Relazione progetto “Galactic Adventures”

Federico Pettinari, Alessandro Oliva, Andrea Betti

30 aprile 2018

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software proposto consiste nella realizzazione di un videogioco a piattaforme a scorrimento orizzontale dove il giocatore dovrà vestire i panni di un alieno e completare diversi livelli evitando ostacoli, combattendo nemici e raccogliendo monete per aumentare il proprio punteggio.

Requisiti funzionali

- Il gioco dovrà essere diviso in più livelli (chiamati anche mondi).
- All'avvio dell'applicazione verrà mostrato il menu principale attraverso il quale sarà possibile:
  - Iniziare una nuova partita.
  - Continuare il gioco ripartendo dall'ultimo livello giocato.
  - Terminare l'applicazione.
- Il giocatore potrà muoversi orizzontalmente all'interno del livello e saltare nel caso sia a contatto con una superficie sottostante.
- Al giocatore sarà abbinato un indicatore di salute che potrà calare in caso venga ferito da entità maligne od ostacoli. A vita esaurita si dovrà ricominciare il livello o tornare al menu principale.
- Nei livelli saranno dislocati alcuni oggetti collezionabili o interattivi:
  - Monete di diverso valore che aumenteranno il punteggio di livello;

  - Chiavi di diverso colore che consentiranno al giocatore di attraversare determinati ostacoli (che chiameremo lucchetti) dello stesso colore.
  - Porte aperte o chiuse attraverso le quali si completerà il livello.
  - Leve che potranno essere usate per aprire porte chiuse.

- I livelli dovranno contenere personaggi nemici in grado di muoversi e ferire il giocatore spingendolo via.
- Al completamento di un livello verrà visualizzata una schermata mostrante un indicatore di punteggio ottenuto.
- Il punteggio verrà calcolato in base ad azioni di gioco quali il raccoglimento di oggetti (in special modo le monete) e l'uccisione di nemici.
- Salvataggio dei progressi di gioco a livello completato. E' sufficiente che l'applicazione si ricordi l'ultimo livello completato per poter proseguire.
- Opzionalmente il gioco potrà offrire diversi livelli di difficoltà selezionabili prima dell'inizio di una partita.

Requisiti non funzionali

- Minima performance sufficiente per poter giocare una partita.

1.2 Analisi e modello del dominio

Il fulcro del gioco è la simulazione di un mondo bidimensionale con le proprie leggi fisiche e le interazioni fra le molteplici entità che lo compongono allo scorrere del tempo. Tutte le entità posseggono quindi un corpo individuato da una posizione ed una dimensione all'interno di tale mondo. Dal momento che possono muoversi, entrare in contatto ed influenzarsi a vicenda, i corpi devono poter rispondere a stimoli (impulsi) esterni. Le entità sono ulteriormente caratterizzate dalla presenza (o assenza) di una serie di macro componenti personalizzati quali:

- Vita: in particolare giocatore e nemici avranno una vita che potrà essere danneggiata fino all'eventuale morte.
- Movimento: determina come un'entità si muove nel mondo. Ad esempio il giocatore ed alcuni nemici potranno camminare e saltare, altri volare.

- Arma: uno dei requisiti è che giocatore e nemici si danneggino quando entrano in contatto ma non vorremmo precluderci la possibilità, in futuro, di modellare anche armi a distanza.
- Inventario: per poter trasportare chiavi e monete.
- Cervello: determina il comportamento dell'entità ed usa l'eventuale arma. Ha una sua Personalità (ad esempio il giocatore è buono mentre i nemici sono malvagi).

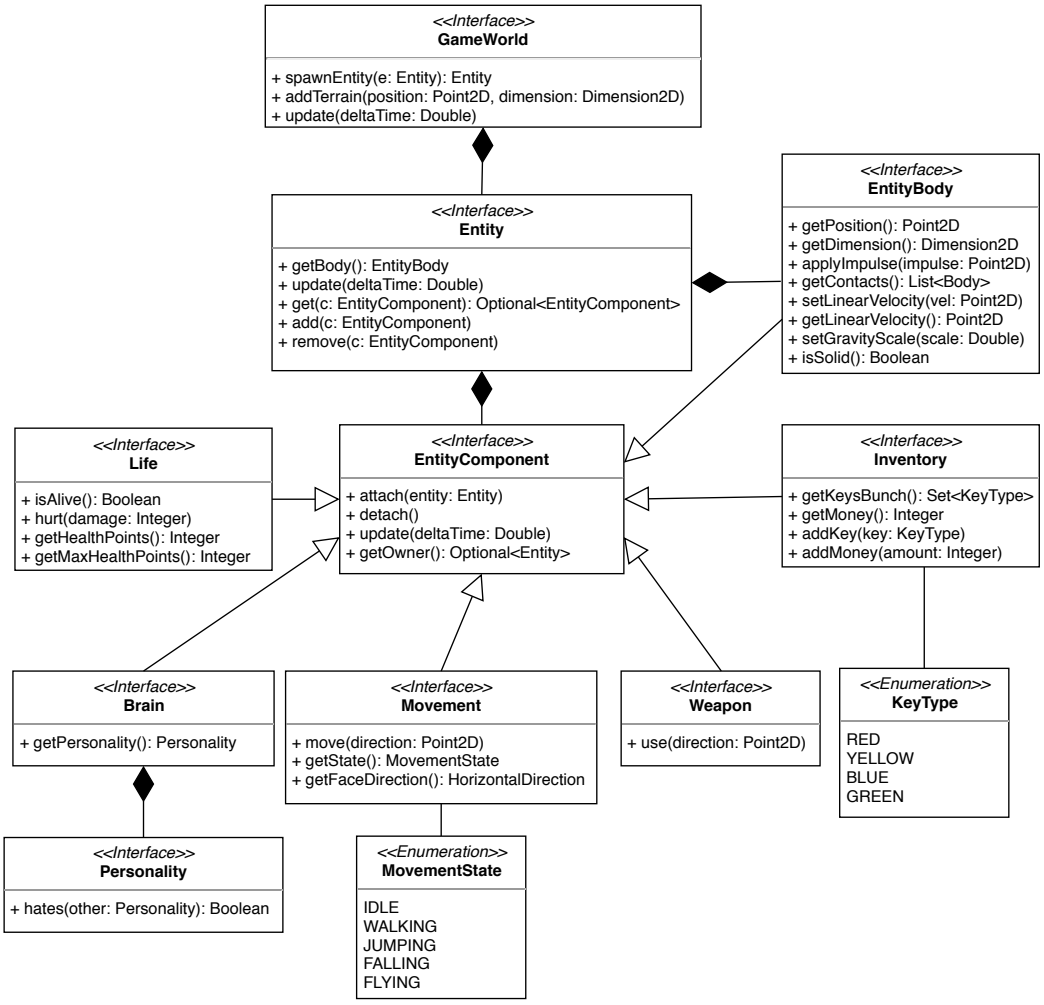

Figura 1.1: Schema del modello Entity - Components

Capitolo 2

Design

2.1 Architettura

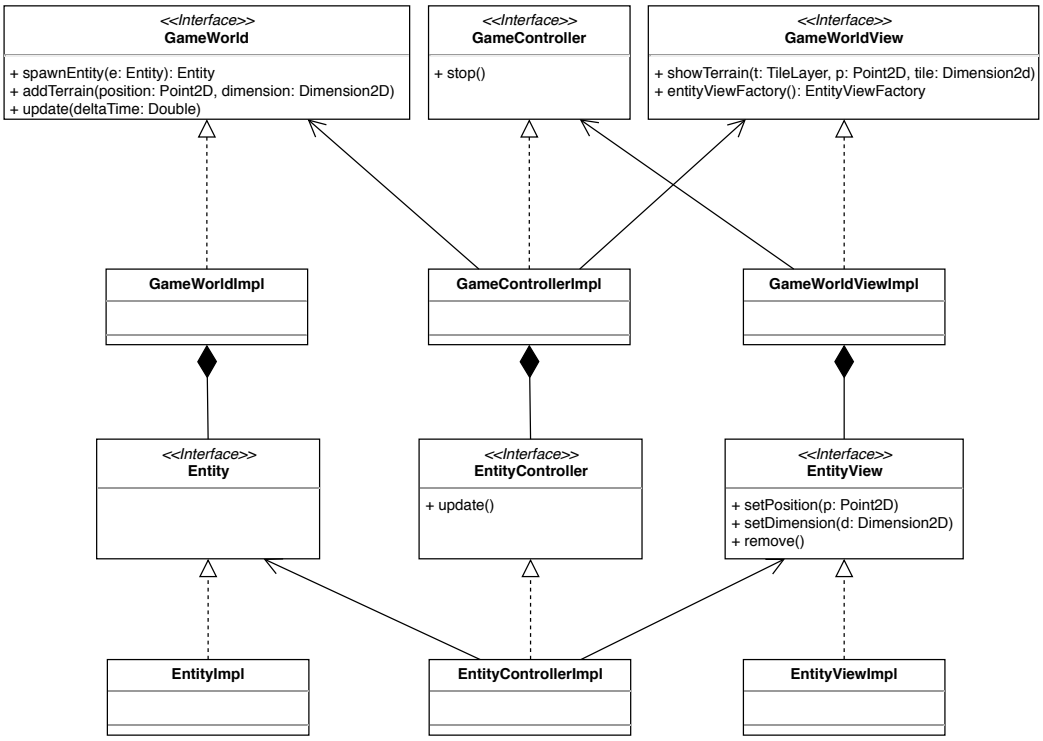

Figura 2.1: Schema dell'architettura MVC generale. Per approfondimenti su EntityViewFactory si rimanda al paragrafo di Alessandro Oliva.

Il software è basato su architettura MVC come da Figura 2.1. Il controller principale del gioco (GameController) inizializza e coordina il model (Ga-

meWorld) ed aggiorna la view di conseguenza (GameWorldView). La gestione delle Entity viene delegata a singoli EntityController specializzati che aggiornano la relativa EntityView in base allo stato del model.

L'architettura così definita non fa assunzioni sulla tecnologia impiegata dalla view e ne permette la modifica o sostituzione senza bisogno di scomodare né controller né tanto meno model.

2.2 Design dettagliato

Federico Pettinari

Mi sono occupato principalmente del model lato Entity ed EntityComponents, della fisica e della gestione del giocatore. A seguire un'illustrazione delle parti di maggior rilievo da me curate.

Per disaccoppiare il più possibile gli EntityController dagli EntityComponents ho adottato un'architettura publish-subscribe ad eventi come da Figura 2.2. I controller si registrano agli eventi delle Entity senza bisogno di sapere da chi effettivamente tali eventi saranno generati. Le Entity non hanno bisogno di sapere chi effettivamente riceverà l'evento. Gli EntityComponents utilizzano questo meccanismo per pubblicare cambiamenti di stato. Questo sistema, seppur non ancora perfetto, ha aumentato la flessibilità del software nonchè alleggerito il codice lato EntityController.

Ho inoltre cercato di massimizare il riutilizzo del codice racchiundendo in classi astratte le parti in comune degli EntityComponents e ricorrendo talvolta al pattern Template Method come in Figura 2.3.

Dato che in fase di sviluppo si è deciso che alcuni componenti (fra i quali Brain e Movement) si sarebbero dovuti disabilitare dopo la morte, ho aggiunto un semplice metodo "detachesOnDeath" ad AbstractEntityComponent che ritorna un booleano. Le classi che lo estengono possono liberamente fare override e decidere il comportamento opportuno.

Con la stessa logica ho realizzato "handleContact" di AbstractContactAwareComponent che fornisce un entry-point minimale per la gestione delle collisioni.

AbstractMovement racchiude la logica base del movimento ed è configurabile facendo override di "computeMovement" e "applyMovement" per poter personalizzare, a scelta, il modo in cui si calcola il movimento da effettuare o il modo in cui questo valore viene effettivamente applicato al corpo.

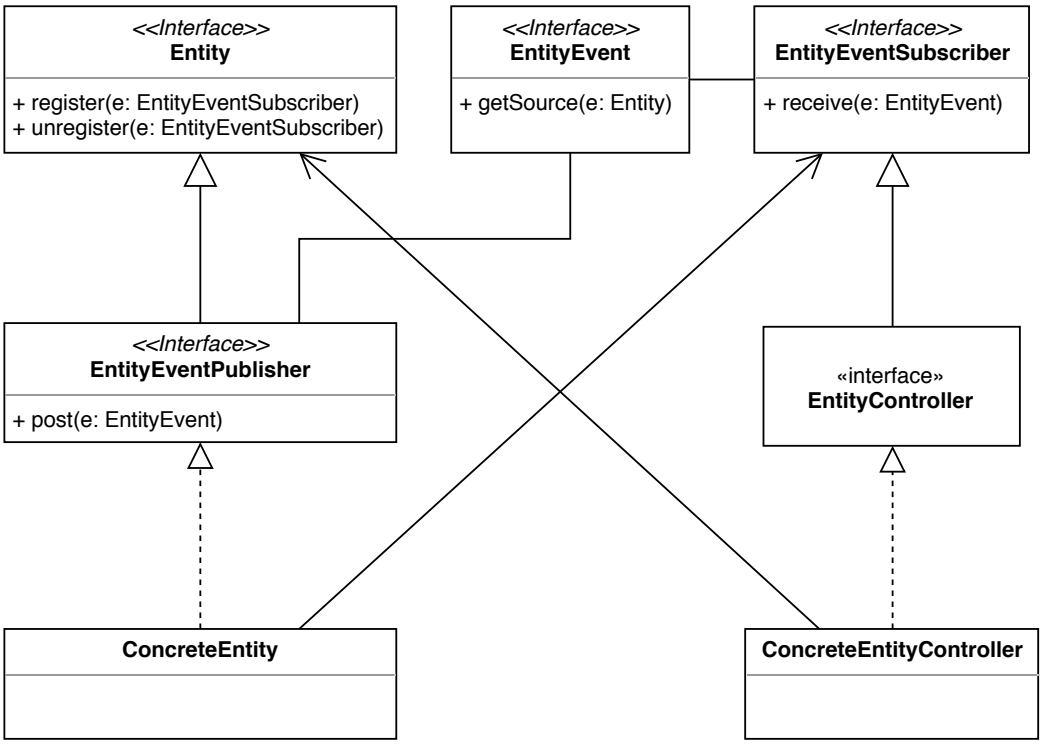

Figura 2.2: Schema dell'architettura Publish-Subscribe fra Entity ed EntityController

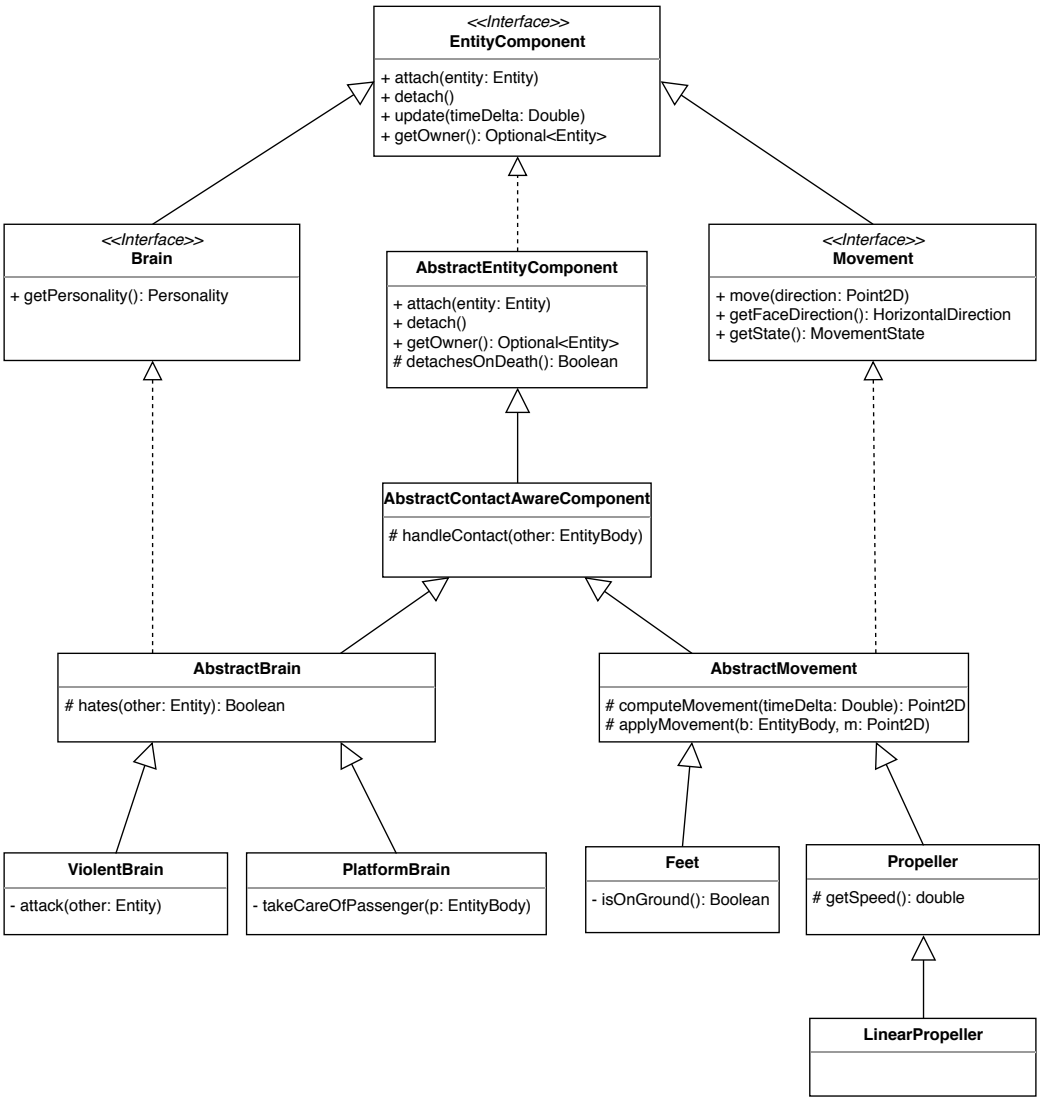

Figura 2.3: Illustrazione di scheletro e gerarchia dei componenti ed una sua parziale estensione

Per quanto riguarda la realizzazione della fisica mi sono avvalso dei pattern Strategy, Builder e Facade (Figura 2.4) per disaccoppiare il nostro software dal motore fisico adottato (al momento Box2D).

Tutta la gestione dei corpi è nascosta dietro le interfacce PhysicsEngine, BodyBuilder ed EnityBody. L'implementazione del BodyBuilder fa uso della Strategy B2DEntitySpawner offerta da B2DPhysicsEngine per creare un Body (di Box2D) inserendolo nel World ed abilitandone la simulazione. Infine B2DBodyFacade è un wrapper per il Body che funge anche da Facade celandone la gestione delle API ed esponendo il minimo indispensabile alla realizzazione del nostro modello. Questo layer d'astrazione si è reso necessario anche perchè le API di Box2D espongono campi pubblici e metodi che possono essere chiamati solo in determinati momenti e che potrebbero sortire effetti indesiderati o imprevedibili se combinati in modo non corretto.

Siccome tutte le entità in grado di attaccare avrebbero usato una MeleeWeapon, e la differenza principale sarebbe stata il verso dell'attacco, ho congegnato tale classe perchè fosse configurabile tramite Strategy. Essa prende al costruttore una funzione (Predicate) per decidere se sia possibile attaccare verso un determinato lato dell'Entity. In questo modo è stato facile modellare l'attacco del Player (che può potesse essere diretto solo sotto se stesso) e quello di altre Entity che possono attaccare in direzioni multiple.

La realizzazione del personaggio guidato dal giocatore è consistita nel semplice mix degli opportuni EntityComponents ed uno speciale EntityController. Tale controller riceve input tramite pattern Observer che traduce in comandi per il componente Movement dell'entità del giocatore. Inoltre aggiorna la view con informazioni aggiuntive quali il livello di vita, quantità di soldi e chiavi raccolte. Tali informazioni (ricevute tramite eventi) sono poi mostrate a schermo in sovrimpressione tramite un'altra view interna (HudView). Riguardo alla parte Controller e View ho provveduto ad estendere il sistema realizzato da Andrea Betti e ad implementare la HUD. Questo sistema è illustrato in Figura 2.5.

Infine ho realizzato la Static Factory InfiniteSequence che è servita a supporto dei pattern di movimento fissi di alcune entità.

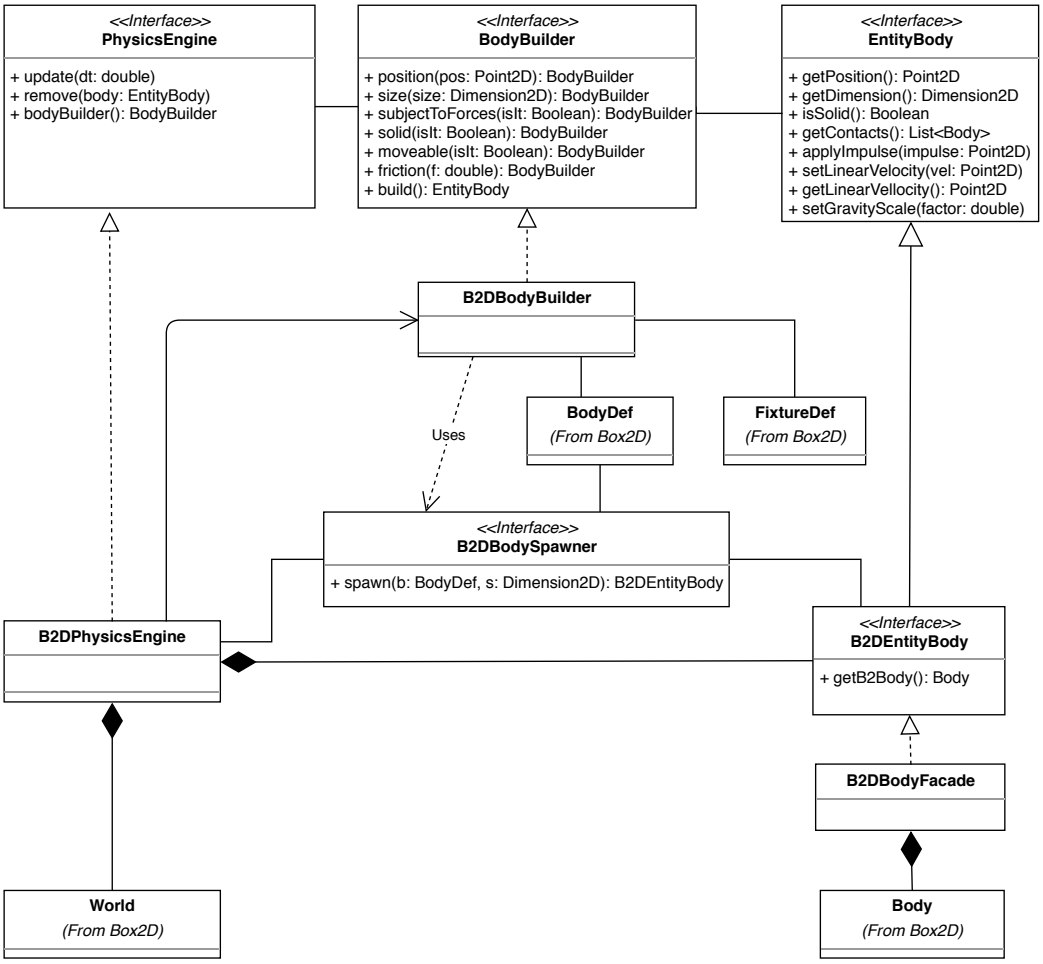

Figura 2.4: Gestione della fisica ed incapsulamento di Box2D

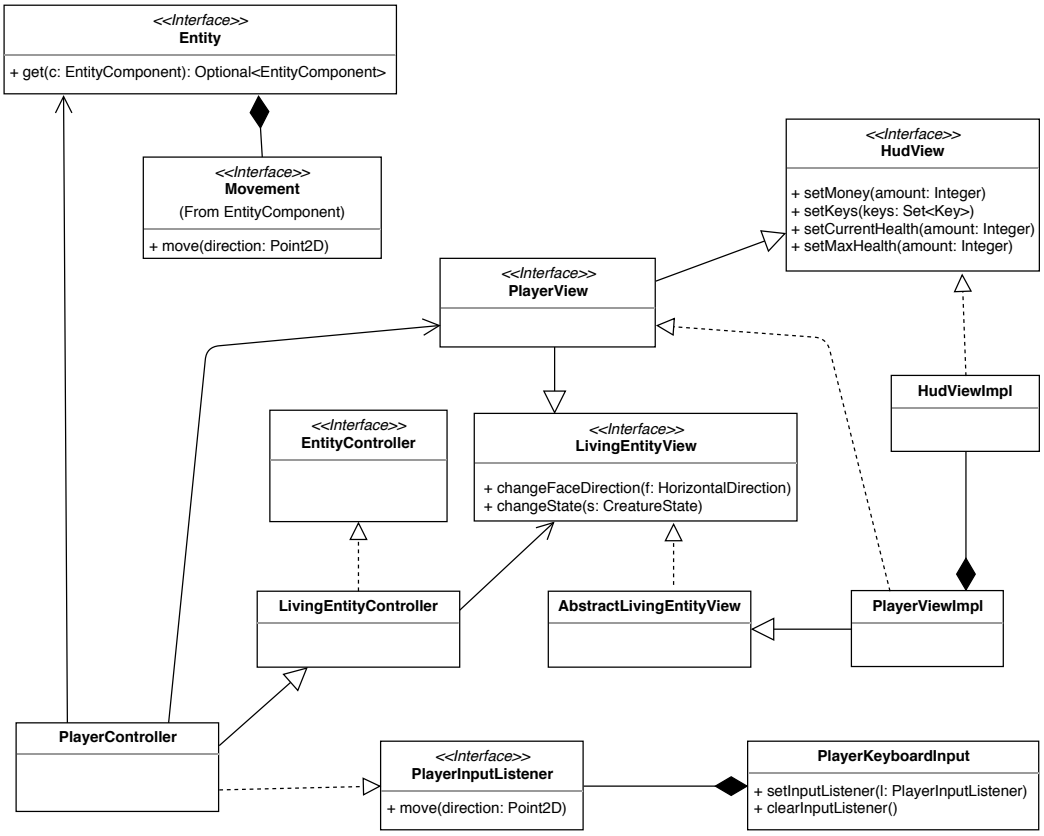

Figura 2.5: Controllo del giocatore ed aggiornamento della view tramite estensione di LivingEntityController ed AbstractLivingEntityView realizzati da Andrea Betti

Alessandro Oliva

In questa sezione tratterò di come viene caricato un livello assemblandone i componenti, i menù, il comparto audio, lingue e la gestione dei progressi di gioco e opzioni.

È prima necessaria una breve premessa. Il "cuore" dell'applicazione è composto dal MainController, che coordina il funzionamento dell'applicazione attraverso il pattern Observer, gestendo tutti i controller specifici e il MainView. A sua volta, col medesimo pattern, il MainView gestisce tutte le view specifiche. Tutte le schermate hanno un'architettura MVC con interfacce. Alla richiesta di un azione da parte di un sotto-controller o all'intercettazione di un evento, il MainController gestisce la richiesta.

Durante la partita, alla morte del Player o al completamento di un livello viene cambiata la schermata corrente della view.

Di seguito descriverò il funzionamento delle classi principali degli elementi citati precedentemente.

Un livello è composto da mappa ed entità. La mappa contiene informazioni sulla posizione e la natura delle entità, su ciò che è solido e la veste grafica, il caricamento del livello avviene tramite la lettura di queste informazioni attraverso LoadLevel. Viene prima caricata nel GameWorldView la parte grafica, successivamente vengono create le zone tangibili, infine caricate le entità. Per caricare le entità ne viene letta la posizione, poi viene letta la natura dell'entità, a ciascun type corrisponde un'entità diversa. Col pattern Factory viene generata l'entità richiesta e collocata nelle coordinate corrispondenti.

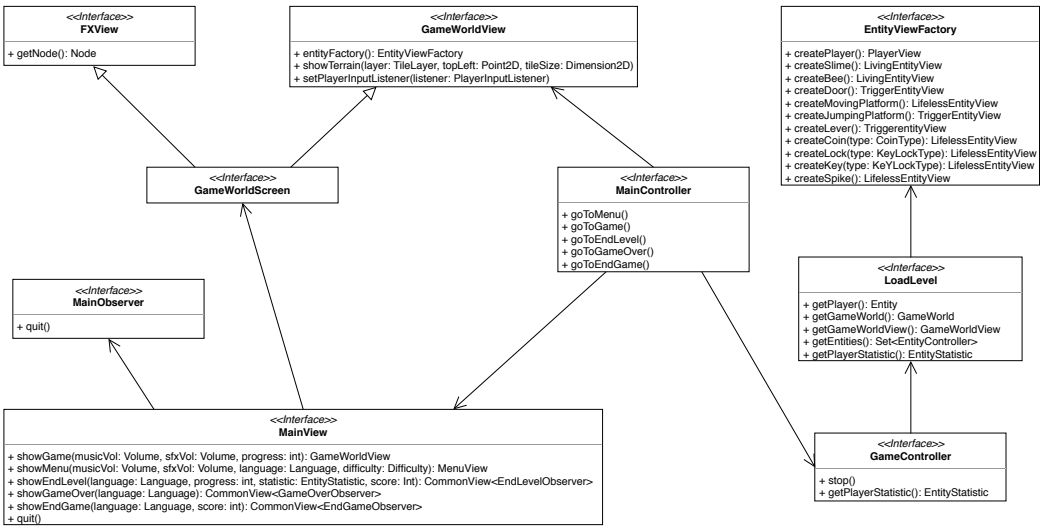

Figura 2.6: Caricamento del livello e aggiornamento della view

I diversi menù hanno alla base la medesima architettura, un controller gestisce le richieste mentre una view permette all'utente di interagire. Attraverso il pattern Observer le richieste vengono gestite dal controller locale o passate al MainController. A livello completato vengono mostrate le statistiche del giocatore e punteggio derivato, a fine gioco il punteggio totale.

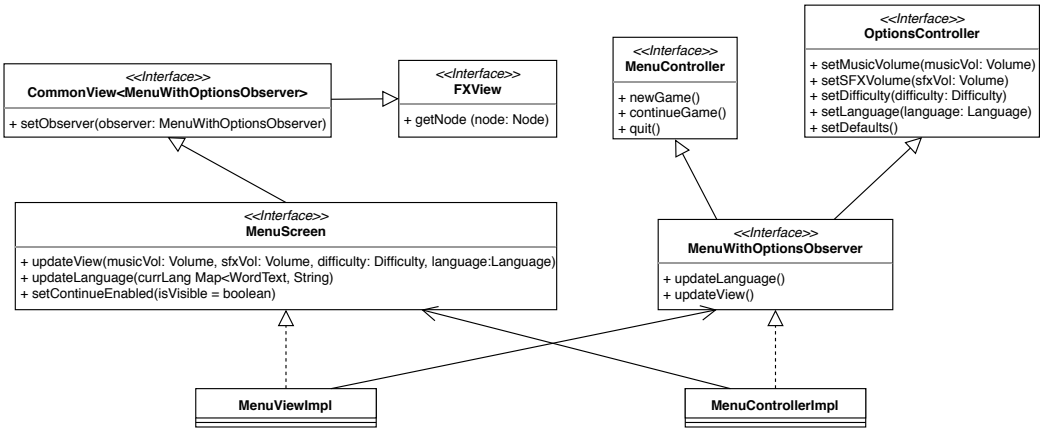

Figura 2.7: Menù con architettura MVC che utilizza il pattern Observer. Gli altri menu sono implementati allo stesso modo

Il comparto audio è gestito dalla classe AudioPlayer, che riproduce musica e effetti sonori. Attraverso il controller viene ordinata dalle altre classi la riproduzione di un dato file audio.

Le lingue sono gestite dalla classe LoadLanguage, in base alla lingua scelta dalle opzioni tutto il testo presente nel gioco viene tradotto di conseguenza.

La gestione dei progressi di gioco e opzioni avviene tramite MainController, che salva e carica usando la classe SaveLoadManager, che registra progressi e opzioni attuali. I salvataggi di gioco e opzioni sono modellati come oggetti con dei campi.

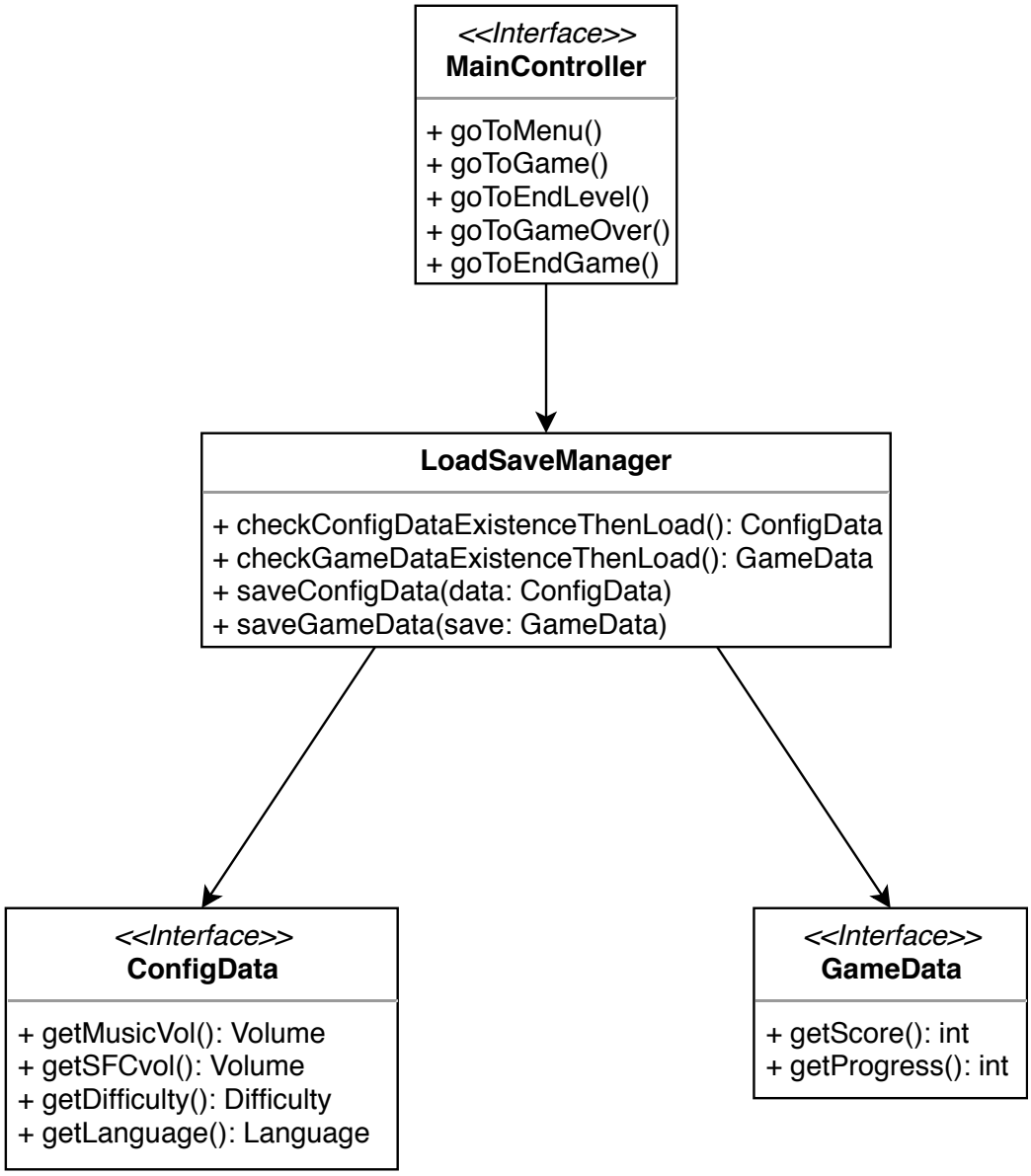

Figura 2.8: Lettura e scrittura dei salvataggi

Andrea Betti

In quanto responsabile della creazione e del comportamento di qualsiasi entità del gioco, all'infuori del Player, in questa sezione descriverò il funzionamento delle principali classi che determinano il loro specifico comportamento all'interno del mondo di gioco, denominate componenti.

Attraverso l'utilizzo del pattern Observer, i componenti vengono associati alla classe implementante Entity interessata, la quale mantiene i riferimenti a essi e si occupa di fornire metodi atti ad aggiornare il loro stato. A seguito del verificarsi di determinate condizioni, i componenti inviano un messaggio d'evento che viene poi ascoltato e gestito da classi esterne predisposte.

A ogni categoria di entità con proprietà in comune sono state associate classi di View e Controller dedicate per raggruppare i metodi di ricezione degli eventi necessari (la loro implementazione è rappresentata in Figura 2.9 e in Figura 2.10).

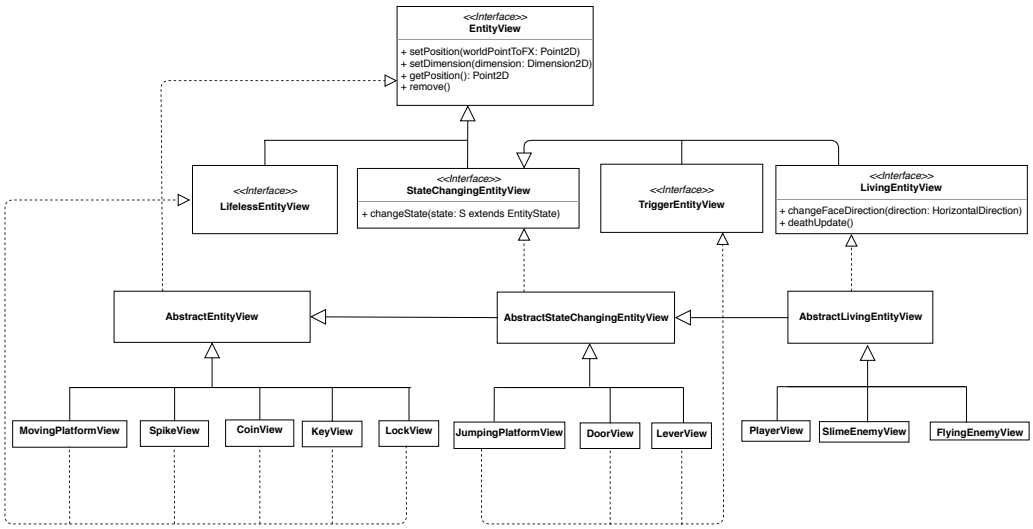

Figura 2.9: Schema UML rappresentante la suddivisione delle EntityView applicata

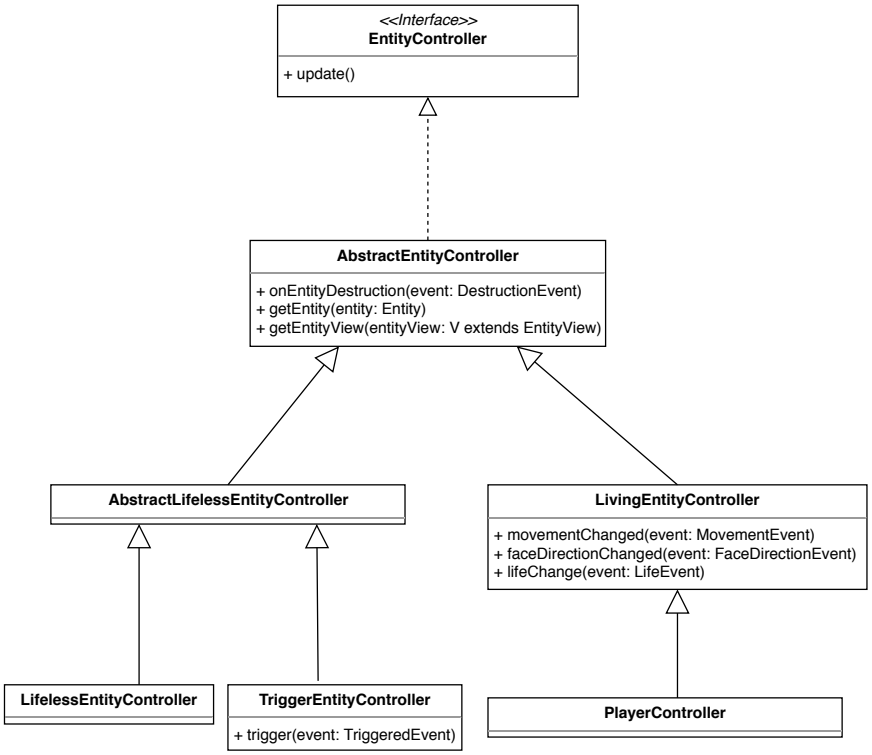

Figura 2.10: Schema UML rappresentante la suddivisione degli EntityController applicati

Di seguito, tratterrò dei componenti da me sviluppati di cui ritengo sia interessante studiare la loro struttura e/o l'utilizzo di design pattern.

I Brain sono dei componenti con lo scopo di gestire il comportamento delle entità a cui sono collegati rispetto alle entità con cui entrano in contatto nel mondo di gioco.

Il PickupableBrain consente all'entità a cui è associata di rimuovere qualsiasi suo riferimento dal Controller nel caso venga rivelato un contatto con un'entità a cui è associato un componente Inventory, per poi aggiornare le informazioni contenute in quest'ultimo. Attaverso il pattern Strategy, è possibile determinare quali cambiamenti subirà l'inventory, e quindi estendere l'utilizzo del componente PickupableBrain a entità collezionabili con funzioni diverse. Il suo utilizzo è esemplificato in Figura 2.11.

Il pattern Strategy è utilizzato anche dal componente FixedPatternPilot, che prende in input un Supplier di punti bidimensionali: essi determinano il percorso che il corpo dell'entità seguirà attraverso l'utilizzo dell'eventuale componente Movement. Il suo utilizzo è esemplificato in Figura 2.12.

I Trigger e i Triggerable hanno entrambi la funzione di inviare un messaggio di evento di attivazione, con la differenza che i secondi possono essere predisposti a invarlo a seguito di un messaggio di attivazione dei primi.

Nella pratica, l'uso di classi estendenti Trigger e Triggerable è stato abbinato rispettivamente alle entità Lever e Door: in questi casi specifici l'attivazione avviene tramite la rilevazione di contatti con altre entità, ma attraverso opportune estensioni è possibile gestire il loro comportamento in diverso modo.

Per fare in modo che dei componenti Trigger e Triggerable comunichino, vengono associate loro delle password per collegare lo scatenamento degli eventi inviati dal Trigger all'attivazione dei Triggerable a cui è stata associata la stessa password dell'evento. In tal caso, gli eventi inviati dai Trigger vengono ascoltati da un classe TriggerLinker, che si occuperà di attivare i componenti Triggerable collegati; a loro volta, anche questi invieranno un evento di attivazione. La struttura è esemplificata in Figura 2.13.

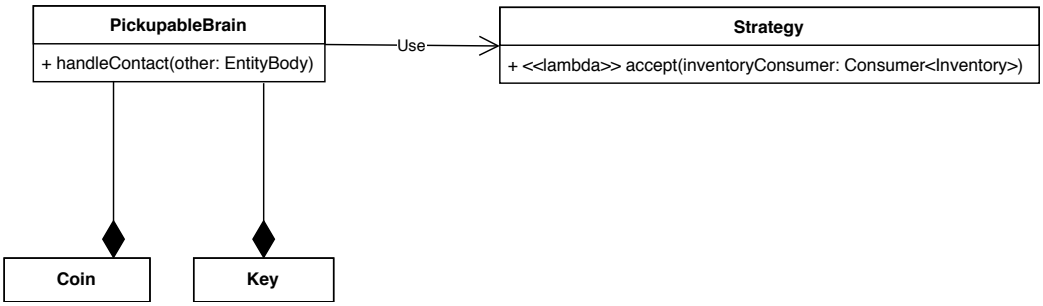

Figura 2.11: Rappresentazione UML del pattern Strategy per PickupableBrain

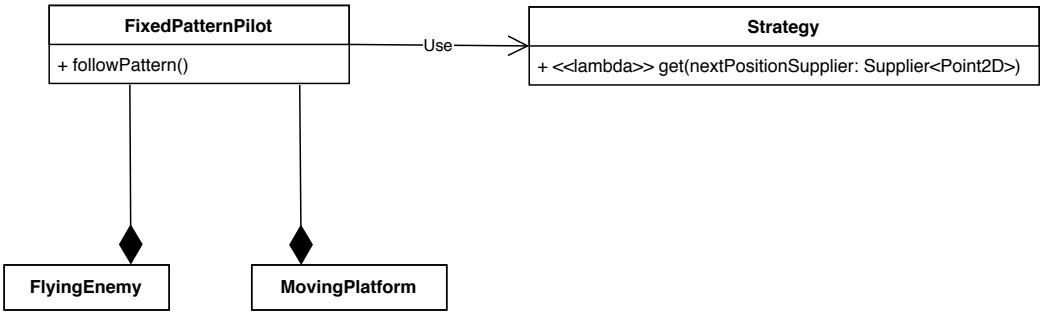

Figura 2.12: Rappresentazione UML del pattern Strategy per FixedPatternPilot

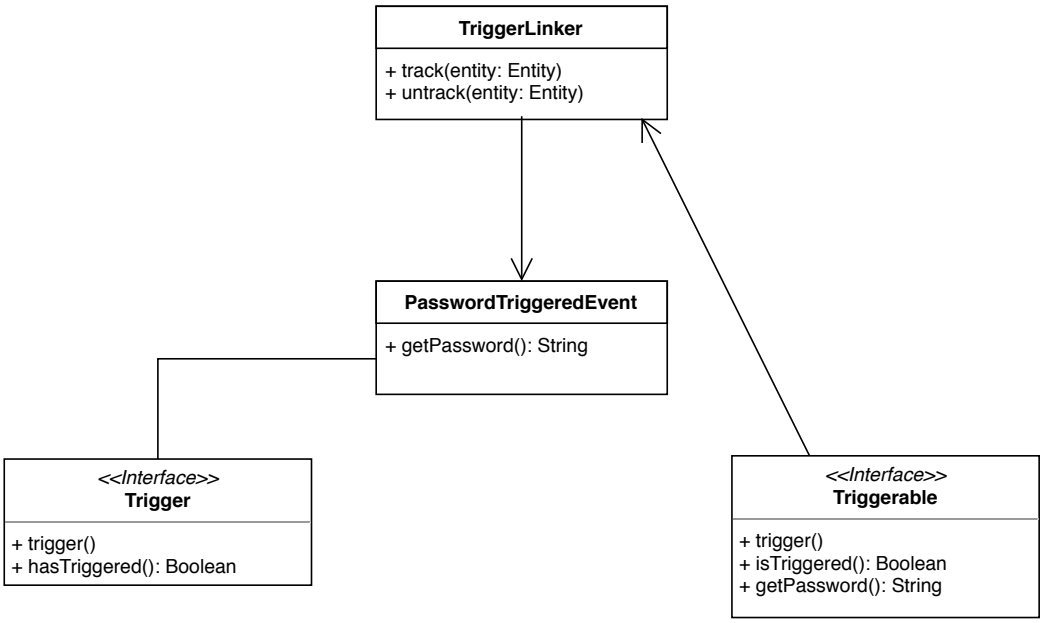

Figura 2.13: Rappresentazione UML del modello di comunicazione tra Trigger e Triggerable

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Abbiamo realizzato molteplici test automatici basati sulla suite di JUnit. Tutti i test si trovano nei package it.unibo.oop17.ga_game.tests e it.unibo.oop 17.ga_game.tests.entities e sono stati nominati in base alla classe che testano.

InterfacesBagTest è stato creato soprattutto per gli edge-cases riguardanti il meccanismo di reflection. Questa classe è vitale per il sistema degli EntityComponents.

Dentro it.unibo.oop17.ga_game.tests.entities si trovano alcuni test che verificano la correttezza e consistenza delle interfacce e degli eventi prodotti da Entity ed EntityComponent.

Il MovingPlatformTest si concentra sul corretto trasporto dei passeggeri della piattaforma ed è stato creato a fronte di un bug scovato durante lo sviluppo.

3.2 Metodologia di lavoro

La parte di analisi è stata eseguita collettivamente e di persona durante alcune giornate precedenti la fase di sviluppo.

La suddivisione finale del lavoro è risultata come segue:

- Alessandro Oliva: View e Controller principali e del menu. Effetti sonori per le entità, musiche di sottofondo, caricamento della mappa e

salvataggio del progresso di gioco. Ha deciso di aggiungere il supporto multi-lingua e la configurazione di difficoltà e volumi.

- Andrea Betti: IA entità nemiche e meccaniche relative ad oggetti collezionabili ed interattivi (monete, chiavi, leve, porte) inclusi i relativi EntityComponents. Ha sviluppato View e Controller delle entità.

- Federico Pettinari: Simulazione fisica dei corpi, scheletro di Entity ed EntityComponent, gestione del giocatore (inclusi alcuni dei suoi componenti) e controllo tramite tastiera, HUD.

Siamo partiti fin da subito con l'idea di base di dividere le Entity in più componenti sostituibili; tuttavia in fase di sviluppo abbiamo scoperto la necessità di ulteriori componenti e questo ci ha costretto a tornare in fase di progettazione. Nonostante ciò, lo scheletro del modello non è mai cambiato e si è rivelato più che sufficiente per i nostri scopi.

Ogni membro ha proceduto a lavorare alla sua parte il più possibile in autonomia.

Abbiamo sfruttato il DVCS GIT cercando ognuno di lavorare individualmente su un feature-branch relativo al compito da svolgere. Eventualmente è stato effettuato il merge su un branch di sviluppo generale chiamato develop. Un terzo branch principale, il master, contiene solo versioni principali funzionanti e risultanti da merge di develop. E' comunque capitato occasionalmente che piccoli commit fossero eseguiti direttamente su develop ma ciò non ha portato a problemi di alcun tipo. I vari merge sono stati per la maggior parte senza conflitti.

3.3 Note di sviluppo

Federico Pettinari

Ove possibile ho sempre fatto uso delle feature avanzate di Java quali:

- Stream al posto dei classici costrutti imperativi per parti algoritmiche.

- Optional al posto di null per esprimere campi e parametri opzionali in modo esplicito.

- Interfacce offerte da java.util.function per piccole Strategy

- Lambda e method references per ottenere un codice più compatto e leggibile.

Ho utilizzato la libreria "Guava"[1] in particolar modo per l'implementazione del sistema precedentemente descritto degli EntityEvents tramite "EventBus"[2]. Come già menzionato ho utilizzato "JBox2D"[3] per la simulazione fisica degli EntityBody in quanto è attualmente il più famoso motore fisico in 2D.

Ho utilizzato generici e reflection per sviluppare la classe InterfacesBag, la quale modella una particolare mappa da interfaccia ad implementazione. Specificata un'interfaccia padre, la suddetta classe offre metodi per aggiungere e rimuovere mappings e trovare un'implementazione partendo dall'interfaccia senza bisogno di usare cast ed instanceof. Tale classe è stata la base per l'implementazione di Entity ed ha consentito l'aggiunta e rimozione di componenti di diverso tipo in modo estremamente flessibile.

La classe SpriteAnimation è stata presa da https://netopyr.com/2012/03/09/creating-a-sprite-animation-with-javafx/ con minimi riadattamenti.

Alessandro Oliva

Ho fatto uso delle seguenti feature avanzate:

- Generici per l'interfaccia CommonView, in quanto il tipo dell'observer cambia a seconda della View che lo implementa.
- Optional per l'activeGameController nel MainController poiché questo potrebbe non essere presente alla chiamata del metodo stop.
- Lambda e method references per ottenere un codice più compatto e leggibile.
- Stream per il caricamento delle lingue.

Per il caricamento della mappa ho usato la libreria "libtiled"[4], con cui si può accedere alle tile (immagini dei blocchi) e a posizione e tipo di una entità per il caricamento. Per la gestione degli eventi ho usato Google Guava.

Andrea Betti

Ho utilizzato i generici bounded nelle classi implementanti EntityView ed EntityController. Nelle prime, vi è l'interfaccia covariante StateChangingEntityView in S, dove S estende EntityState, in modo da permettere alle

[1] Google Guava: `https://github.com/google/guava`

[2] EventBus: `https://github.com/google/guava/wiki/EventBusExplained`

[3] JBox2D: `http://www.jbox2d.org/`

[4] libtiled: `https://github.com/bjorn/tiled/tree/master/src/libtiled`

classi implementanti StateChangingEntityView di utilizzare i metodi dichiarati che utilizzano parametri di tipo S attraverso una implementazione specifica di EntityState. Per quanto riguarda gli EntityController, sono state usate classi e interfacce covarianti in tipi implementanti interfacce di EntityView più o meno specifiche, al fine di gerarchizzare i metodi comuni ed evitare ridondanze.

Ho utilizzato espressioni lambda nelle situazioni dove ho trovato conveniente passare interfacce funzionali come parametri di metodo, per rendere il codice più elastico possibile. Esempi rilevanti sono:

- l'utilizzo di un Supplier nella dichiarazione di parametri della classe FixedPatternPilot.
- l'utilizzo di Consumer da parte della classe PickupableBrain, nel metodo ifPresent di oggetti di tipo Optional in diverse classi (in SlimeEnemyBrain, TriggerLinker, LockBrain, ViolentBrain) e nel metodo forEach invocato dalla mappa istanziata in TriggerLinker.
- l'utilizzo di Runnable nelle EntityView per avviare le animazioni degli sprite di gioco.

Ho inoltre esteso l'utilizzo della reflection da parte delle classi Entity sviluppato da Pettinari per individuare e utilizzare gli eventuali componenti associati attraverso l'interfaccia che implementano.

Per quanto riguarda le librerie esterne, ho fatto uso della classe EventBus della libreria Google Guava per gestire l'invio e l'ascolto di eventi tra classi senza che siano collegate esplicitamente tra di loro.

Capitolo 4

Commenti finali

Il progetto trae ispirazione da molteplici fonti quali:

- Le lezioni sul game-programming del Prof. Alessandro Ricci.
- Numerose letture sugli Entity-Component systems (anche se il nostro sistema è differente in quanto i nostri components hanno un comportamento).
- Il progetto consigliato "delirium".
- L'esercizio "robot" dalla lezione 4 di laboratorio.

4.1 Autovalutazione e lavori futuri

Federico Pettinari

Sono molto soddisfatto del risultato ottenuto anche se sicuramente imperfetto: alcune parti come HUD e PlayerView dovrebbero essere ulteriormente scomposte e rifattorizzate. Probabilmente l'interfaccia Entity espone troppe cose. Sarebbe stato interessante considerare la possibilità di nascondere completamente la struttura a componenti ed esporre solo le caratteristiche obbligatorie come la posizione e dimensione; lasciando agli eventi il compito di comunicare cambiamenti di stato.

Un ulteriore elaborazione avrebbe però fatto sconfinare il monte ore per cui ho deciso di tralasciare.

Il sistema così congegnato si è comunque rivelato alquanto flessibile tantochè è stato molto facile aggiungere ulteriori entità come la MovingPlatform e JumpingPlatform. Sicuramente gran parte del codice potrebbe essere riutilizzata per lo sviluppo di altri videogiochi.

Sicuramente una limitazione consiste nel'assunzione che tutte le entity abbiano forma rettangolare. Ritengo però che per la maggior parte dei giochi 2D sia un'approssimazione più che accettabile.

Purtroppo in fase di analisi e sviluppo ho implicitamente ed involontariamente assunto la figura (non ufficiale) di leader e talune volte anche code-reviewer. Questa situazione mi ha addossato ulteriori responsabilità che avrei preferito evitare in favore di una situazione più paritaria. Immagino che ciò sia successo anche a causa di un certo dislivello di esperienza fra i membri del gruppo.

Alessandro Oliva

Sono soddisfatto del risultato finale, nonostante certi elementi potessero essere sviluppati meglio se preceduti da un'adeguata progettazione.

Riconosco la poco omogenea qualità del codice, che in certi punti passa da un'implementazione studiata accuratamente a sezioni scritte con pattern ingenui, senza esser riuscito a ritagliare del tempo per riscrivere le parti di codice interessate.

Ho cercato di rendere la mia parte flessibile per quanto riguarda la scomposizione e sostituzione di classi, e avrei interesse a estendere il progetto in futuro implementando alcune idee scartate in corso d'opera per il poco tempo a disposizione.

Un grosso scoglio da superare è stata la realizzazione di un JAR portabile, operazione assai costosa in termini di tempo, che però mi ha dato occasione di approfondire lo studio del problema.

Il gioco soffre di memory leak, di conseguenza durante il gameplay subisce un lag crescente di volta in volta che viene caricata una nuova mappa; per motivi di tempo non sono riuscito ad indagare a sufficienza sull'origine del problema e a risolverlo.

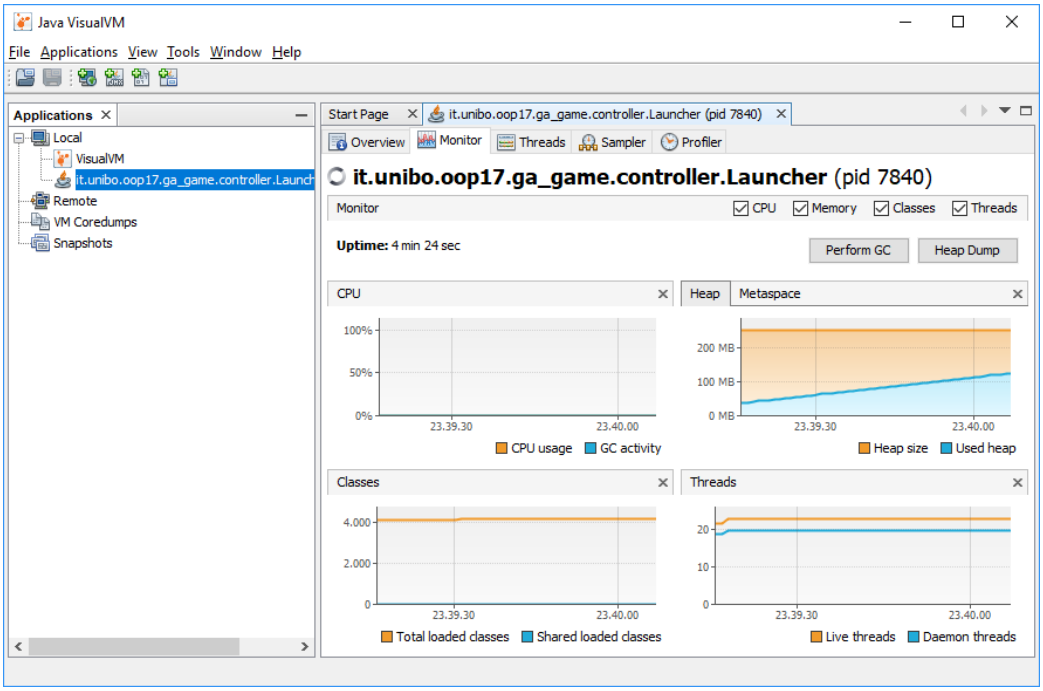

Figura 4.1: Memoria occupata all'avvio

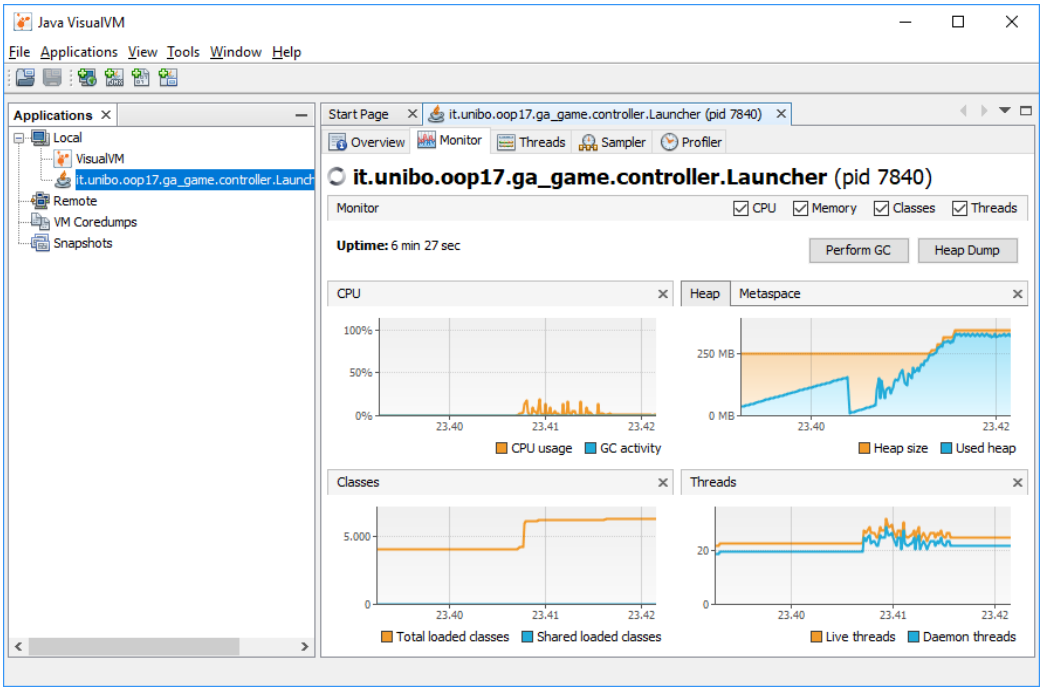

Figura 4.2: Memoria occupata dopo aver ricaricato un livello 10 volte senza chiudere l'applicazione

Andrea Betti

Mi ritengo abbastanza soddisfatto del lavoro svolto, nonostante mi piacerebbe effettuare delle aggiunte di contenuto per rendere il gioco più completo in futuro (più tipologie di nemici, più livelli, migliore estetica).

Per quanto riguarda la struttura della parte del software da me prodotta, ritengo sia abbastanza flessibile e facilmente estendibile. Quest'esperienza mi ha aiutato a migliorare le mie capacità di programmazione ad oggetti e a imparare proprietà che non conoscevo del linguaggio Java.

Oggettivamente, il lavoro da me svolto è stato un po' minore rispetto a quello di Oliva e Pettinari: in quanto la mia parte, riguardante le entità con cui il Player entra in contatto, è stata dipendente da quella di Pettinari, il mio lavoro è prevalentemente consistito nell'estendere l'architettura che lui ha sviluppato per il Player alle altre entità. Ho comunque fatto del mio meglio per dare una mano agli altri componenti del team quando ve ne fosse bisogno.

4.2 Difficoltà incontrate e commenti per i docenti

Vorremmo cogliere l'occasione per suggerire ai professori di inserire esercizi di laboratorio più orientati alla progettazione e che partano da un progetto vuoto.

Le difficoltà maggiori sono state proprio la progettazione e una suddivisione dei compiti equa e realistica.

Appendice A

Guida utente

All'avvio dell'applicazione viene visualizzata la schermata di menu principale (Figura A.1).

Figura A.1: Schermata di menu principale

Il pulsante "Continua" viene visualizzato solamente se si ha completato almeno il primo livello del gioco, e consente di riprendere il gioco all'ultimo livello non completato.

"Nuova partita" consente di cominciare il gioco dal primo livello ed resetta l'eventuale salvataggio precedente.

"Opzioni" consente di regolare il volume della musica e degli effetti sonori, di cambiare la lingua di gioco (italiano, inglese, tedesco, spagnolo, francese) e la difficoltà di gioco (Figura A.2).

"Esci" permette di terminare l'applicazione, analogamente al tasto ESC usabile in qualsiasi schermata del gioco ci si trovi.

Figura A.2: Schermata di opzioni

In Figura A.3 è rappresentato un momento di gioco. Il giocatore può muovere l'alieno utilizzando i tasti A e D o le frecce direzionali "sinistra" e "destra" per muoversi orizzontalmente e W o la freccia "su" per saltare.

Per danneggiare la vita dei nemici, l'alieno deve saltare esattamente sopra di loro; altri tipi di contatti danneggeranno la vita del giocatore.

Per interagire con chiavi, monete, lucchetti, leve e porte, è sufficiente che l'alieno collida con essi senza la necessità che il giocatore prema pulsanti specifici.

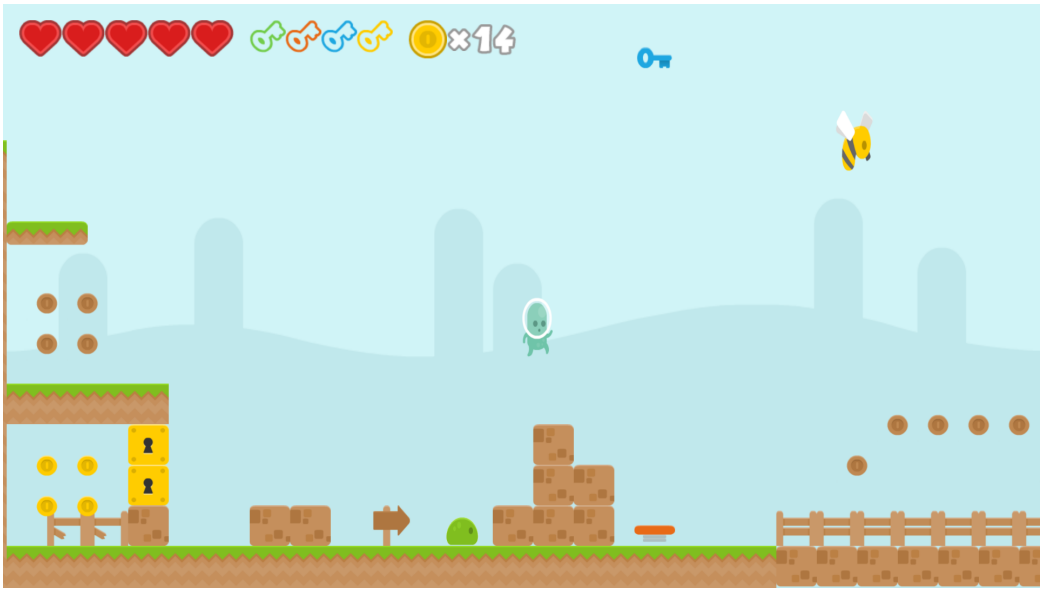

Figura A.3: Schermata di gioco