

Relazione per
“Ciccio Pier - THE GAME”

Alessandro Aldini Davide Valdifiori
Riccardo Mingozzi MD Shokot Alam

25 aprile 2022

Indice

1	Analisi	2
1.1	Requisiti	2
1.2	Analisi e modello del dominio	3
2	Design	5
2.1	Architettura	5
2.2	Design dettagliato	6
3	Sviluppo	23
3.1	Testing automatizzato	23
3.2	Metodologia di lavoro	24
3.3	Note di sviluppo	26
4	Commenti finali	28
4.1	Autovalutazione e lavori futuri	28
A	Guida utente	30

Capitolo 1

Analisi

Il software realizzato per il corso di Programmazione ad Oggetti è Ciccio Pier - THE GAME, un videogioco ispirato a Super Mario Bros. L'obbiettivo del gioco è superare i livelli, con difficoltà progressiva, cercando di ottenere il punteggio più alto possibile.

La difficoltà del gioco consiste nel superare vari ostacoli, tra cui nemici, trappole, platforming e gestire il sistema della stamina.

1.1 Requisiti

Requisiti funzionali

- Inserire il proprio username per salvare il punteggio ottenuto a fine livello e visualizzare la classifica con i punteggi migliori
- Vari livelli con ambientazioni differenti e difficoltà crescente
- Possibilità di cambiare le impostazioni di gioco ad esempio suoni, risoluzione, etc...
- Tutorial all'interno del gioco
- Diversi tipi di nemici con comportamenti ed attacchi differenti
- Il giocatore può attaccare i nemici con l'azione del morso
- Il giocatore perde vita quando viene attaccato dai nemici o mangiando cibi grassi, può recuperarla uccidendo i nemici
- Il giocatore perde stamina saltando, può recuperarla mangiando nemici o cibi grassi. Se la stamina scende sotto un certo livello il giocatore sarà svantaggiato nel movimento

- I boosts conferiscono vantaggi al giocatore
- Il punteggio ottenuto a fine livello è dato da quanti nemici sono stati uccisi e dal numero di oggetti raccolti, che sono boosts, cibi e monete

Requisiti non funzionali

- Il gioco dovrà avere una grafica intuitiva e accattivante
- Il gioco dovrà avere musiche ed effetti sonori per migliorare l'esperienza di gioco

1.2 Analisi e modello del dominio

Il gioco è formato da vari livelli.

Ogni livello è composto da un inizio che è il punto dove viene creato il giocatore, una fine che è il punto da raggiungere per poter completare il livello, dei nemici e degli oggetti da raccogliere .

Il gioco dovrà essere in grado di accedere a tutti gli elementi del livello corrente ed essi dovranno poter interagire fra loro.

Trattandosi di un platform, bisognerà sviluppare un adeguato ambiente di gioco, avente una corretta fisica, nemici con pattern di movimento prestabiliti e idonea iterazione tra gli elementi a schermo.

Dovremo implementare concetti fisici di base come forza di gravità, fisicità degli oggetti e corretto funzionamento delle forze in campo, come ad esempio il salto. Un altro ostacolo potrebbe riguardare la gestione delle collisioni tra gli elementi.

Avremo necessità dunque di controllare quello che accade nel gioco ripetutivamente senza pesare troppo sulle prestazioni. Il quantitativo di elementi a schermo potrebbe infatti risultare elevato, andrà trovato un modo efficace di gestire parallelamente tutti i nemici e il giocatore (mosso dall'utente tramite input).

Il sistema di movimento dei nemici dovrà attivamente cercare il giocatore per attaccarlo e compiere movimenti che non li porti a uscire dal mondo o ad auto eliminarsi.

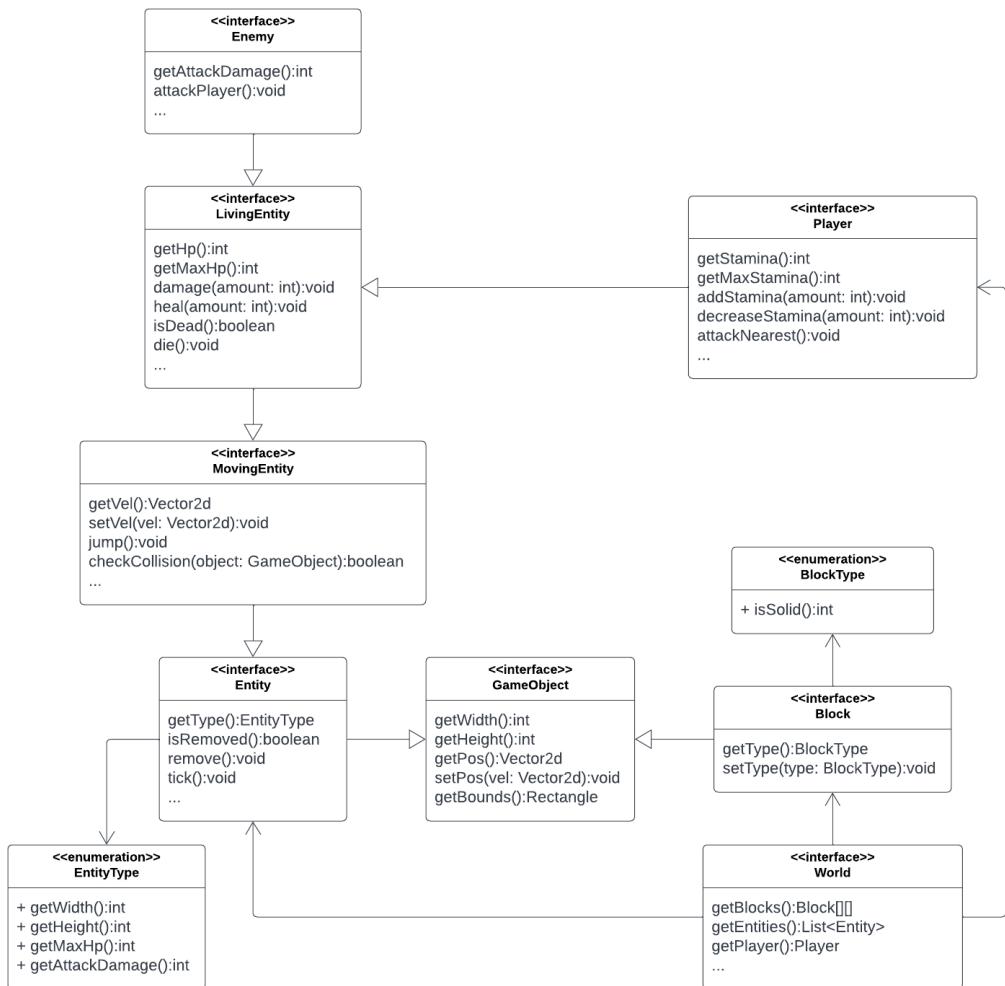


Figura 1.1: Schema UML delle entità del livello, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Per realizzare l'applicazione è stato scelto di fare uso del pattern architettonico MVC.

L'implementazione del menu di gioco sfrutta il pattern MVC ed è formato delle interface MenuController e ManagerView. Quando si avvia uno specifico livello, viene istanziato un sistema MVC dedicato.

Tutti gli oggetti del livello vengono creati nel Model. Qui sono definite le caratteristiche e il comportamento degli oggetti principali: il Player, i blocchi e le entità. Nel pattern MVC la componente Model non conosce le altre due, per questo motivo al suo interno non vi sono riferimenti ad esse. Nel caso di "Ciccio Pier", il punto di ingresso al Model è rappresentato dall'implementazione dell'interfaccia World con la classe GameWorld.

La componente Controller rappresenta il filo conduttore fra Model e View, ha il compito di manipolare le informazioni fornite dal Model e comunicarle alla View. Per gestire le iterazioni tra le componenti del gioco è stata implementata l'interfaccia classe Loop (GameLoop), che modella un vero e proprio game loop. Nel caso di "Ciccio Pier - THE GAME", il controller è rappresentato dall'implementazione dell'interfaccia Engine con la classe GameEngine.

La componente View ha il compito di rappresentare a schermo tutti gli oggetti del gioco rappresentando le componenti tramite immagini in Swing e di intercettare gli input dell'utente servendosi di un KeyListener per poi comunicarli al controller. Nel caso di "Ciccio Pier", la view è rappresentata dell'implementazione dell'interfaccia View con la classe GameView.

2.2 Design dettagliato

Alessandro Aldini

In questa sezione si analizzerà la struttura del menu principale e dei problemi che sono stati risolti durante la sua creazione, verranno inoltre citati aspetti come la gestione delle impostazioni di gioco e il funzionamento del ridimensionamento dello schermo in quanto facenti parte del contributo apportato al progetto.

Il menu principale si basa sul pattern MVC, composto come segue:

Model

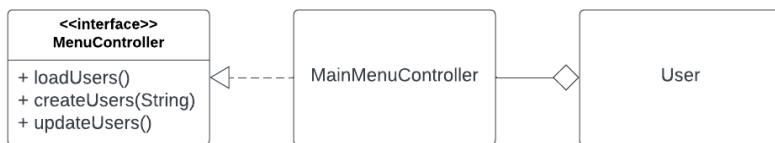


Figura 2.1: Rappresentazione UML della gestione degli utenti all'interno del MenuController .

User Rappresentato dalla omonima classe rappresenta un utente che definisce la composizione degli utenti, questo model è estremamente rilevante in quanto contiene le informazioni necessarie che verranno caricate al momento del login.

Pertanto, la prima difficolta è stata identificare un metodo per salvare e caricare correttamente i dati da un file esterno in modo che non venissero persi una volta usciti dall'applicazione.

Per risolvere questo problema ho deciso di optare per un file di tipo json in quanto ottimo per salvare e caricare oggetti tramite l'utilizzo della libreria gson, oltre ad avere già una leggera conoscenza pregressa sull'utilizzo di questo tipo di file.

inoltre, durante la progettazione si sono tenuti in considerazione una serie di possibili scenari in modo da gestirli correttamente evitando errori:

In caso il file json dove vengono salvati gli utenti non venga trovato ne viene automaticamente generato uno.

L'aggiunta di nuovi livelli viene correttamente gestita in modo che durante il caricamento degli utenti se viene a mancare un'informazione riguardante un nuovo livello aggiunto viene automaticamente inizializzato con un valore di default.

E in fine il file viene salvato ad ogni modifica significativa in modo onde evitare la perdita di dati dovuta alla erronea chiusura dell'applicazione

View

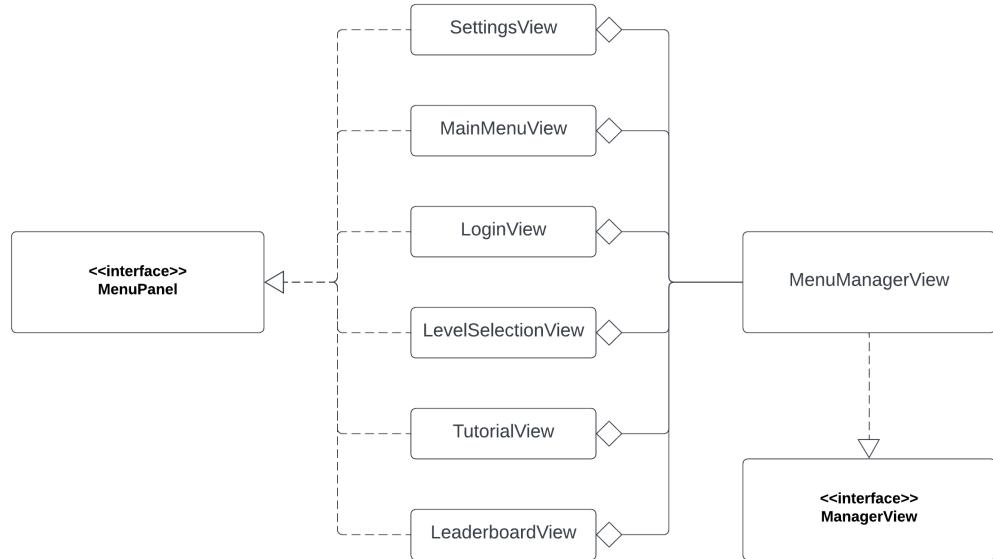


Figura 2.2: Rappresentazione UML della gestione dei pannelli della view.

ViewManager Uno dei problemi principali nella parte view era quella di mostrare correttamente i diversi pannelli, in modo da evitare inutili ripetizioni e gestire in modo facile le varie view.

per ovviare a questo problema si è optato per un manager che contenesse al suo interno i vari pannelli ed è rappresentata dalla classe MenuManagerView.

Questa è un'implementazione della interfaccia ManagerView e si occupa della gestione delle view utilizzate nel menu principale, rappresenta il JFrame di tutta l'applicazione e gestisce il suo contenuto tra le diverse view presenti nel menu principale tramite le chiamate effettuate dal controller oltre a contenere elementi condivisi tra le diverse implementazioni di MenuPanel come animazioni, bottoni e scritte.

MenuPanel Questa interfaccia identifica e definisce tutti i pannelli che la implementano dato loro semplici funzioni di base quale load e update

Interfaccia utente customizzata Una grande parte del mio contributo dal punto di vista visuale e funzionale viene dalla completa customizzazione dei vari Jcomponents con classi come CustomButton e CustomCheckBox e molte altre allo scopo di poter personalizzare aspetto in modo da essere più accattivante e di poter cambiare ogni proprietà del component in sé per adattarlo all'utilizzo necessario, questo è specialmente marcato nelle sottoclassi di CustomButton che rappresentano tutti i vari tipi ed utilizzi di bottoni all'interno dei vari menu tutti posizionati usando coordinate relative alla grandezza dello schermo in frazioni per poter assicurare una facile scalabilità.

Controller

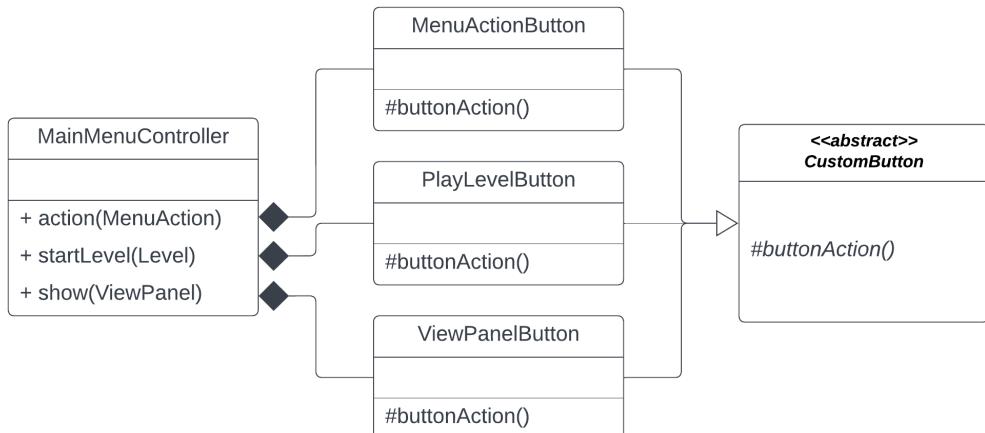


Figura 2.3: Rappresentazione UML della classe `MainMenuController`.

Il controller del menu principale Per la gestione del menu principale ho deciso di optare per un unico controller che si occupasse di gestire ogni aspetto del menu principale, dalle azioni da eseguire dopo che uno specifico bottone è stato premuto all'aggiornamento del file dove vengono salvati gli utenti.

Questo tipo di approccio è risultato molto efficace nella gestione delle impostazioni di gioco e del salvataggio degli utenti poiché direttamente relative al menu di gioco stesso e alle sue funzioni come il popolamento della leaderboard e la riproduzione di suoni e musiche.

Oltre alla gestione di altre classi come l'aggiornamento degli utenti e delle dimensioni dello schermo.

I settings e i boost

Per finire mi sono anche occupato della gestione e modifica dei settings e dell'implementazione di boost (intesi come modificatori di statistiche del giocatore come velocità) all'interno del gioco, in questa parte la criticità più grande è stata e stata fare in modo che lo schermo si adattasse in maniera corretta, controllata e automatica.

Il problema era trovare un modo per far sì che ogni componente fosse correttamente scalato e che l'utente non potesse erroneamente trovarsi in una situazione scomoda avendo una finestra più grande del proprio schermo.

Per risolvere questo problema ho optato per una semplice risoluzione fissa con rateo 16:9 e una variabile che conteneva un'oltiplicatore che viene utilizzato da tutti i componenti visualizzati a schermo, controllando all'avvio dell'applicazione lo schermo in cui la finestra è aperta e mettendo la risoluzione più alta in 16:9 possibile senza uscire dai limiti dello schermo.

Inoltre, un altro controllo fondamentale è stato quello di verificare che quando si prova a modificare la dimensione dello schermo la dimensione non ecceda i limiti dello schermo.

Infine, questa impostazione insieme alle altre viene salvata all'interno del json in modo che venga caricato automaticamente al login.

La creazione dei boost non ha portato nessuna particolare difficoltà in quanto sono facilmente riuscito a riutilizzare il codice già creato in precedenza da Alam MD Shokot per entità con scopo simile modificando e aggiungendo le parti che mi interessavano e aggiungendo e modificando in parte l'implementazione del player

Davide Valdifiori

In questa sezione l'attenzione è focalizzata sull'implementazione del livello di gioco col relativo caricamento della mappa da file e sul sistema dei blocchi.

Livello

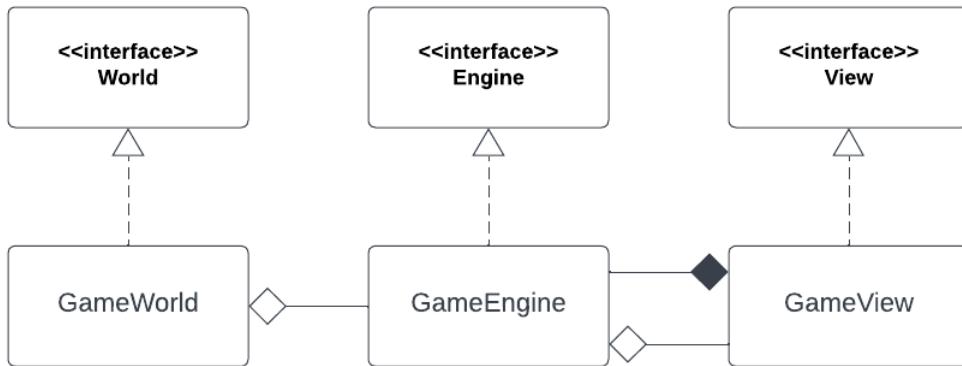


Figura 2.4: Rappresentazione UML del livello di gioco con le classi principali del pattern MVC e i rapporti fra loro.

Problema Ciccio Pier ha più livelli di gioco, deve quindi essere possibile istanziare un nuovo livello partendo da un singolo parametro che lo identifica.

Soluzione Il sistema del livello di gioco utilizza il pattern MVC, come da Figura 2.4. E' formato dalle classi:

- **Model:** World - contiene gli oggetti del livello di gioco
- **Controller:** Engine - gestisce il livello e aggiorna gli elementi in gioco
- **View:** View - mostra a schermo il livello

BlockFactory e EntityFactory

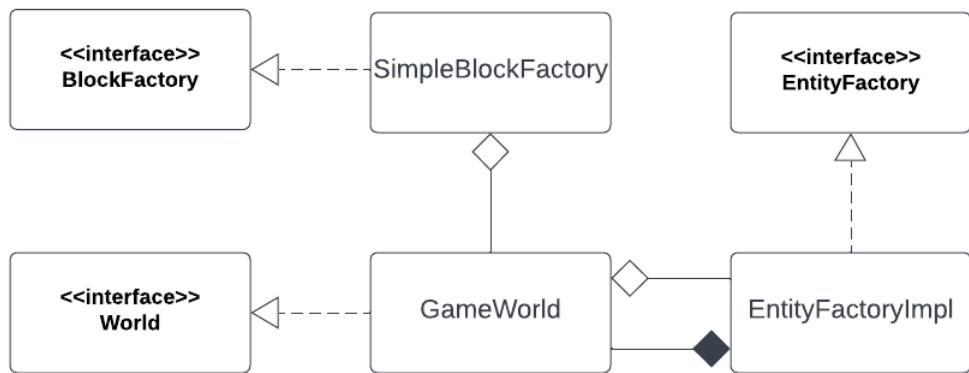


Figura 2.5: Rappresentazione UML della classe GameWorld e delle factories.

Problema Durante la fase di implementazione ho trovato la necessità di implementare un metodo per creare nuove istanze di `Block` ed `Entity` partendo rispettivamente da `BlockType` ed `EntityType`, indipendentemente da quale implementazione delle interfacce venga usata.

Soluzione Ho deciso di utilizzare il factory method, come da Figura 2.5, creando così due factories:

- **BlockFactory** contiene una unica funzione `createBlock(BlockType type)` per creare un nuovo `Block` dato un `BlockType`
- **EntityFactory** contiene le funzioni `createPlayer()` e `createEntity(EntityType type)` per creare rispettivamente il `Player` e una nuova `Entity` data una `EntityType`

WorldLoader

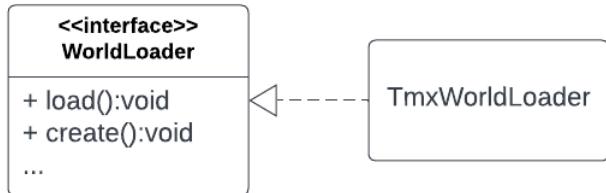


Figura 2.6: Rappresentazione UML del *WorldLoader*, con la sua implementazione *TmxWorldLoader*.

Problema In fase di sviluppo è nata la necessità di creare un modo semplice per caricare i dati del livello di gioco da file.

Soluzione Dato in futuro potrebbe essere necessario poter cambiare tipologia di file, ho optato per la creazione di una interfaccia *WorldLoader*, come visibile in Figura 2.6.

WorldLoader è stata creata con lo scopo di rendere semplice la creazione di una nuova implementazione che possa popolare un oggetto *World* da altri tipi di files con estensioni e formati differenti.

La classe *TmxWorldLoader* è l'implementazione dell'interfaccia *WorldLoader*. Ha lo scopo di caricare tutte le informazioni da un file con estensione *.tmx*, il costruttore ha come parametri il nome del file e una istanza di *World*.

View del Livello

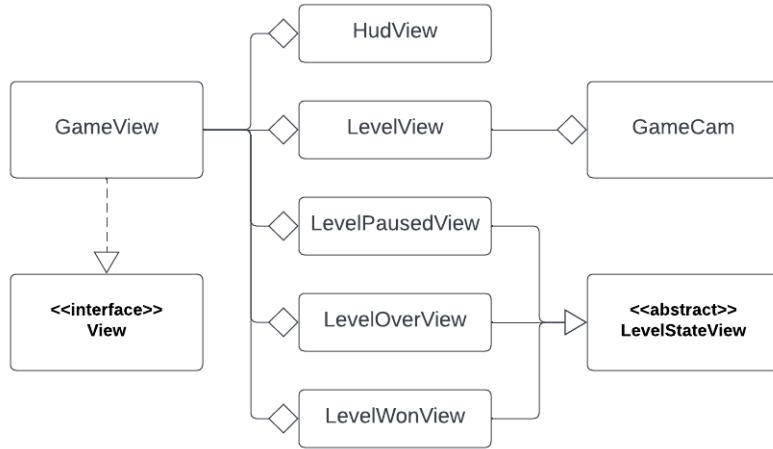


Figura 2.7: Rappresentazione UML della classe GameView.

Problema Rappresentare a schermo il livello di gioco e sovrapporre la visualizzazione di schermate di pausa, vittoria, sconfitta.

Soluzione Utilizzo di una classe **GameView** che estende **JLayeredPane**, permettendo così di inserire pannelli sovrapposti. Come visibile in Figura 2.7, è composta da:

- **LevelView**: viene utilizzato per mostrare a schermo tutti gli oggetti di gioco presenti in World. Al suo interno contiene una istanza di *GameCam*, che viene utilizzata per calcolare quale area del mondo mostrare a schermo, centrando la visuale sulla posizione corrente del Player
- **HudView**: viene utilizzato per mostrare le barre di vita e stamina, le monete e il punteggio del giocatore
- **LevelPausedView**: viene mostrato solo quando il gioco è in pausa
- **LevelOverView**: viene mostrato solo quando il giocatore muore e perde il gioco
- **LevelWonView**: viene mostrato solo quando il giocatore arriva alla fine del livello e vince il gioco

Blocchi

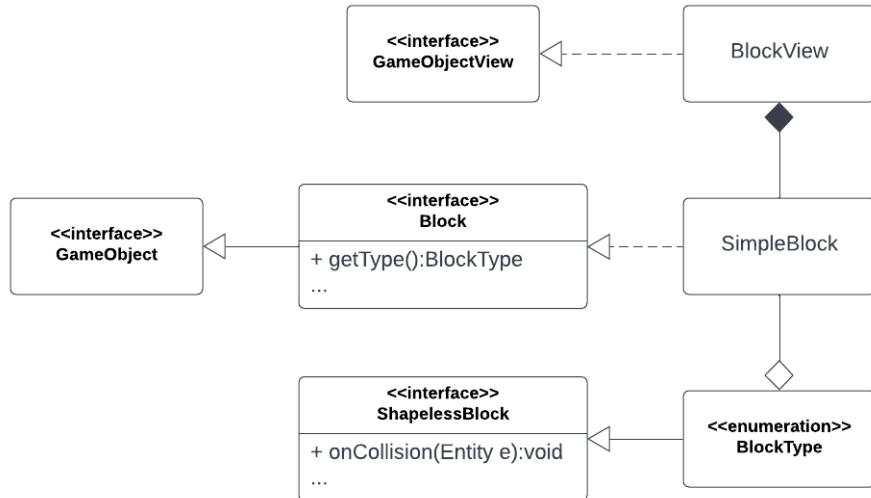


Figura 2.8: Rappresentazione UML delle interface `Block` e `BlockView`, con le loro relative implementazioni e classi associate.

Problema Ogni livello è composto da dei blocchi fisici o decorativi che devono essere rappresentati in gioco.

Soluzione I blocchi non utilizzano il pattern MVC, poichè non devono svolgere nessuna manipolazione sul Model e il Controller diverrebbe inutile. Per ridurre il numero di classi istanziate ho optato per una soluzione in cui esso non è presente, quindi si ha solo l'interface `Block` che estende `GameObject`, come visibile in Figura 2.8. Estendendo `GameObject` si dovrà quindi fornire anche una istanza di `GameObjectView` quando viene chiamata la funzione `getView()`, a cui verrà delegato il compito di mostrare a schermo il blocco.

Block Model

Anche per i blocchi è presente una enumerazione `BlockType` che ne definisce la tipologia e altre proprietà generiche per quel tipo di blocco.

La classe `SimpleBlock` è l'implementazione dell'interfaccia `Block`, il suo costruttore ha come parametro `BlockType`.

E' presente anche una interface `ShapelessBlock` che rappresenta i blocchi non solidi con cui si può interagire, permette di eseguire una azione nel momento in cui il giocatore lo attraversa.

Block View

La classe BlockView è l'implementazione dell'interfaccia GameObjectView, si occupa di rappresentare a schermo un blocco.

Riccardo Mingozi

Entità generali

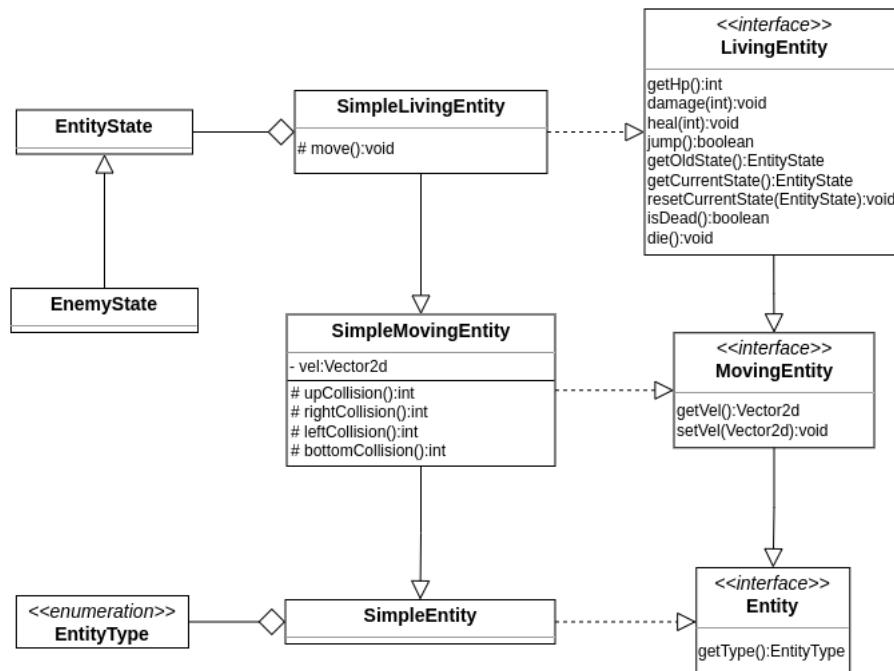


Figura 2.9: Rappresentazione UML della struttura delle classi base delle entità.

Problema Trovare un modo di unificare e generalizzare la rappresentazione di tutte le entità, riuscendo ad astrarre i loro aspetti distintivi principali, permettendo una facile e permissiva estensione o aggiunta di ulteriori entità

Soluzione Ideazione di una gerarchia, consistente in una distinzione tra entità statiche, semoventi e viventi.

La base di questa gerarchia sono infatti le entità statiche, che in composizione con una enumerazione rappresentate tutte le entità, permette di avere un riferimento ad informazioni comuni a tutte, quali larghezza, altezza, even-

tuale vita ed eventuale danno, senza dover salvare queste informazioni in ogni classe individuale.

Le entità semoventi presentano invece un Vector2d aggiuntivo (oltre quello della posizione attuale) rappresentate la sua accelerazione e vari metodi per controllare le collisioni.

La classe infine delle entità viventi presenta campi e metodi per gestire i loro punti vita, metodi per gestire il movimento (salto incluso) tenendo conto di eventuali collisioni, oltre che possedere degli stati, rappresentati dalla composizione con enumerazione. Da quest'ultima classe di entità sono poi state derivate le classi del player e dei nemici.

Nemici

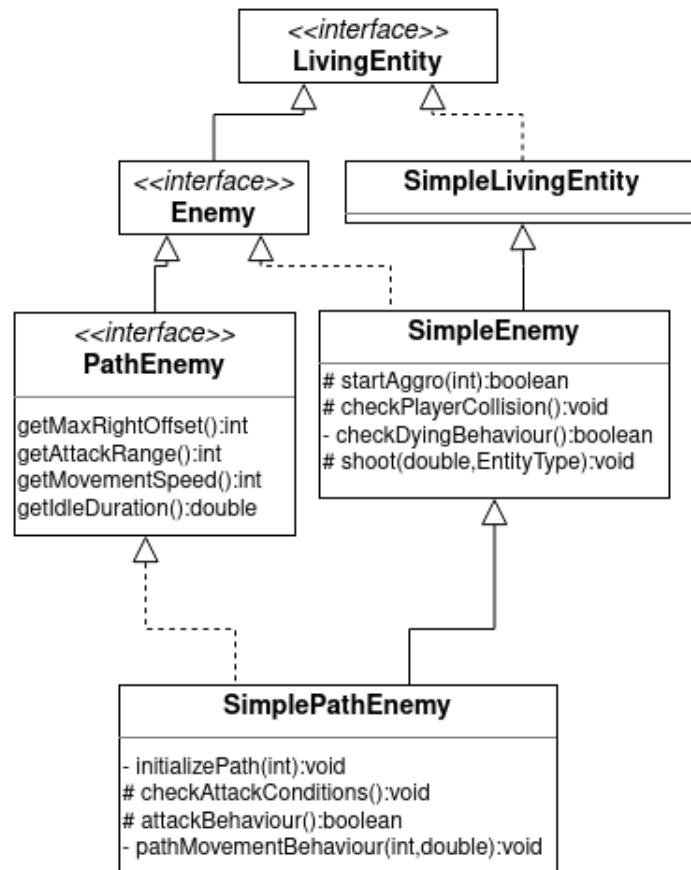


Figura 2.10: Rappresentazione UML della gestione dei nemici.

Problema Trovare un’astrazione rappresentante i nemici, generica e flessibile tale che permetta facilmente di aggiungere nuovi nemici, riducendo gli elementi distintivi da implementare di ogni nemico al minimo.

Soluzione Inizialmente creata una sola classe per i nemici, dove si sono astratti i controlli di morte e di collisione con il player, identici per tutti i nemici.

Successivamente questa classe è stata estesa in un’altra sempre generica, rappresentante la tipologia di nemico più comunemente diffuso nei platform 2D, ossia quello che pattuglia un segmento di mappa, attaccando il nemico solo quando in range.

Questo comportamento è stato dunque astratto, generalizzando il movimento, prendendo solo la lunghezza del segmento e la velocità da parametri individuali per ogni nemico, utilizzando invece un template method per l’attacco. Per quest’ultimo, il template consiste nel chiamare un metodo quando il nemico soddisfa le condizione per iniziare l’attacco ed un altro quando invece non attacca, entrambi poi implementati in modo singolo per ogni nemico.

Proiettili

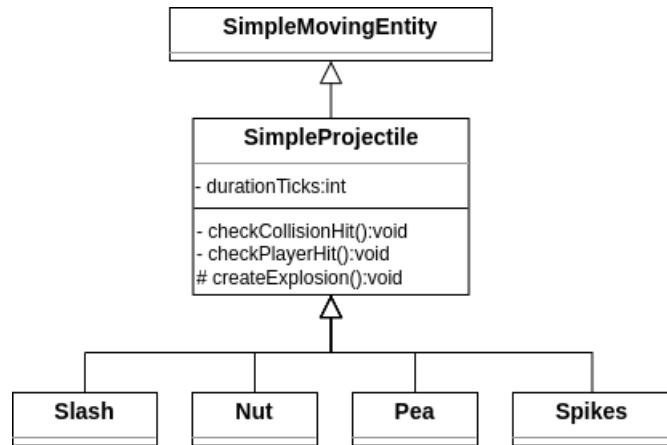


Figura 2.11: Rappresentazione UML dei proiettili e delle relative sottoclassi.

Problema Riuscire a rappresentare i proiettili in modo efficiente permettendo un’estrema facilità di aggiunta

Soluzione Creazione di una classe astratta per i proiettili, estendendo tutto il comportamento, principalmente il movimento, permettendo che l’ag-

giunta di nuovi proiettili richieda, in caso seguano il comportamento comune individuato, solo poche informazioni individuali, quali velocità di movimento e durata.

MD Shokot Alam

Gestione rappresentazione del movimento

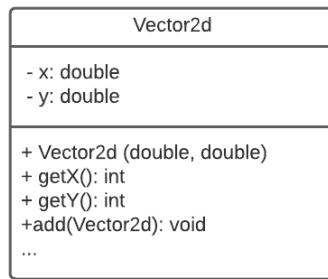


Figura 2.12: Rappresentazione UML della classe Vector2d.

In fase di sviluppo si presenta il problema della gestione del sistema di movimento di qualsiasi entità vivente o non nella finestra di gioco.

Ho dunque proceduto alla creazione di una classe ausiliaria Vector2d preposta al supporto delle altre entità per il posizionamento e metodi adibiti allo spostamento.

Gestione Item

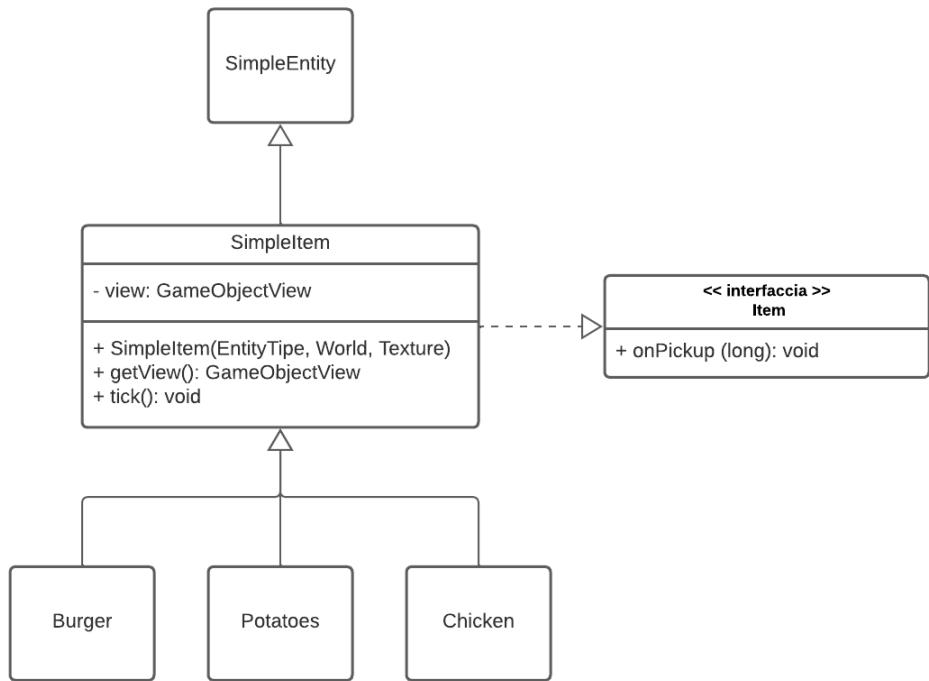


Figura 2.13: Rappresentazione UML della classe SimpleItem e relative estensioni.

La rappresentazione dei vari oggetti es: Potatoes, Burger e i vari potenziamenti ha costituito un problema di duplicazione del codice, per questo ho creato una classe SimpleItem che generalizza le loro funzionalità comuni e ciò ha dato seguito alla realizzazione di una classe analoga per la renderizzazione.

Gestione animation

All'inizio si era manifestata la problematica della diversificazione rappresentativa delle view. Per unificare i vari modi di rappresentazione ho creato la classe Animation.

Gestione suoni

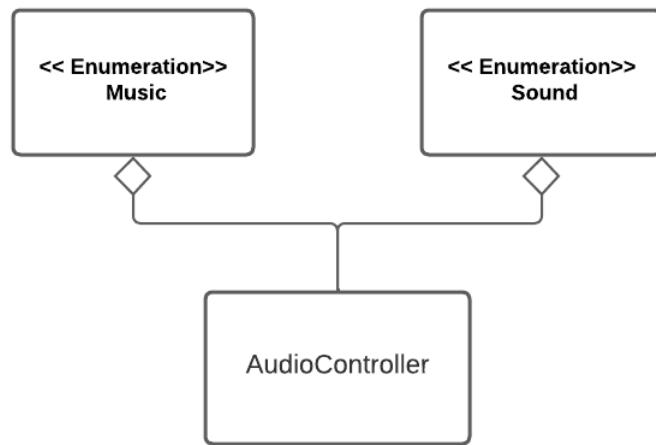


Figura 2.14: Rappresentazione UML della classe `AudioController` e delle enumerazioni `Sound` e `Music`.

Per la gestione dei suoni ho creato due enumerazioni `Music` e `Sound`, una per il controllo della musica e l'altra per gli effetti sonori.

In entrambi ho utilizzato la classe `Clip` fornita da `javax.sound.sampled`, che mi permette la riproduzione di un qualsiasi suono in formato wav.

Successivamente ho realizzato la classe `AudioController` per la gestione del volume e la riproduzioni dei vari suoni.

Gestione Boss

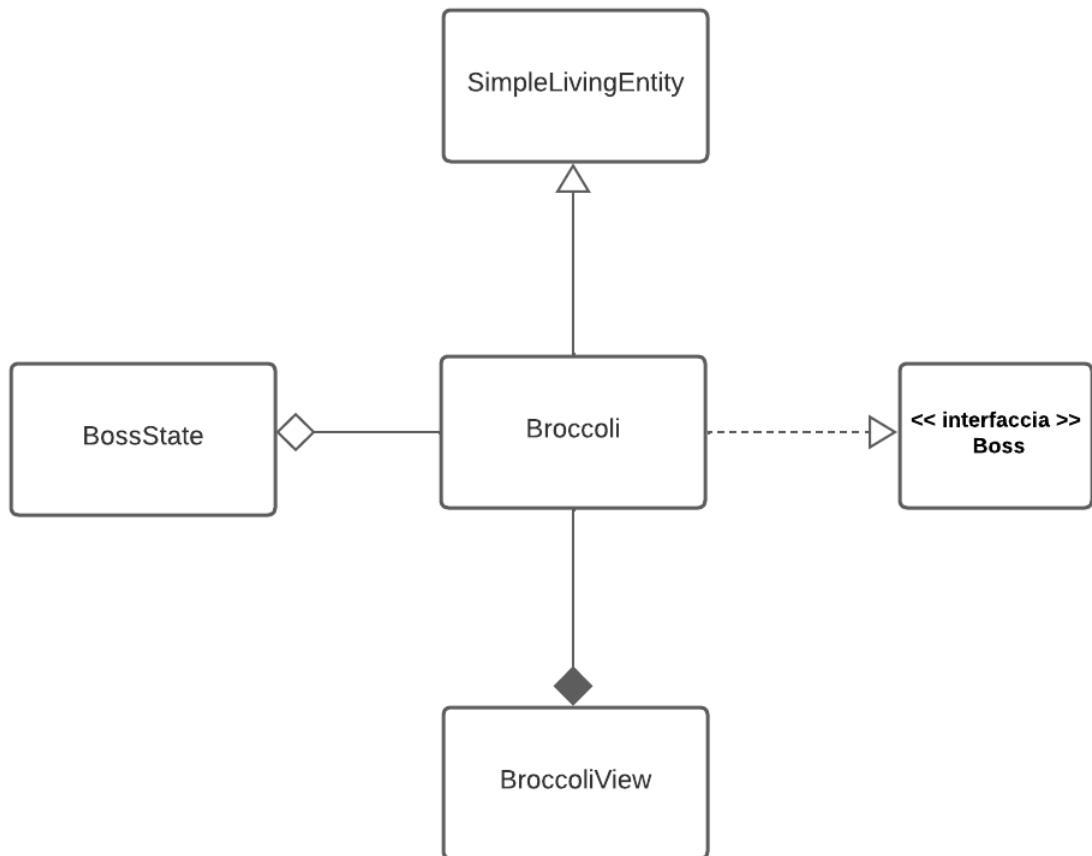


Figura 2.15: Rappresentazione UML della classe Broccoli.

In fase di analisi ho realizzato un nemico che ha più stati, i più importanti sono: lo stato di inseguimento del player e i tre stati dell'attacco.

Per gli attacchi ho cercato di differenziarli tra:

- lancio di missili ad inseguimento
- caduta di meteore
- spara raggio laser

Gestione SimpleEntityView

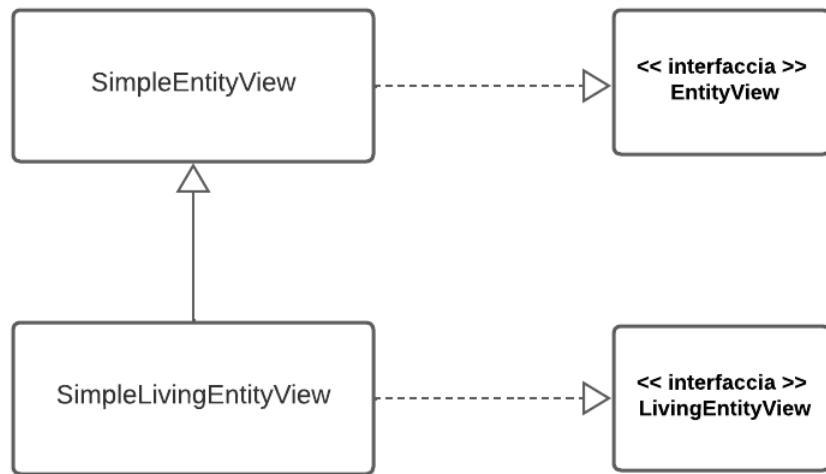


Figura 2.16: Rappresentazione UML delle classi `SimpleEntityView` e `SimpleLivingEntityView`.

In fase di analisi si è presentata la medesima casistica degli item ma in questo caso in rapporto all'entità. ho proceduto alla riunificazione dei vari tipi di entità quali:

- `SimpleEntityView`
- `SimpleLivingEntityView`

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Le problematiche più complesse erano riscontrabili solo in casi limite molto difficili da ricreare in un ambiente automatizzato, per questo abbiamo optato per una verifica manuale del corretto funzionamento di alcune funzionalità del gioco, come per esempio le collisioni delle entità coi blocchi.

Dato che alcuni di noi han programmato e testato su Linux e Mac durante tutta la durata del progetto, siamo sempre riusciti a controllare subito se una funzionalità o una porzione di codice non funzionava sui principali sistemi operativi.

Ognuno di noi ha creato alcuni test in JUnit per testare alcune funzionalità della propria area di progetto:

- **BoostTest:** funzionamento di tutti i boost quando vengono raccolti dal player
- **BossTest:** spawn e cambio di stato del boss
- **EnemiesTest:** spawn e cambio di stato dei nemici e attacco del player
- **GameWorldTest:** gran parte delle funzionalità della classe GameWorld
- **PlayerTest:** punteggio, vita e stamina del player
- **TmxWorldLoaderTest:** caricamento degli elementi di gioco da file tmx
- **UsersTest:** caricamento e salvataggio degli utenti da file json

3.2 Metodologia di lavoro

La fase di analisi e fase di design architetturale sono state svolte in gruppo. Nel corso della realizzazione del progetto, ci siamo periodicamente incontrati virtualmente sulla piattaforma Discord per discutere delle scelte implementative ed aiutarci reciprocamente nel caso di dubbi.

Tuttavia è capitato in alcuni casi che il design progettato non sia stato rispettato perché uno o più componenti del gruppo han trovato una soluzione migliore per realizzare alcune parti del progetto di loro competenza, che han poi portato a volte alla necessità di applicare modifiche ad altre parti. In questi casi si è sempre concordato e comunicato come applicare le modifiche agli altri membri.

Nonostante tutto, la divisione iniziale dei lavori e delle parti è stata rispettata e possiamo affermare che anche la suddivisione dei ruoli è stata equilibrata. Le parti che ciascuno di noi ha sviluppato si sono rivelate essere in alcuni casi leggermente dipendenti fra loro, ma siamo comunque riusciti a portare a termine il progetto senza ostacolarci a vicenda accordandoci in fase di design comune.

Come specificato nella sezione di design dettagliato, qui di seguito la suddivisione dei compiti che ciascun componente ha svolto:

Alessandro Aldini

- Creazione di JComponent personalizzati
- Gestione base di model, view e controller del menu principale
- Gestione impostazioni di volume di gioco
- Gestione leaderboard di gioco
- Gestione risoluzione e ridimensionamento dello schermo
- Gestione di creazione e modifica degli utenti
- Implementazione di boosts

Durante lo sviluppo il mio metodo di programmazione è stato quello di cercare di avere subito delle classi seppur grezze funzionanti, in modo da poter avere una visione di insieme e un'idea su come riuscivo a gestire i tempi.

E successivamente migliorare e perfezionare le classi o il loro funzionamento in modo da rendere il codice più pulito e corretto, questo metodo è

stato molto efficace per me perché durante lo sviluppo riuscivo a notare i punti che erano migliorabili e a modificarli nel tempo, arrivando ad un risultato di cui mi sento soddisfatto.

Inoltre, ho cercato sempre di pensare all'espandibilità del mio codice facendo sì che sia facilmente modificabile, sfortunatamente per via di mancanza di tempo disponibile al progetto rimangono alcune enum che sarebbero dovute diventare classi per poterle rendere perfettamente espandibili.

La parte di integrazione fra GameEngine e MainMenuController è stata sviluppata in collaborazione con Davide Valdifiori.

La parte di implementazione dei boost è stata sviluppata in collaborazione con Alam MD Shokot

Davide Valdifiori

- Gestione base di Model, View e Controller del livello
- Caricamento della mappa da file
- Gestione dei blocchi
- Gestione dell'input da tastiera e controller
- Loop del gioco

La parte di integrazione fra GameEngine e MainMenuController è stata sviluppata in collaborazione con Alessandro Aldini.

Riccardo Mingozzi

- Struttura base delle entità
- Struttura base del player e attacco
- View (inizialmente) e Model di tutti i Nemici

Avevo inizialmente creato un sistema per gestire la view dei nemici, basato su degli stati che venivano aggiornati nel model, in quanto usati per determinare i loro comportamenti.

In fase di analisi avevamo ritenuto corretto dividere le view di nemici, player e boss, in quanto molto diversi come comportamenti. Successivamente abbiamo deciso di astrarre le varie view, estendendo il concetto di rappresentazione basata su stati a tutte le entità.

Questa parte è stata poi implementata dal mio collega MD Shokot Alam.

MD Shokot Alam

- Sviluppo del boss
- Cibi grassi
- Sistema delle monete
- Sistema della stamina e della vita del player
- Collisioni delle entità
- Sistema di movimento delle entità
- View base e animazioni delle entità
- Sistema di suoni e musica del gioco

Per la realizzazione del sistema audio ho riscontrato problemi per la riproduzione di suoni su sistemi Linux, per risolvere questo problema, ho trovato la soluzione su StackOverflow. Ho poi adattato il codice alle mie esigenze.

3.3 Note di sviluppo

Alessandro Aldini

Features avanzate utilizzate:

- Uso di lambda expressions
- Uso di Stream, Optional e altri costrutti funzionali
- Uso di librerie di terze parti:
 - Gson per caricare e salvare gli utenti su un file json
 - SLF4J logger

Davide Valdifiori

Features avanzate utilizzate:

- Creazione classe Pair<T>
- Uso di lambda expressions

- Uso di Stream, Optional e altri costrutti funzionali
- Uso di librerie di terze parti:
 - Libtiled per caricare la mappa da files tmx
 - Jamepad per gestire l'input da Controllers
 - SLF4J logger

Riccardo Mingozi

Features avanzate utilizzate:

- Uso di lambda
- Uso di Stream, Optional e altri costrutti funzionali
- Sviluppo di algoritmo di attacco del player: un metodo che filtra selettivamente le entità in range, facendo gli appositi controlli, attaccando il nemico più vicino che rispetti le condizioni di attacco

MD Shokot Alam

Features avanzate utilizzate:

- Uso di Optional e altri costrutti funzionali
- Uso di parti di libreria non spiegate a lezione:
 - javax.sound.sampled per gestire i suoni e le musiche di gioco

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Alessandro Aldini

Alla fine di questo progetto mi ritengo estremamente soddisfatto del mio lavoro e di questo progetto in generale.

Sono contento del metodo di lavoro che ho deciso di seguire durante questo progetto e sono orgoglioso del mio lavoro considerando le mie conoscenze pregresse e le svariate conoscenze acquisite durante tutta la fase di sviluppo; ma soprattutto sono davvero fiero del risultato finale, tenendo conto che nessuno di noi aveva mai fatto qualcosa di simile e che si tratta comunque di un progetto universitario penso che il prodotto finale sia davvero curato nei dettagli e appassionato.

Penso, inoltre, che ci sia stato un ottimo team work in quanto si era sempre disponibili per confronti e aiuti durante la parte di sviluppo nei confronti di tutti così da evitare possibili problemi e ritardi durante questa parte.

Durante lo sviluppo una pecca che ho notato verso la fine è stata l'aggiunta di JavaDoc dopo aver già pushato le funzioni e classi, questo è dovuto al fatto che spesso si lavorava insieme e si comunicavano i cambiamenti quindi non sentivo che questa parte fosse fondamentale ma mi rendo conto che in un ambiente di lavoro questa rappresenta una mancanza importante e ne ho preso atto.

Sarei interessato nel futuro a proseguire la creazione di questo progetto, portandolo anche nel mondo mobile.

Davide Valdifiori

Sono abbastanza soddisfatto di come ho sviluppato la mia parte del progetto e del risultato finale.

Avendo già conoscenze pregresse in Java sono riuscito a portare avanti la mia parte di progetto senza troppe difficoltà, ma sono comunque riuscito a imparare molto da questa esperienza, soprattutto grazie alla collaborazione dei miei compagni di progetto e alle loro idee.

Questo progetto mi ha insegnato le basi del lavoro di gruppo e tanto altro che da solo non avrei potuto imparare, quindi nel complesso valuto questa esperienza positivamente.

Sarei interessato nel futuro a proseguire la creazione di questo progetto, portandolo anche nel mondo mobile.

Riccardo Mingozi

Considerando come questo sia stato il mio primo approccio al game development, al teamworking, allo sviluppo di un progetto complesso ed articolato, non sono totalmente insoddisfatto del mio risultato.

Riconosco tuttavia che tutto il mio lavoro poteva essere fatto molto meglio, ed infatti questo progetto mi ha illuminato sulle lacune che devo affrontare per approcciarmi a futuri progetti di questa complessità.

Mi ha inoltre insegnato tanto relativamente il lavoro di gruppo, permettendomi di sviluppare parallelamente tante soft skills che sicuramente torneranno utili in futuro.

MD Shokot Alam

Sono molto soddisfatto del progetto eseguito e felice di averlo portato a termine.

L'obiettivo principale che mi ero prefissato era creare un nemico univoco che si autogestisca in base allo stato in cui si trova con un codice pulito e più comprensibile possibile. A causa della mia inesperienza ho dovuto però spesso ristrutturare il codice per eliminare duplicazioni e facilità di utilizzo.

Nel complesso valuto questa esperienza molto positivamente in quanto ha permesso di lavorare in team, una soft skill importante nell'ambito lavorativo.

In futuro vorrei creare altri progetti, anche in settori diversi dall'ambiente gaming.

Appendice A

Guida utente



Figura A.1: Schermata iniziale del gioco.

Quando si apre il gioco viene mostrata la schermata iniziale, come visibile in Figura A.1.

Premendo il bottone in alto a destra si possono cambiare le impostazioni, mentre premendo il bottone in basso a destra è possibile visionare il tutorial del gioco.

Il pulsante *Play* permette di selezionare un livello da giocare, *Leaderboard* permette di visionare la classifica dei giocatori in base al punteggio, *Quit* chiude il gioco.



Figura A.2: Schermata di selezione del livello.



Figura A.3: Schermata di tutorial del gioco.

Per mettere in pausa mentre si sta giocando un livello basta premere ESC.