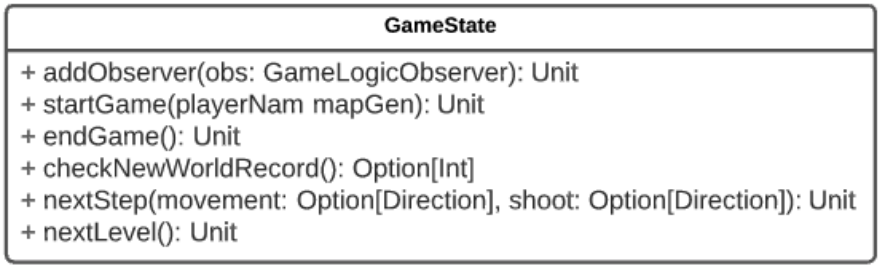

Figura 6.3: Metodi pubblici in GameState

updateMap

Successivamente mi sono occupata della gestione della notifica di movimento del player da parte del controller. In un primo momento, la notifica di tale movimento era processata attraverso un semplice spostamento del personaggio da una posizione a quella vicina nella direzione in cui era stato richiesto lo spostamento.

Collectibles

Ho introdotto le due tipologie di oggetti collezionabili e la loro gestione. La prima permette al giocatore di recuperare una vita nel caso in cui non abbia già il numero massimo di vite possibili, la seconda incrementa il punteggio della specifica partita. Per differenziare le diverse modalità di gioco, all'interno della classe di utility *Difficulty* ho inserito un parametro che permette di generare un diverso numero di bonus in base alla difficoltà scelta. Le due case class di bonus estendono un'unica sealed trait *Collectible* perciò non potranno essere inseriti nuovi bonus esternamente se non andando ad introdurli all'interno dello stesso sorgente. In questo caso, mi sono occupata anche della parte grafica dei diversi collectibles.

Hero

Le caratteristiche principali del personaggio riguardano le vite a disposizione, che verranno incrementate o decrementate in base alle sue azioni, e lo score ottenuto o attraverso l'uccisione di un nemico o raccogliendo un bonus che ha come effetto un'incremento del punteggio.

Enemy

L'entità relativa ai nemici è caratterizzata da un movimento randomico che permette ad ognuno di essi di spostarsi. Inoltre, anch'essi, come l'hero, vengono generati con un totale di cinque vite che verranno poi decrementate ad ogni colpo sparato dal personaggio.

Door

In collaborazione con Lorenzo Chiana, mi sono occupata della terminazione di un livello e la generazione di quello successivo. In particolare, ho sviluppato la parte relativa alla fine del livello che avviene individuando l'assenza di nemici e che provoca creazione in una posizione randomica della *Door* che permette al personaggio di passare al livello successivo.

Una volta che la posizione del player corrisponde con quella della porta di uscita verrà creata una nuova mappa con la porta d'ingresso nella posizione esattamente opposta rispetto a quella da cui l'hero era uscito al livello precedente.

```
door.get.position.point match {
  case Point(0, y) => door = Some(entranceDoor(Position(Point(arenaWidth - 1, y), None)))
  case Point(x, 0) => door = Some(entranceDoor(Position(Point(x, arenaHeight - 1), None)))
  case Point(x, y) if x.equals(arenaWidth - 1) => door = Some(entranceDoor(Position(Point(0, y), None)))
  case Point(x, y) if y.equals(arenaHeight - 1) => door = Some(entranceDoor(Position(Point(x, 0), None)))
}
```

Figura 6.4: Creazione della porta d'ingresso nella posizione opposta a quella di uscita nel livello precedente.

All'inizio di una partita, viene creata una porta fittizia, anch'essa in maniera randomica, in cui viene posizionato il giocatore. Il giocatore, finchè si trova all'intero della porta d'ingresso, non può sparare ai nemici.

```
private def movePlayerOnDoor(): Unit = {
  door.get.position.point match {
    case Point(0, _) => hero = hero.moveTo(Position(door.get.position.point, Some(Right)))
    case Point(_, 0) => hero = hero.moveTo(Position(door.get.position.point, Some(Down)))
    case Point(x, _) if x.equals(arenaWidth - 1) => hero = hero.moveTo(Position(door.get.position.point, Some(Left)))
    case Point(_, y) if y.equals(arenaHeight - 1) => hero = hero.moveTo(Position(door.get.position.point, Some(Up)))
  }
}
```

Figura 6.5: Posizionamento del giocatore all'interno della porta d'ingresso.

6.2.2 Refactor

Refactor view observer

Durante un primo refactor mi sono occupata di unire in un unico object *GameManager* e nella relativa trait *ViewObserver*, tutte le funzionalità relative al settaggio delle diverse scene ed i metodi che permettono l'interazione tra le classi appartenenti al package di view e quelle appartenenti al controller.

In questo modo ho potuto ottenere un unico punto d'ingresso al controller che avviene attraverso la notifica degli eventi agli observers.

Inoltre, ho riunito all'interno di un unico trait *Scene* la gerarchia composta delle diverse scene che in precedenza estendevano ognuna un diverso trait.

Ho scelto di rifattorizzare il codice in questo modo al fine di ottenere un accesso più uniforme alle diverse classi.

Refactor gamelogic

Una volta raggiunta una composizione del progetto contenente tutte le funzionalità che ci eravamo prefissati, osservando il quadro generale della logica di gioco ho notato la presenza di diverse interrelazioni tra le entità che la componevano e lo scarso utilizzo dell'immutabilità all'interno delle entità.

Ho, quindi, comiciato una fase di refactor che ha interessato tutto il package relativo al model sfruttando il principio di immutabilità e il meccanismo di match case all'interno degli elementi di base. Successivamente, insieme a Giada Gibertoni ci siamo occupate dell'integrazione di queste nuove entità nel resto dell'applicazione.

Questo refactor ha portato a diversi cambiamenti:

- Si è introdotto un nuovo elemento nella gerarchia di entità che riguarda le entità con una vita (*LivingEntity*). Si tratta di un decorator che permette di introdurre il concetto di vita, con il conseguente incremento e decremento, nelle entità relative ai nemici ed all'eroe.

- Si è creata una separazione tra i due concetti principali che erano contenuti all'interno della classe *Player* generando così due diverse classi: *Hero* contenente i dati relativi alla specifica partita (vita, punteggio e posizione del personaggio) e *Player* contenente invece i dati relativi al particolare giocatore (nome e record).

- I controlli relativi alla possibilità per un'entità di muoversi o meno sono stati spostati all'interno della classe *Arena* che è l'unica che ha la visione completa delle posizioni in cui sono collocate tutte le entità.

- Nel tentativo di rendere le entità più immutabili possibile, l'individuazione di una nuova posizione è stata spostata all'interno della classe di utility *Position*, la quale restituirà direttamente una nuova posizione ad ogni richiesta di cambiamento.

- Seguendo la stessa logica di immutabilità, per ogni classe base del game logic, ogni volta che ne deve cambiare lo stato, viene restituito un nuovo oggetto della medesima classe con lo stato modificato.

6.3 Giada Gibertoni

La maggior parte del mio lavoro si è concentrata sull'implementazione della parte *GameView* e della parte *GameManager*, ho aiutato anche i miei compagni nella parte relativa alla *GameLogic*. Inoltre, ho partecipato, insieme ai miei colleghi, alla risoluzione di vari bug e alla rifattorizzazione del codice. Infine, insieme a Lorenzo Chiana, ho impostato il file *build.sbt*.

6.3.1 GameManager

GameManager

Principale elemento relativo al controller ed ha varie funzionalità. Una di queste è *initializeGame*. per l'inizializzazione del gioco: questo metodo permette di inizializzare la parte di *GameView*, inoltre permette il caricamento di tutte le sprites utilizzate successivamente nella partita.

GameManager rappresenta un ponte tra la *GameView* e il *GameLogic*, in quanto ascolta gli eventi compiuti dall'utente lato *GameView* e li comunica al *GameLogic*. Per osservare gli eventi della view il *GameManager* implementa il *ViewObserver*, che verrà approfondito in un'altra sezione. inoltre il *GameManager* implementa il *GameLogicObserver*, si è scelto di usare il pattern observer per permettere al *GameLogic* di notificare ogni qual volta viene eseguita un azione. Questo viene poi sfruttato dal GameManager per la riproduzione dei suoni.

GameLoop

Si è scelto di utilizzare un approccio ad event-loop single-thread per gestire le dinamiche di gioco. In accordo con i colleghi, questo approccio è stato scelto poichè si adatta bene al tipo di gioco sviluppato: il *GameLoop* ha in particolare due task fondamentali da svolgere in sequenza, ovvero deve aggiornare la parte di *GameLogic* e successivamente aggiornare la parte di *GameView*. Inoltre l'approccio a event-loop, essendo in esecuzione su un singolo thread e processando un task alla volta, risulta essere thread safe. Quindi questa classe è incaricata di implementare l'Event-loop: all'avvio della partita il *GameManager* fa partire un thread separato sul quale viene lanciato un event-loop incaricato di aggiornare la parte view ed, ad ogni ciclo, far compiere uno step alla parte logica. Per gestire gli eventi generati dall'utente vengono utilizzate due queue che tengono traccia degli spari e dei movimenti compiuti dall'utente e non ancora processati dall'event-loop. Gli eventi vengono poi recuperati ad ogni step come riportato in figura 6.6, e passati al *GameState* che li elaborerà.

```scala
def checkNewMovement(): Option[Direction] = {
  playerMoves.length match {
    case 0 => None
    case _ =>
      val direction = playerMoves.dequeue._1
      playerMoves = playerMoves.dequeue._2
      direction
  }
}
```

Figura 6.6: Recupera un movimento notificato

Una struttura generale della gestione degli eventi è riportata in figura 6.7: *FXGameScene* rappresenta l'emitter degli eventi, che vengono poi, tramite *GameManager*, salvati in due queue da cui il *GameLoop* recupererà i singoli eventi e li darà in pasto al *GameState*.

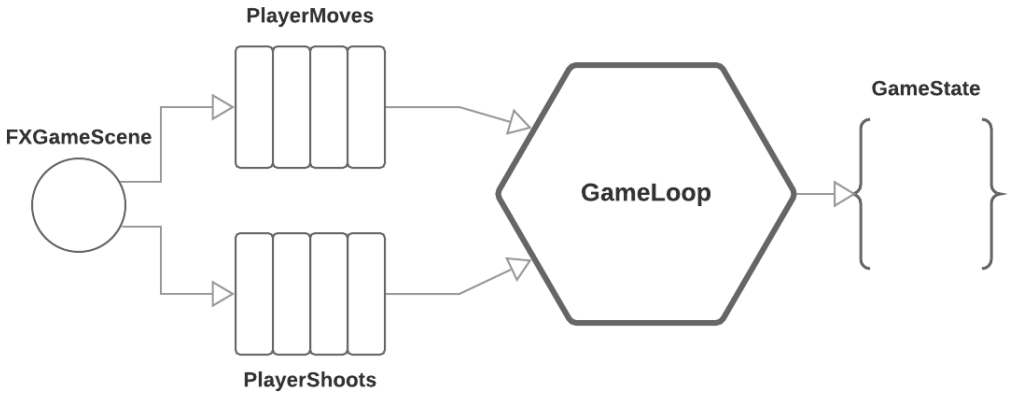

Figura 6.7: Struttura gestione eventi

ImageLoader

Come accennato in precedenza, esso permette, all'apertura dell'applicazione, di caricare tutte le sprites utilizzate successivamente, in modo da limitare le letture da file durante l'esecuzione del gioco; questa operazione è fatta sfruttanto il meccanismo delle Future, quindi al suo completamento verrà visualizzata la finestra dell'applicazione. Durante l'esecuzione del gioco invece permette alla *GameView* di recuperare le *Image* di cui ha bisogno.

SoundLoader

Gestisce la riproduzione dei suoni all'interno del gioco. Quando viene richiesta la riproduzione di un suono, viene generata una nuova *Future* che creerà lo stream e quindi il clip per poi riprodurlo.

6.3.2 GameView

Lato view ho implementato la mappa di gioco, con il supporto di Meshua Galassi e Lorenzo Chiana. La mappa di gioco, rappresentata dalla case class *FXGameScene*, viene aggiornata dal *GameManager* ad ogni step. Mantiene una serie di *Map* che rappresentano i vari elementi che devono essere disegnati e spostati ad ogni step. La collezione mappa l'id dell'elemento alla propria *ImageView*. Ad ogni update si controlla se gli elementi hanno subito variazioni dall'update precedente, in caso affermativo vengono ridisegnate nella posizione corretta o eliminate. Mantenendo questa serie di *Map* si evita di svuotare e ripopolare l'arena di gioco ad ogni step. In questo modo non si rischia di sovraccaricare il *JavaFX Application Thread*.

Inizialmente *FXGameScene* conteneva la gestione di tutti gli elementi da visualizzare. Successivamente si è scelto di rifattorizzarla per renderla meno prolissa, più organizzata e più leggibile. Durante la rifattorizzazione sono state create le seguenti case class di supporto:

- **ArenaRoom**
- **Bullets**
- **Collectibles**
- **Enemies**
- **Player**

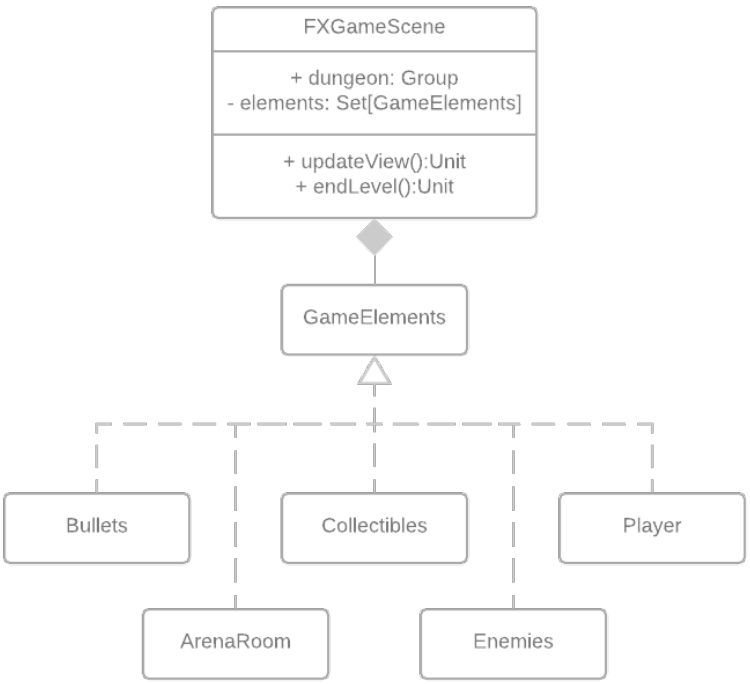

Figura 6.8: Organizzazione dell'arena di gioco

Come si evince dalla figura 6.8, tutte estendono dalla trait *GameElements*, che vuole rappresentare genericamente qualsiasi elemento che deve essere disegnato e aggiornato nell'arena di gioco. L'object *GameElements* fornisce metodi di util, utili per i vari elements come riportato in figura 6.9

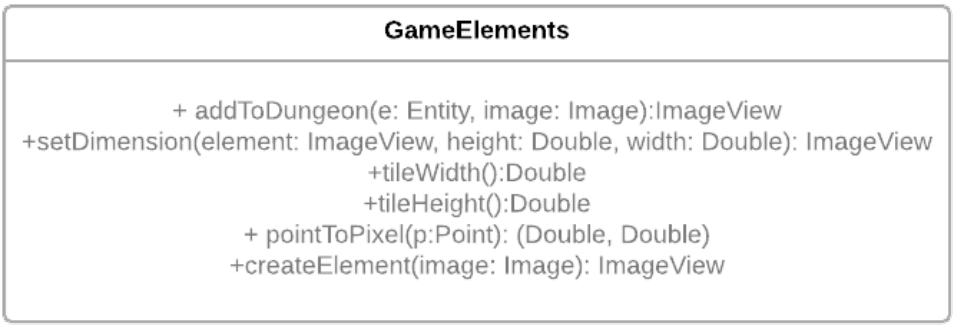

Figura 6.9: Metodi util di GameElements

6.3.3 GameLogic

Lato model ho contribuito nella risoluzione di bug e rifattorizzazioni. Ho sviluppato la parte riguardante le *Bullet*.

Ho ampliato *Obstacle*, che precedentemente era rappresentato da un unico tipo, in 3 tipologie diverse, successivamente rifattorizzate da Lorenzo Chiana.

Ho implementato alcune funzionalità di *Arena* come *checkShoot*, *checkBullets*, *checkHitEnemies*, *playerInjury*. Inizialmente queste funzionalità erano contenute all'interno di *updateMap* ma, successivamente, sono state rifattorizzate e suddivise in metodi separati da Giacomo Pasini.

Infine ho aiutato Meshua Galassi nel refactor di *GameLogic*.

6.3.4 Utilities

Per quanto riguarda le utilities ho sviluppato le seguenti:

Action

Rappresenta i tipi di azioni che può compiere il giocatore, ovvero il movimento e lo sparo.

Difficulty

Rappresenta le varie difficoltà che il gioco può avere, contiene una serie di parametri utilizzati all'interno del gioco. *Difficulty* è stata poi ampliata, da me e dai miei colleghi, durante tutto lo svolgimento del progetto, per introdurre nuovi parametri.

ImageType

Rappresenta un enumerazione contenente tutti i tipi di sprite utilizzati all'interno del gioco con i relativi path.

SoundType

Come ImageType, rappresenta un enumerazione contenente tutti i tipi di suoni utilizzati all'interno del gioco con i relativi path.

6.4 Giacomo Pasini

Ho svolto la maggior parte del mio lavoro programmando il package *gamelogic*, concentrandomi inizialmente sulla definizione di una gerarchia di entità adatta al contesto del gioco, poi lavorando principalmente sulla gestione della mappa (*Arena*) e delle interazioni tra entità e ambiente di gioco. Per quanto riguarda gli altri package, ho aiutato con la gestione dei task in *gamemanager* (*GameLoop*) e inizialmente con la visualizzazione della mappa in *gameview* (*FXGameScene*).

6.4.1 Entity

Ho scelto di realizzare una gerarchia di entità che potesse comprendere tutti i componenti importanti ai fini della partita. L'unica caratteristica che è stata quindi scelta per l'entità più generica è solo la sua posizione (inoltre, la direzione specificata nella posizione è opzionale). Una successiva scelta di suddivisione, che è sembrata la più coerente con il dominio del gioco, ha portato a *MovableEntity*, che rappresenta tutto ciò capace di compiere un movimento. A seguire sono stati specificati anche *EnemyCharacter* per raggruppare i diversi tipi di nemici e *LivingEntity* per quanto riguarda i personaggi con un punteggio di vita.

In seguito alla specifica di *MovableEntity* ho partecipato allo sviluppo dei metodi relativi al movimento delle entità e a tutti quelli conseguentemente necessari per il controllo e la gestione dell'area di gioco.

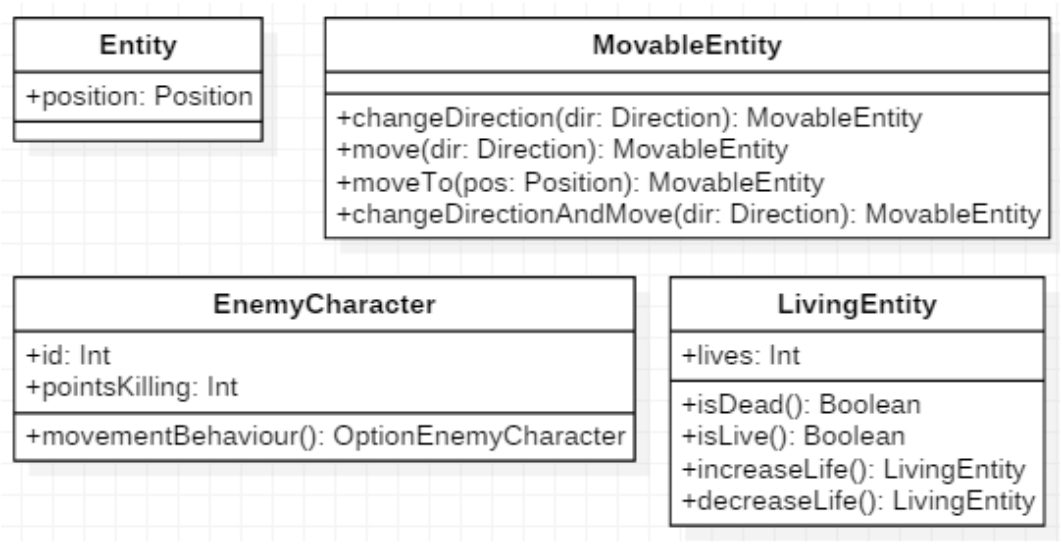

Figura 6.10: Trait delle entità del gioco

6.4.2 Arena

Insieme alla entità del gioco, ho iniziato a realizzare un semplice sistema di coordinate per gestire movimenti nell'arena, tenendo conto delle premesse fatte in fase di design:

- la mappa di gioco deve essere rappresentata logicamente da una griglia;
- la mappa di gioco deve avere un contorno non attraversabile (*Wall*) e un'area di gioco interna (*Floor*);
- ogni entità all'interno dell'arena è caratterizzata da una posizione (*Position*);
- ogni posizione si compone di un punto 2D (*Point*) e di una direzione opzionale (*Option[Direction]*);
- ogni entità può muoversi solo in quattro direzioni;
- le coordinate possono essere solo numeri interi positivi.

Ho poi partecipato all'implementazione della classe *Arena*, contenente tutte le entità che popolano la mappa durante il gioco. Alcuni componenti, vista la loro natura statica, vengono inizializzati con le loro posizioni in fase di creazione, mentre il resto viene gestito dalle funzioni *generateMap()* e *updateMap()*. Si noti la differenza tra *Player*, che rappresenta l'utente, e *Hero*, che invece rappresenta il personaggio giocabile. La classe ha come parametri il nome del profilo associato alla partita corrente ed un *MapGenerator*, a sua volta inizializzato con la difficoltà della partita.

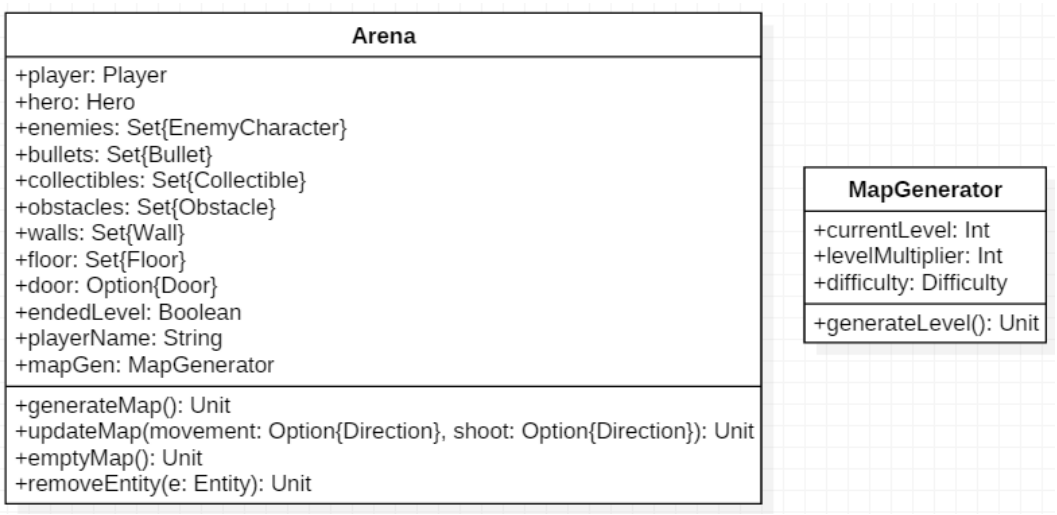

Figura 6.11: La classe *Arena*

Vista l'importanza logica di *Arena*, è anche necessario fornire metodi esterni generici per permettere alle altre classi di lavorare con l'area di gioco. Per lo scopo, un companion object contiene tutti i metodi che possono essere utili alle entità esterne per controllare lo stato del gioco, nonchè per il testing. Una parte fondamentale dello sviluppo è stata la scelta del tipo di funzioni da implementare in questo caso, che ha portato a refactoring in parti diverse dell'implementazione, dovuto soprattutto alla separazione tra funzioni della classe e del companion object e a scelte legate alle strutture dati utilizzate.

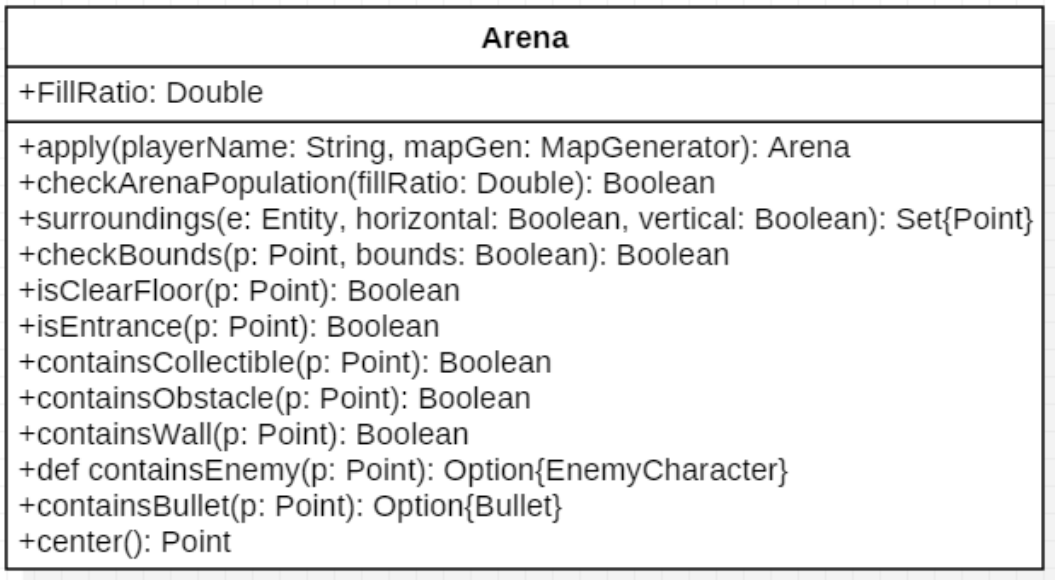

Figura 6.12: Il companion object di *Arena*

6.4.3 Utilities

Nello sviluppare le cose sopracitate ho dovuto creare classi di utilità facenti parte del package *utilities*, tra cui:

- **Direction**: Combinazione di sealed trait e case object per enumerare le possibili direzioni in cui un'entità può muoversi o essere rivolta.
- **Position**: La case class *Position(point: Point, dir: Option[Direction])* rappresenta la posizione di un'entità nell'arena;
- **Point**: La case class *Point(x: Int, y: Int)* rappresenta un punto in 2D e serve anche a comporre Position.
- **Range**: La case class *Range(min: Int, max: Int)* rappresenta i range di valori ammissibili per la generazione delle entità in *MapGenerator*.
- **ImplicitConversions**: Object che contiene i metodi di conversione impliciti per semplificare la scrittura del codice. Il più importante ed utilizzato permette di poter instanziare un *Point* scrivendo una semplice tupla (x, y).

6.4.4 Testing

Per la mia parte di testing ho implementato la classe *ArenaTest*, che fa uso di *Scalatest* (*FlatSpec*) e si occupa di verificare la coerenza delle operazioni di generazione e aggiornamento della mappa, controllando anche che le entità rispettino le regole dell'ambiente di gioco, verificandone il corretto posizionamento, e che vengano generate nel range associato alla difficoltà scelta.

6.5 Pair Programming - Meshua Galassi, Giada Gibertoni

6.5.1 Interazione tra GameView e GameManager

Abbiamo deciso di utilizzare il pattern Observer per dare la possibilità alla view di notificare gli eventi.

Abbiamo quindi creato la trait *ViewObserver* che rappresenta l'observer della view ed è utilizzata per la comunicazione da view a controller. Si occupa di notificare le transizioni tra le varie scene. Inoltre, notifica i *KeyEvent* generati dall'utente e catturati dal *GameView*: quando un evento viene notificato, viene aggiunto alla relativa coda per essere, successivamente, consumato dal *GameLoop* come riportato in figura 6.13.

```
def notifyAction(action: Action): Unit = action.actionType match {
  case Movement => playerMoves = playerMoves :+ action.direction
  case Shoot => playerShoots = playerShoots :+ action.direction
}
```

Figura 6.13: Aggiunge l'azione notificata alla relativa Queue

6.5.2 Testing Controller

Abbiamo creato i test riguardanti la parte del controller, utilizzando *AnyFlatSpec* come tipologia di sintassi. Abbiamo testato:

- La creazione della mappa di gioco all'avvio della partita.
- Il corretto riempimento delle queue di eventi relative al movimento e allo sparo.
- Il corretto dequeue degli eventi dalle due queue.
- La fine del gioco.

6.5.3 Prolog

Abbiamo implementato il comportamento dei nemici utilizzando Prolog. Questa parte non è presente nella release finale in quanto, per mancanza di tempo, non è stata perfettamente integrata e comporta ancora qualche problema. Si trova, però, nel branch *Prolog*.

Sono stati creati due diversi tipi di nemici, distinti solamente dal loro *movementBehaviour*, di seguito riportati:

EnemyWithRandomMove

Questa tipologia di nemico ha lo stesso behaviour implementato dai nemici presenti nella release attuale, come si evince dalla figura 6.14.

```
%move(+X1, +Y1, +Value, -XY) :- Value corrisponde ad un valore passato in input
%che permette di scegliere randomicamente una direzione
move(X1, Y1, 0, XY) :- X2 is X1+1, XY = (X2,Y1).
move(X1, Y1, 1, XY) :- X2 is X1-1, XY = (X2,Y1).
move(X1, Y1, 2, XY) :- Y2 is Y1+1, XY = (X1,Y2).
move(X1, Y1, 3, XY) :- Y2 is Y1-1, XY = (X1,Y2).

%calc_point(+X, +Y, +Non_walkable_tiles, -Point)
calc_point(X, Y, Non_walkable_tiles, Point) :- rand_int(4, Rnd), move(X,Y, Rnd, XY),
    (member(XY, Non_walkable_tiles) -> calc_point(X, Y, Non_walkable_tiles, XY); Point = XY).
```

Figura 6.14: Random behaviour in Prolog

EnemyWithLeftRightMove

Questa tipologia di nemico si muove da destra verso sinistra e viceversa nelle caselle in cui può muoversi, invertendo il proprio senso di marcia quando incontra un ostacolo, come riportato in figura 6.15.

```
%move(+X1, +Y1, +Direction, -XY)
move(X1, Y1, right, XY) :- X2 is X1+1, XY = (X2,Y1).
move(X1, Y1, left, XY) :- X2 is X1-1, XY = (X2,Y1).

%calc_point(+X, +Y, +Non_walkable_tiles, +Direction, -Point, -Dir)
calc_point(X, Y, Non_walkable_tiles, right, Point, Dir) :- move(X,Y, right, XY),
      (member(XY, Non_walkable_tiles) -> calc_point(X, Y, Non_walkable_tiles,left, Point, Dir); (Point = XY, Dir = right)).
calc_point(X, Y, Non_walkable_tiles, left, Point, Dir) :- move(X,Y, left, XY),
      (member(XY, Non_walkable_tiles) -> calc_point(X, Y, Non_walkable_tiles,right, Point, Dir); (Point = XY, Dir = left)).
```

Figura 6.15: LeftRight behaviour in Prolog

Problemi riscontrati

Con l'introduzione di Prolog, arrivati ad un certo numero di nemici, abbiamo riscontrato che l'applicazione mostra pesanti lag. Pensiamo che sia dovuto all'architettura dell'event-loop a single-thread. Per risolvere questo problema si dovrebbe rendere il clacolo delle nuove posizioni dei nemici, quindi lo svolgimento delle query Prolog, concorrente.

6.6 Pair Programming - Lorenzo Chiana, Giacomo Pasini

6.6.1 Integrazione tra Arena ed interfaccia grafica

Durante il primo sprint abbiamo lavorato a stretto contatto dato che uno si occupava della creazione dell'interfaccia grafica e l'altro della creazione dell'arena di gioco a livello logico.

Il grosso problema da risolvere era quello di mappare la posizione logica delle varie entità all'interno di un'interfaccia grafica. Dopo varie proposte si è arrivati alla soluzione attuale: dare ad ogni difficoltà una grandezza logica [6.16] dell'arena di gioco per poi andare a suddividere la finestra grafica in piastrelle (Tile) di dimensione pari al rapporto tra la grandezza della finestra fisica e quella logica [6.17].

Così facendo si può facilmente mappare un punto logico (*Point*) in un pixel [6.18].

```
/** Represents the various difficulties that the game can have (Easy, Medium, Hard or Extreme). */
object Difficulty extends Enumeration {
  val Easy: DifficultyVal = DifficultyVal(
    arenaWidth = 12,
    arenaHeight = 8,
    enemiesRange = Range(1, 3),
    collectiblesRange = Range(3, 5),
    maxLife = 5,
    obstaclesRange = Range(1, 3),
    bonusScore = 40,
    obstacleDimension = Range(1, 2),
    levelThreshold = 8
  )
  val Medium: DifficultyVal = DifficultyVal(
    arenaWidth = 18,
    arenaHeight = 12,
    enemiesRange = Range(5, 8),
```

Figura 6.16: Dimensione logica delle difficoltà

```scala
/**
 * Defines the width of each tile that will make up the arena
 *
 * @return the width of the tile
 */
def tileWidth: Double = Game.width / arenaWidth

/**
 * Defines the height of each tile that will make up the arena
 *
 * @return the height of the tile
 */
def tileHeight: Double = Game.height / arenaHeight
```

Figura 6.17: Dimensione tile

```scala
/** Converts a logic [[Point]] to a pixel for visualization purposes.
 *
 *  @param p the [[Point]] to convert
 *  @return a tuple of coordinates in pixels
 */
def pointToPixel(p: Point): (Double, Double) = (p.x * tileWidth, p.y * tileHeight)
```

Figura 6.18: Mappatura tra Point e pixel

Capitolo 7

Retrospettiva e commenti finali

Analizzando il nostro percorso di sviluppo, ci siamo accorti che si potrebbero introdurre migliorie sia per quanto riguarda la parte implementativa che per quanto riguarda quella organizzativa.

Punto di vista organizzativo

Nonostante abbiamo fatto una buona pianificazione, seguendo la metodologia scrum ed utilizzando Trello, qualche volta non sono state rispettate le linee guida previste per la sprint. Ovvero, a volte non sono state rispettate le urgenze indicate nei task in quanto venivano svolti prima task con un grado di urgenza inferiore.

Inoltre, sarebbe stato più appropriato utilizzare una metodologia git-flow basata su fork/pull request poichè, in questo modo si sarebbe introdotto un ulteriore controllo sulla qualità del codice caricato all'interno del repository.

Punto di vista implementativo

Avendo inizialmente sottovalutato la complessità dei calcoli presenti all'interno del gamelogic, è stato scelto un approccio di tipo single thread. Guardando lo stato finale del progetto, la complessità del gameLogic è salita, quindi sarebbe stato meglio se le computazioni al suo interno fossero state sviluppate concorretemente. In questo modo, si sarebbero evitati, probabilmente, anche i problemi nati con l'introduzione del codice Prolog.