R'lyeh's Call

Relazione progetto
"Programmazione orientata agli oggetti"
A.A 2020/2021

A cura di:

- Francesco Magnani (Matricola: 0000916594)
- Cavallucci Lorenzo (Matricola: 0000915669)
- Capelli Thomas (Matricola: 0000925150)
- Stricescu Ciprian (Matricola: 0000915547)

Indice

Capitolo 1

Analisi

Il progetto, realizzato per il corso di Programmazione Orientata agli Oggetti, consiste in un gioco roguelike 2D chiamato R'lyeh's Call.
Il genere roguelike è un sottogenere del genere RPG (Role Playing Game) caratterizzato dall'immergersi in dungeon generati proceduralmente.
Il nostro gioco in particolare deve la sua ispirazione al famoso "The Binding of Isaac", seppur mantenendo una sua identità cercando un'implementazione più "arcade" delle varie tematiche del genere. In particolare combina la tematica dei dungeon ad un'ambientazione basata sulle opere di H.P.Lovecraft.

1.1 Requisiti

1.1.1 Requisiti funzionali

- Il gioco dovrà offrire un'esperienza divertente e accattivante al giocatore, guidandolo attraverso stanze dalla difficoltà crescente. Ogni stanza verrà generata in modo procedurale in modo da rendere ogni sessione diversa dall'altra.

- Una stanza contiene un certo numero di ostacoli e nemici dal tipo diverso. Ogni tre stanze quando il numero di nemici vivi arriverà a zero verrà generato un oggetto che fornirà un PowerUp, oggetti in grado di potenziare il giocatore. Il personaggio giocante inizierà con un numero di vite pari a tre che potrà essere aumentato attraverso l'uso di PowerUp o diminuito ricevendo danno dai nemici.

- L'obiettivo finale del giocatore sarà riuscire ad arrivare alla trentesima stanza e superarla, senza mai scendere sotto zero vite.
- I nemici potranno infliggere danno al giocatore collidendo con la sua hit box oppure colpendolo con un proiettile da loro sparato.
- Il gioco dovrà essere portabile su più piattaforme (Windows, Linux e MacOS).

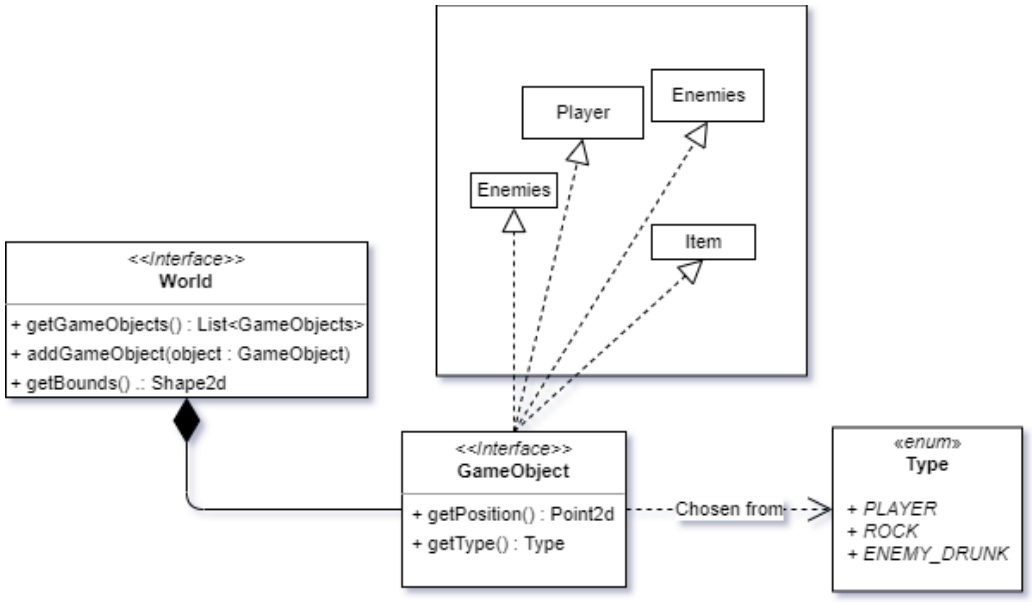
1.1.2 Requisiti non funzionali

- Il gioco dovrò essere parco nell'uso di risorse del sistema.
- L'applicazione potrà essere messa in pausa e ripresa a piacimento da parte del giocatore
- La presenza di una "Developer Mode" che permetterà all'utente di provare gli aspetti chiave del gioco senza dover per forza "giocare".

1.2 Analisi e modello del dominio

Il giocatore dovrà muoversi in una mappa in cui sono presenti sia ostacoli sia nemici. L'obiettivo è quello di eliminare tutti i nemici, sfruttando se possibile anche gli ostacoli presenti nella mappa per ripararsi, ed entrare infine nella botola che lo condurrà al prossimo livello. Ogni dati livelli, il gioco fornirà, a livello finito, un bonus al giocatore raccoglibile sulla mappa, che verrà scelto casualmente tra una serie di PowerUP predefiniti.
La mappa verrà generata ogni livello prendendo in considerazione il numero di stanze esplorate, definendo una lista di entità (nemici ed elementi di mappa) che popoleranno l'ambiente in cui si trova il giocatore. La difficoltà risulterà così incrementale ma nello stesso tempo il giocatore si troverà a essere man mano più forte.
Un aspetto principale risiede nella gestione e controllo delle collisioni tra entità e con il mondo di gioco. La collisione con il mondo di gioco avviene attraverso l'uso dei "Bounds", confine del mondo di gioco pensato come figura geometrica.
Le collisioni tra entità avvengono in maniera diversa a seconda del loro tipo. I nemici, in caso di impatto con entità passive (la cui posizione non cambia una volta generate) o con i confini della mappa stessa, cambiano la propria posizione in modo da "rimbalzare" e riprendere il proprio pattern di movimento. In caso di scontro con il giocatore non ci sarà nessun "rimbalzo".

Il giocatore reagirà anche'esso alla collisione con i "Bounds" in maniera simile ma in caso di impatto con un nemico dovrà perdere una vita e riuscire a continuare il proprio moto, come previsto da input.



Schema UML dell'analisi del dominio di Model, con rappresentazione delle interfacce principali e i rapporti tra loro.

Il giocatore dovrà essere in grado di sparare proiettili ai nemici per attaccare, mentre i nemici potranno essere di vari tipi: un nemico che ti insegue, un nemico che spara proiettili in direzioni casuali, un nemico che invece spara proiettili nella tua direzione e un nemico che invece si muove completamente in modo randomico. Solo a livelli più avanzati potrà comparire invece un altro tipo di nemico, in grado di sparare e muoversi, molto pericoloso e difficile da affrontare.

Gli ostacoli potranno essere di due tipi: un ostacolo simile a "fuoco", che infligge danno se gli passi attraverso, e un ostacolo "roccia" che impedirà semplicemente il movimento, costringendo il giocatore ad aggirarlo.
Problema principale sarà costituito dal rendere i vari tipi di entità riconoscibili e gestibili in maniera univoca da parte del world, minimizzando la ripetitività e massimizzando l'estendibilità del progetto.

Un altro obiettivo non banale sarà riuscire a gestire collisioni e movimento, rendendo la fisica più fluida e naturale possibile e in grado di generare eventi di varia natura (es. game over, raccolta di un oggetto, perdita di una vita ecc.).

Si è ritenuto oggetto di estensioni future, a causa del banco di ore disponibili, l'aggiunta di un nemico "finale" più forte, un boss la cui eliminazione porterebbe alla conclusione e vittoria del gioco. Si è scelto di concludere il gioco, invece, una volta raggiunto un certo numero di stanze esplorate.

Capitolo 2

Design

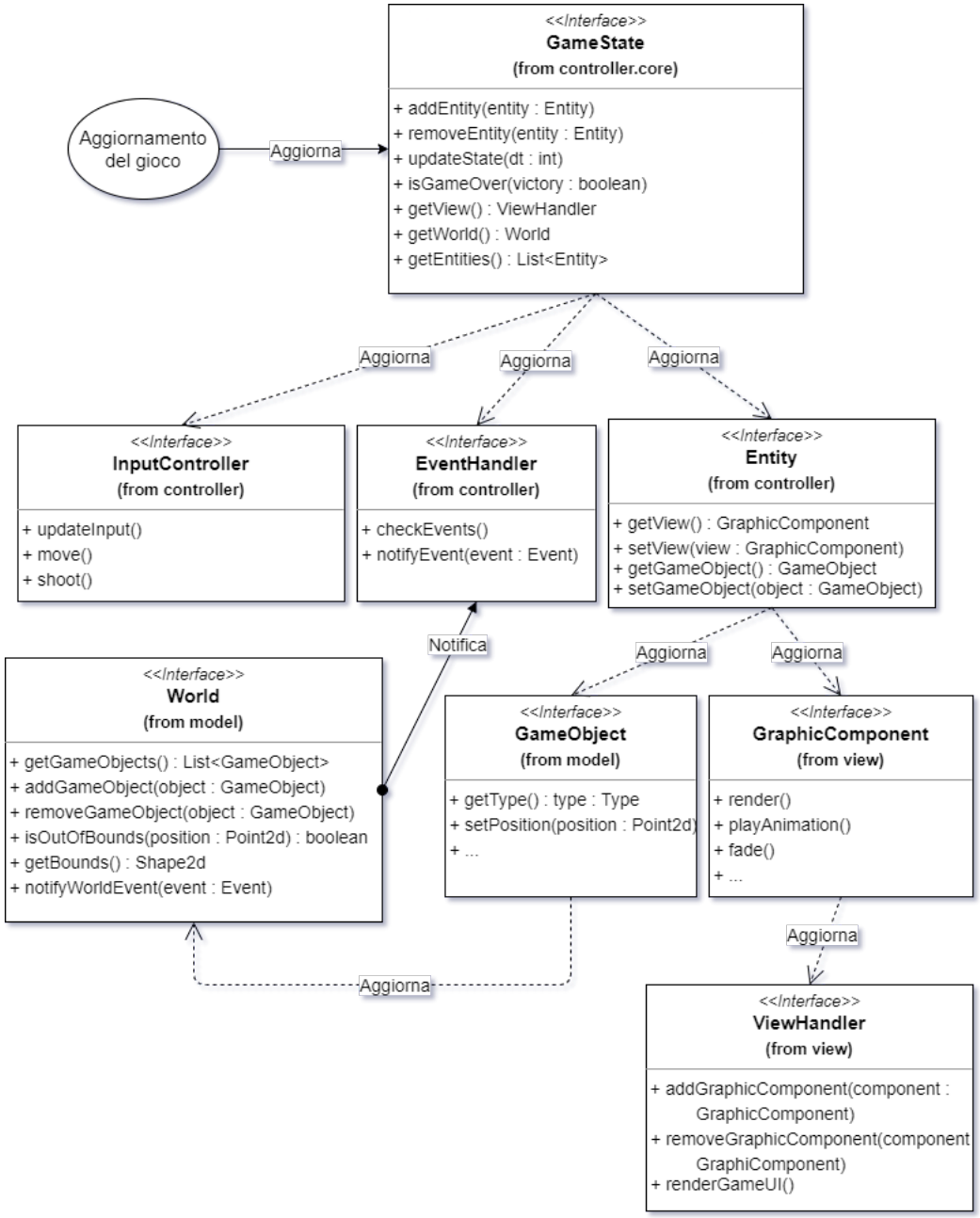
2.1 Architettura

Per garantire una chiara suddivisione del codice da punto di vista pratico e logico abbiamo scelto di utilizzare il pattern architetturale MVC (Model-View-Controller). Questa scelta ci permette di scernere la componente grafica del gioco dalla logica sottostante e il dominio delle entità. In particolare:

- **Model:** Conterrà tutte le varie tipologie di "attori" previste dal dominio del gioco sotto forma di GameObject, e il mondo dove inserirle. Inoltre gestirà l'implementazione della fisica di gioco.
  - **GameObject:** ogni "attore" è un GameObject, dotato di vari Component che ne definiscono i caratteri principali.
  - **World:** gestisce l'aggiunta e la rimozione di GameObject così come la definizione del mondo giocabile.
- **Controller:** Si occupa dell'aggiornamento dello stato di gioco in modo che le informazioni generate dal model vengano sfruttate dalla view.
  - **Entità:** Nel Controller vengono associati i GameObject (appartenenti al Model) ai rispettivi componenti grafici (appartenenti a View) utilizzando dei contenitori che per semplicità chiamiamo entità.
  - **Input:** attraverso l'input da tastiera si aggiornerà il Model, in particolare la posizione del player che verrà resa graficamente attraverso la View.
  - **Eventi:** a seconda di ciò che succede nel mondo di gioco verranno generati degli eventi (perdita di vite, rimozione di entità di gio-

co ecc.). Sarà necessario dunque un impianto che risponda alla chiamata di eventi e che ne gestisca il comportamento.

- **View:** Interagirà con l'utente mostrando le schermate e menu di gioco necessarie. Per una corretta suddivisione, ogni parte della GUI (Graphic User Interface) verrà separata.
  - **GraphicComponent:** ogni entità ha una rispettiva rappresentazione grafica (un GraphicComponent) con un suo comportamento, il quale varia in base alla necessità di animazioni, effetti, ecc.
  - **GameUI:** viene gestita ogni informazione di gioco rilevante per l'utente, come livello corrente o le vite rimanenti, assieme ai vari menù di gioco.



Schema UML che rappresenta la coordinazione tra i vari componenti di MVC

2.2 Design dettagliato

Per meglio spiegare le funzionalità in dettaglio e le varie componenti dell'applicazione che interagiscono tra di loro, abbiamo ritenuto opportuno adottare una visione che mostra in successione ciò che il motore di gioco svolge.
Di seguito verranno spiegate le varie fasi di ogni aggiornamento di quest'ultimo, ponendo attenzione sulle parti a cui ciascun membro ha contribuito maggiormente.

2.2.1 Game Engine e Game State

(A cura di Francesco Magnani)
L'applicativo pone le proprie basi sul Game Engine: una volta avviato il gioco tramite un'interfaccia grafica, il Game Engine entra in esecuzione, prima inizializzando lo stato di gioco e in seguito avviando il Main Loop. Quest'ultimo viene implementato sfruttando una classe della libreria JavaFX, Timeline, che scandisce l'aggiornamento del sistema ogni periodo, valore numerico stabilito come costante e il quale definisce la frequenza di aggiornamento **sia della View, sia del Model**. Diverse soluzioni sono state sperimentate per l'implementazione del loop ma si è deciso di adottare questa seppur dipendente da una libreria esterna di norma utilizzata per scopi grafici in quanto più performante e di alto livello.
Ogni "impulso" del game loop va ad aggiornare il GameState, altra classe chiave dell'applicazione. Mentre il Game Engine non si occupa della gestione del gioco *in sé*, ma solo del *"motore"* appunto, il Game State è la classe forse più importante e imponente del progetto: le sue funzioni sono essenzialmente la gestione di una lista di Entità, quelle presenti in gioco, e dell'aggiornamento in successione di input, game objects, componenti grafiche e gestione di eventi.
Durante la fase di progettazione è sorto più di un dubbio su questo ruolo di Game State, valutando una sua possibile trasposizione in sfera "model" dello stesso o una sua totale eliminazione per lasciare al Game Engine, anch'esso classe di controller, il compito di aggiornare separatamente World e View Handler, classi protagoniste rispettivamente del dominio del gioco e della grafica. Si è infine scelta questa implementazione sia perchè separava in maniera comoda lo *"stato"* e il *"motore"* del gioco, rendendo possibili estensioni future circa la possibilità di iniziare più partite e di salvare i proprio progressi, ma anche perchè meccanismi come quello degli Eventi risultavano di più facile gestione.

2.2.2 Input e InputController

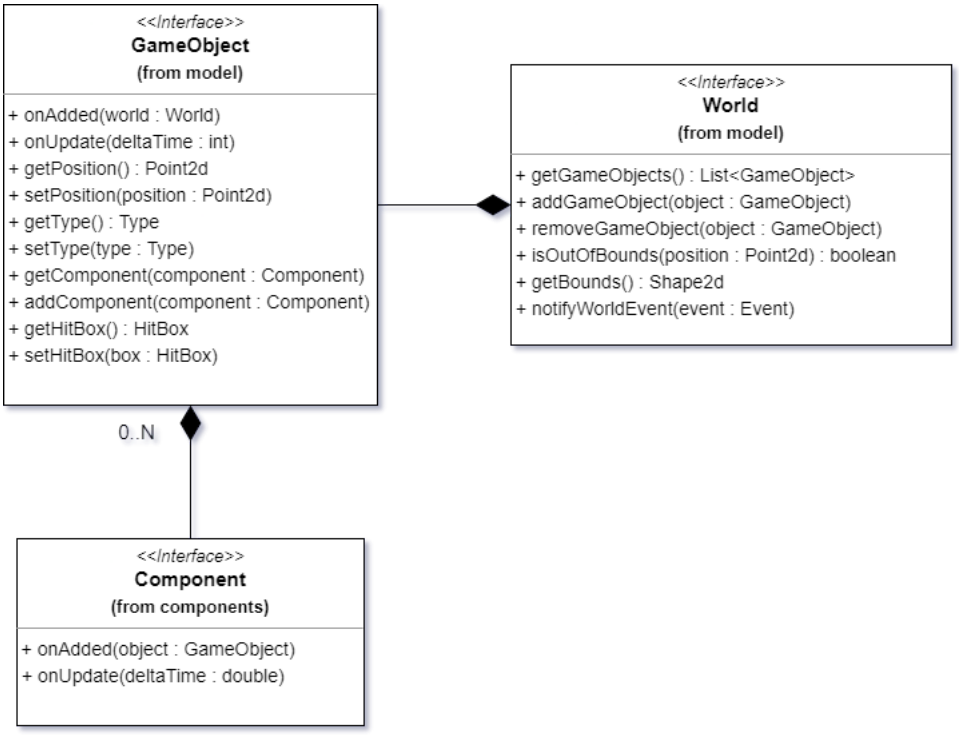
(A cura di Thomas Capelli)
Il movimento del player all'interno della mappa è gestito attraverso l'input da tastiera. In questo caso è stato molto utile l'utilizzo di JavaFX in quanto permette di catturare i vari tipi di input (ad esempio tastiera e mouse) e di gestire le azioni

da compiere quando si verificano determinati eventi come la pressione o il rilascio di un particolare tasto.
Per una questione di gameplay abbiamo deciso di utilizzare solamente W, A, S, D, per muoversi rispettivamente in Alto, a Sinistra, in Basso, a Destra. In particolare, dopo la pressione/rilascio di uno dei tasti elencati, viene aggiornato un componente appartenente solo al giocatore principale, il PlayerComponent, che si occupa di cambiare direzione e velocità. Inoltre sono utilizzati altri tasti come:

- SPACE: si appoggia allo ShootingComponent per spawnare un proiettile nella direzione in cui ci si sta muovendo.
- L: funziona solo se la Developer Mode è attiva e permette di far avanzare arbitrariamente il conteggio dei livelli senza necessariamente sconfiggere tutti i nemici. Utile a scopo di debug.
- K: anch'esso funzionante solo in Developer Mode, permette di raggiungere direttamente l'ultimo livello.
- ESCAPE e P: permettono di mettere in pausa il gioco e di accedere all'apposito menu.

2.2.3 GameObject

(A cura di Francesco Magnani)
Un GameObject è la parte di "model" di ciascuna entità: esso contiene tutte le informazioni che player, nemici, ostacoli o qualsiasi altro oggetto presente in gioco necessita per essere riconosciuto come tale. Ogni Game Object viene prima istanziato e poi aggiunto ad un World, ossia la classe che incarna il vero *dominio* del gioco, compiendo azioni in entrambi i momenti per settare il proprio stato interno. Per la progettazione di questa classe è stata compiuta una scelta chiave che poi ha riadattato tutto il dominio dell'applicazione, ovvero quella di rendere ogni Game Object indistinguibile dall'altro dall'esterno, senza utilizzare ulteriori sottoclassi o specializzazioni di esso. In altre parole, ciò che distingue ogni Game Object è, in primo luogo, l'insieme dei suoi attributi, ma soprattutto la sua lista di "Componenti", classi che ne definiscono il comportamento e che possono essere aggiunti ad un Game Object durante la sua creazione e non solo (più dettagli in 2.2.4).
I GameObject inoltre dispongono di una *Hit Box*, classe che usa forme geometriche bidimensionali per riconoscere la collisione con altri oggetti di gioco.

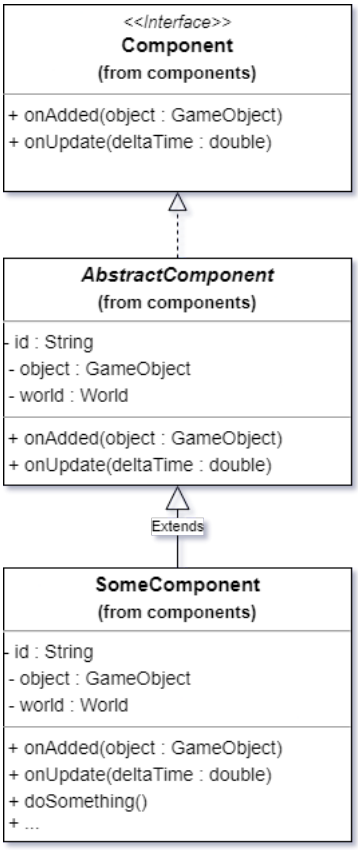


Schema UML che rappresenta gli aspetti chiave del dominio di Ryleh's Call

2.2.4 Componenti e Nemici

(A cura di Ciprian Stricescu)
La scelta di utilizzare i GameObject (2.2.3) sopra citati viene accompagnata da una volontà di rendere gli "attori" una specializzazione di un componente modulare, *Component*. Attraverso l'uso di componenti per realizzare il Model ci siamo proposti un sistema in cui la modularità e la facile aggiunta ad un eventuale GameObject sono il fulcro. Definendo un aspetto comune a tutti i componenti l'aggiornamento da parte di Controller (*GameState*) risulta facile e chiaro, infatti tutti i componenti ereditano il metodo *onUpdate* di Component, essenziale per una visione "componibile" di nemici, oggetti, ostacoli ed il player.
Per rendere ogni Component simile dal punto di vista logico ma "unico" nel suo rapportarsi al mondo sono stati creati vari pattern comportamentali (ad esempio creando varie tipologie di nemici). Utilizzando principalmente un'implementazione di punti e vettori 2d è stato possibile differenziarne il comportamento.
Si è scelta una visione del pattern di movimento dei nemici semplice e familiare, ricadendo in alcuni stereotipi del genere a cui ci siamo ispirati. Per quanto riguarda il pattern d'attacco grazie alla modularità dei componenti siamo riusciti a differenziare quelli dediti al movimento (Es. DrunkComponent) da questi ultimi (Es. ShootingComponent).
Un componente comune e fondamentale per tutti i nemici è il *CollisionComponent*

in quanto permette di rapportarsi agli Eventi (2.2.6) per segnalare la collisione con un altro GameObject, aspetto critico del game design. La collisione con il mondo di gioco viene affrontata a livello di singolo componente, preferendo così una gestione più "specializzata" vista la diversità nei pattern dei nemici (Es. nemici statici e non).



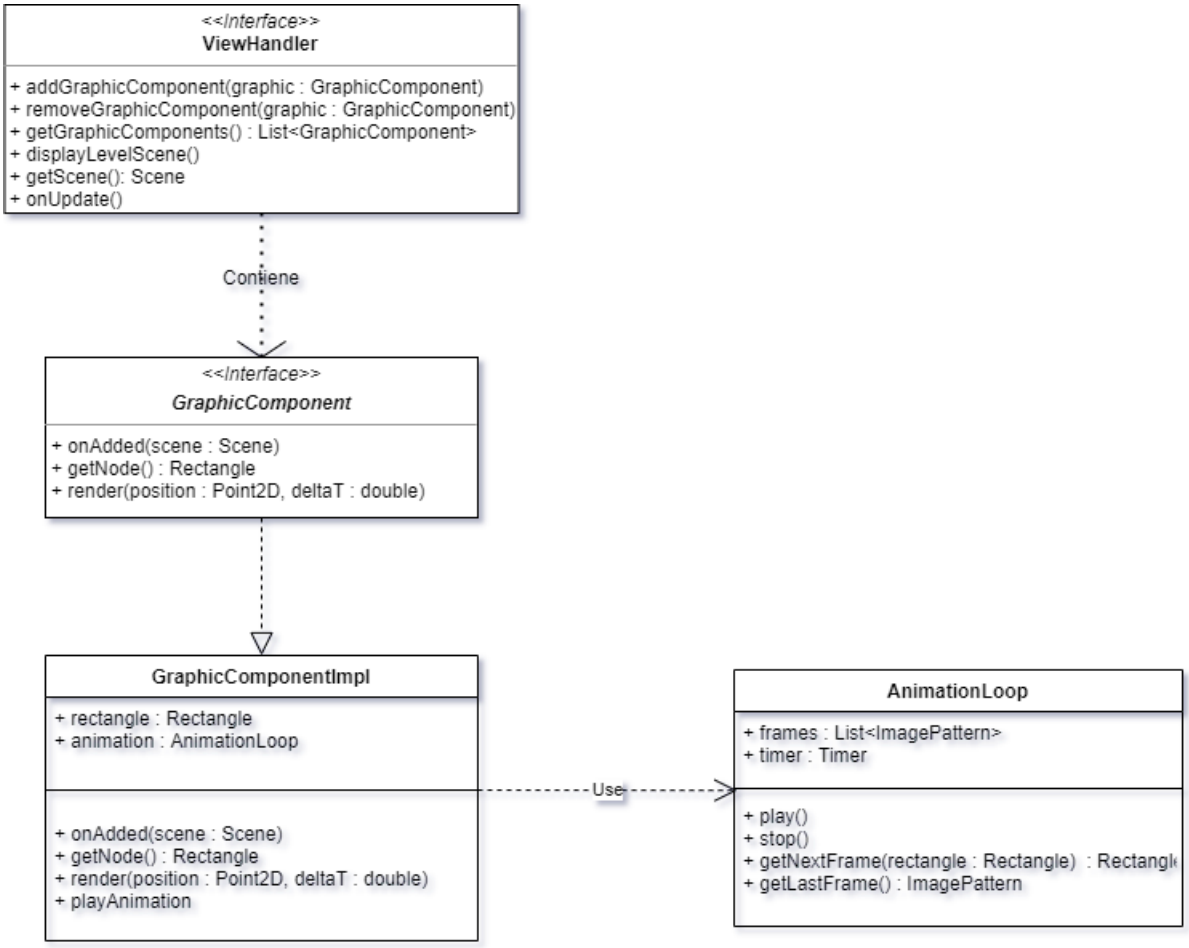
Schema UML che rappresenta il sistema di Component

2.2.5 GraphicComponents

(A cura di Lorenzo Cavallucci)
Ciascuna Entità presente nella scena corrente, ha una sua corrispettiva componente grafica. Qui viene in aiuto l'interfaccia GraphicComponent, la quale permette di essere implementata da ogni tipo di oggetto che deve avere una corrispettiva View, e quindi una visualizzazione a schermo. Perno centrale di ogni graphic component è chiaramente la renderizzazione degli stessi: viene quindi eseguita una render singolarmenre per ciascun componente grafico, su richiesta di quello che è il GameState, di cui abbiamo già esplicato il funzionamento. Chiaramente, questa azione di aggiornamento può variare a seconda delle specifiche caretteristiche delle entità in gioco: se ad esempio un oggetto statico come la roccia non necessita di

alcun cambiamento grafico nel corso del tempo, il Player dovrà invece rispettare l'input fornito dall'utente, spostarsi quindi nella mappa e, in questo caso, eseguire delle animazioni sufficientemente responsive. Per far fronte a tale difficoltà, l'ereditarietà ha se non altro permesso di lavorare in maniera estendibile ad ogni singolo componente. In certi casi, in più, vengono richieste animazioni, per le quali è stata creata una classe a se, AnimationLoop. Tramite il passaggio di un Rectangle (elemento base di ogni GraphicComponent) e della lista di frame che dovranno costituire l'animazione, quest'ultima sarà in grado, sfruttando un timer a lei interno, di aggiornare i frame in base a ciò che viene richiesto: ad esempio, nel caso del Fuoco, questo avrà un loop infinito di un paio di frame, mentre il Player seguirà la direzione dall'utente indicata, con quindi animazione che lo segua adeguatamente.

Per mettere in atto tutte queste caratteristiche e comportamenti, il ruolo da protagonista lo ha l'interfaccia ViewHandler la quale ha svariati compiti, ciascuno dei quali richiesto dal GameState. Essenzialmente, permette di tenere traccia di tutti i GraphicComponents che troviamo a schermo, ne richiama la rimozione e/o aggiunta al momento opportuno, ed è la responsabile della generazione della Scena corrente al passaggio ad un nuovo livello, o, più in generale, ad uno stato di gioco differente. Successivamente, essa ha il compito di scalare gli elementi a schermo su richiesta dell'utente, sistemando quindi le dimensioni di Game UI, Background e i GraphicComponent presenti. Infine, la scelta di costruirla come un'interfaccia viene dalla possibilità ad essa intrinseca di generare, eventualmente, più scene a schermo.



Schema UML che rappresenta le GraphicComponents appartenenti alla View

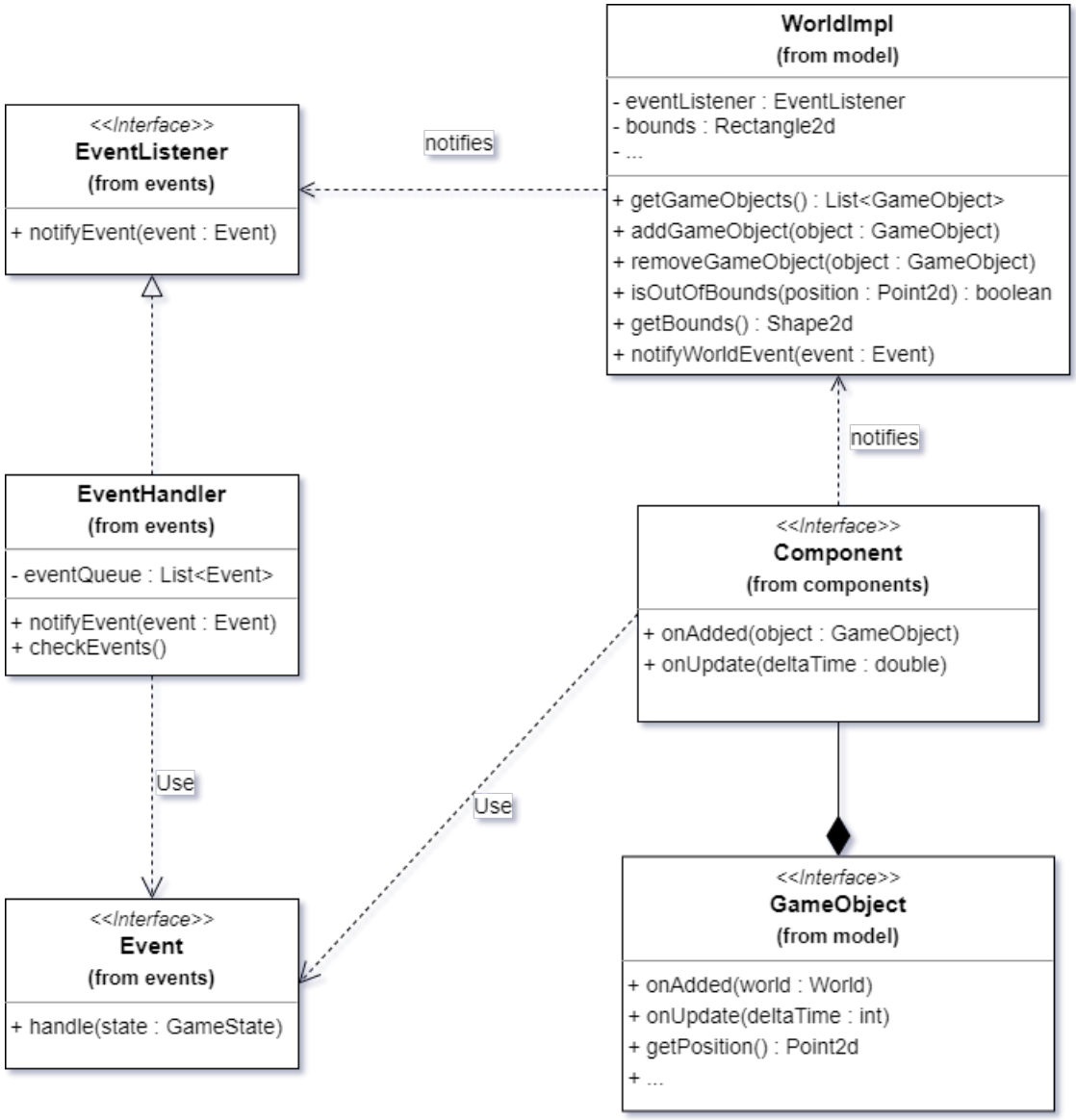
2.2.6 Eventi

(A cura di Thomas Capelli)
Premessa: tra le varie proposte per la gestione di eventi abbiamo inizialmente considerato di controllarne i comporamenti internamente alla classe EventHandler. Abbiamo, tuttavia, constatato che così facendo la classe in questione sarebbe diventata troppo ricca e caotica. Di conseguenza abbiamo deciso di dividerne il decorso di ciascuno di essi all'interno delle singole classi di evento, consapevoli della possibile violazione di MVC che ciò comporta.
Gli eventi rivestono un ruolo fondamentale nell'impianto del nostro progetto, infatti è grazie ad essi se una gran parte di azioni e cambiamenti nel Model vengono trasposti graficamente sulla view. Con eventi si intendono tutte quelle situazioni, particolari e non, che avvengono man mano che si procede con una partita e che necessitano di una gestione immediata (es. GameOver, rimozione di entità dalla mappa, raccolta di oggetti..).

E' stata dunque individuata una serie di eventi diversi fra loro e accorpato altri con comportamento uguale o simile, per ottenere una lista poco estesa in cui ognuno possiede un'implementazione interna diversa. Ruolo importante è svolto dalla notifica di essi, che deve avvenire al seguito di condizioni specifiche che ho verificato andando a creare nuove classi, come BulletComponent, o modificandone altre create dai miei compagni, come ad esempio HealthIntComponent. La notifica di eventi non fa altro che aggiungerne un determinato tipo ad una lista; questa lista è successivamente gestita ad ogni gameloop dall'EventHandler che controlla i suoi elementi e ne esegue i comandi grazie ad un metodo handle che è implementato da ogni classe grazie all'interfaccia Event.
Particolare attenzione è da rivolegere all'ItemPickUpEvent che viene lanciato alla collisione tra Player e l'entità Item. Infatti viene istanziato, in modo pseudo randomico, uno dei Power Up attribuibili al player:

- Heal: Cura il player di tutte le vite mancanti.
- FireSpeedUp: Aumenta permanentemente la velocità di fuoco del giocatore, andando a ridurre il delay tra un proiettile e un altro.
- MaxHealth: Aumenta il numero di vite massime di una unità.

Oltre ad applicare uno di questi effetti, viene anche mostrata una scritta che rappresenta il tipo di oggetto che è stato casualmente scelto.



Schema UML che rappresenta il funzionamento dell'EventHandler ed EventListener

2.2.7 Livelli

I livelli sono i vari step che scandiscono il ritmo di gioco: ogni livello consiste essenzialmente in una lista di Entity, comprendente quindi sia i tipi di "attori" presenti, sia altre informazioni come la loro posizione. Visivamente un livello è una stanza che il giocatore esplora e in cui si trova ad affrontare le minacce già descritte. A parte il primo livello che rappresenta un tutorial, i livelli seguenti vengono generati attraverso gli eventi sopra citati, delegando la generazione a due classi LevelDesigner e LevelHandler.

2.2.7.1 LevelDesigner

(A cura di Thomas Capelli e Lorenzo Cavallucci)
Questa classe si preoccupa essenzialmente di scegliere quali tipi di entità e il rispettivo numero andranno a comporre il livello. Essendo una classe di utility, sfruttata dal LevelHandler, la difficoltà principale è risultata essere l'implementazione di algoritmi basati su funzioni matematiche che simulassero una generazione procedurale con difficoltà crescente all'aumentare del numero di stanze esplorate.

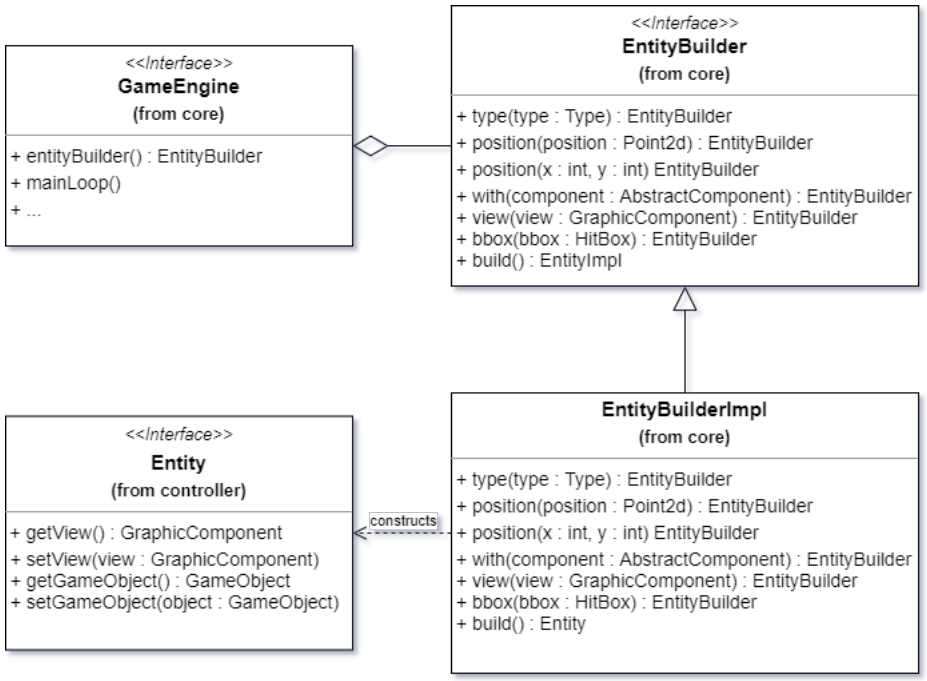
2.2.7.2 LevelHandler

(A cura di Ciprian Stricescu e Francesco Magnani)
Il LevelHandler è il maggior responsabile della generazione dei livelli. Sfruttando il LevelDesigner, il compito di questa classe è quello di formare la vera e propria lista di Entity che andranno a definire il livello. Per fare questo, è risultato necessario definire un sistema di coordinate utilizzabile per una generazione di tipo procedurale, più semplice che un puro posizionamento tramite x e y cartesiani. Si è deciso, quindi, di trattare la mappa come una griglia di "spawn points", ovvero celle dove il sistema decide di collocare le Entity in base ad opportune scelte realizzative e di gamedesign. L'implementazione risulta essere una mappa che associa ad ogni spawn point la relativa Entity.

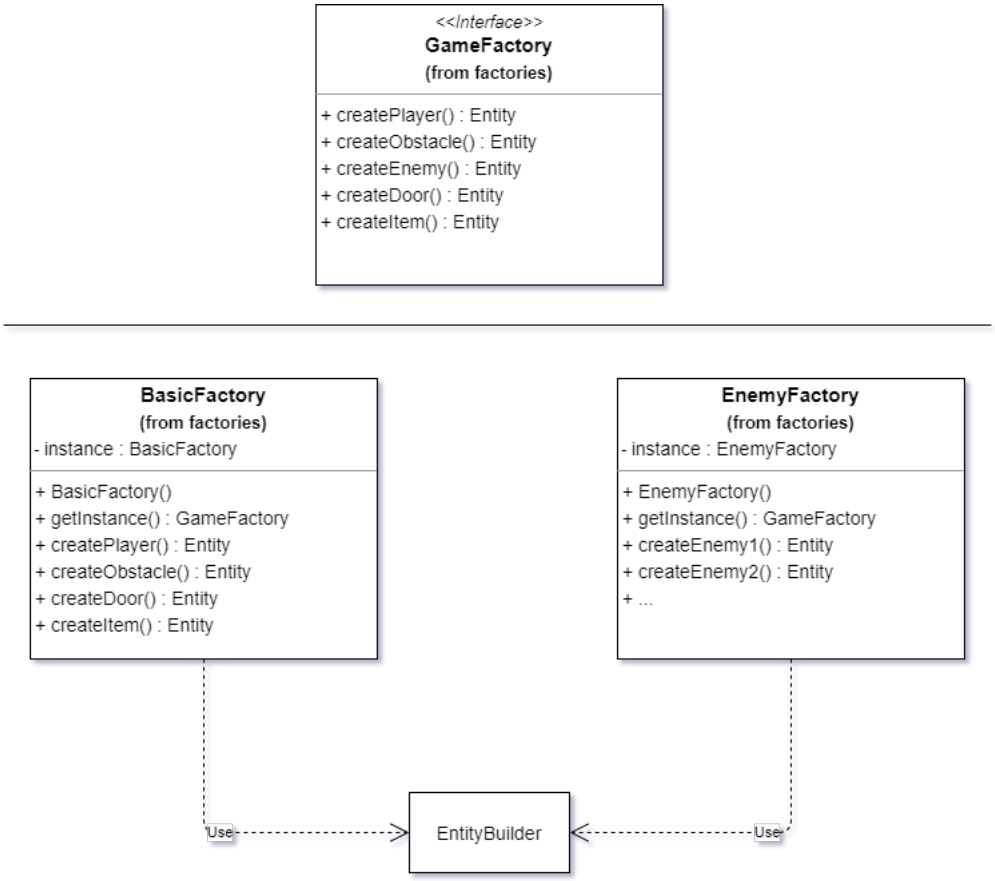
2.2.8 Generazione Entità

(Parte in comune)
Un'Entità, come già citato precedentemente, è l'associazione di componenti di Model (quali vita, fisica, hitbox..) a componenti grafici specifici. Per una generazione agile di quest'ultime abbiamo deciso in fase di progettazione, consapevoli dei rischi associati ad esso, di utilizzare il pattern Singleton all'interno di classi chiamate factory. Una factory sfrutta vari metodi per generare entità specifiche, ognuna con le proprie caratteristiche e Singleton aiuta ad accedervi da qualsiasi punto del codice. Per automatizzare tale processo abbiamo inoltre sfruttato un altro pattern,

Builder, all'interno di un'altra classe chiamata EntityBuilder.
EntityBuilder è la classe principale per la costruzione di oggetti di tipo Entity. Attraverso il suo uso è possibile personalizzare un GameObject in tutti i suoi aspetti (Es. aggiunta di componenti) e associarlo ad un componente grafico. Il risultato finale è un'Entity, contenitore di entrambi. Per concludere, fattore importante sono i comportamenti da attuare all'aggiunta/rimozione al/dal mondo di gioco: quando necessario, verranno eseguite azioni che possono variare in base al tipo di entità, sia al momento di aggiunta (aggiunta alla Scena, eventuale animazione), che a quello di rimozione (transizioni prima della loro eliminazione dalla View, eventuali animazioni).



Schema UML che rappresenta l'EntityBuilder e il relativo pattern Builder



Schema UML che rappresenta le GameFactory e il pattern Singleton

2.2.9 Main Menu

(A cura di Thomas Capelli)
Per la realizzazione della parte grafica del nostro progetto abbiamo scelto di utilizzare la libreria **JavaFX** e dunque anche per il menu principale ho approfondito alcuni degli elementi di quest'ultima, per realizzare un'interfaccia grafica il più semplice ed intuitiva, ma allo stesso tempo accattivante, possibile.
Tramite questa UI è possibile svolgere azioni quali avviare una partita e uscire dal gioco, inoltre è possibile scegliere una risoluzione della finestra di gioco tra alcune disponibili di default; ciò non nega ovviamente la possibilità di ridimensionare in seguito la finestra per ottenere le dimensioni più adatte al proprio piacimento.
Per far si che ogni menu avesse elementi con lo stesso stile grafico abbiamo creato una sorta di factory per la creazioni di elementi basici, che successivamente possono essere posizionati (ed eventualmente personalizzati) diversamente in ogni specifica UI. Allo stesso modo, per isolare al meglio la parte di view dal resto, le interazioni con l'utente e le rispettive operazioni da svolgere sono gestite da una classe comune appartenente al Controller.

2.2.10 Game UI

(A cura di Ciprian Stricescu)
Per la realizzazione del game UI (User Interface) si è scelto di optare per semplicità e chiarezza, in concordanza con il game design. Gli elementi sono quattro: tutorial, contatore delle vite, contatore del livello ed effetto dell'oggetto raccolto.
Essendo una parte di View l'aggiornamento del testo a schermo avviene in sincrono con gli altri elementi del mondo, sfruttando così gli Eventi (2.2.6). Questa scelta ci permette di far visualizzare a schermo le informazioni essenziali al giocatore in maniera veloce e dinamica, fornendo tutte le informazioni di cui ha bisogno per arrivare alla vittoria. L'unico elemento di game UI a non sfruttare gli Eventi è il tutorial vista la sua "staticità". All'inizio di una nuova partita il giocatore verrà inserito in una stanza priva di altre entità il cui unico scopo è di insegnare ad un nuovo giocatore i comandi e l'obiettivo del gioco. A tal proposito si è scelto di inserire nel pavimento della stanza una descrizione testuale che non comparirà in altre stanze ad eccetto della prima.
Altra "unicità" è presente nella comparsa e scomparsa da schermo dell'effetto di un oggetto raccolto dal giocatore. In modo da non rendere la schermata di gioco troppo caotica si è deciso di far partire una *FadeTransition* (effetto grafico) ogni qualvolta tale evento accada, facendo comparire il testo solo per qualche secondo.
La scalabilità è assicurata grazie all'interrazione con altre parti della View, fornendo un risultato rapido in caso di ridimensionamento della finestra di gioco.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per la realizzazione di testing automatizzato abbiamo ritenuto opportuno usare la libreria JUnit 5. I test sono due e si focalizzano sui *WorldBounds* e sul *GameOver*.
Il test sui World bounds si concentra sulla corretta funzionalità della collisione tra entità e mondo, in particolare tra un *Player* e il rettangolo che definisce i confini di una stanza. Generato un GameObject, a cui si aggiunge un *PlayerComponent*, lo si fa muovere in una certa direzione controllando l'eventuale presenza di collisioni con il mondo. Si verifica la corretta gestione della posizione, *Hit Box* e aggiornamento dei componenti di un GameObject.
Il test dedicato al GameOver verifica il corretto funzionamento di alcuni componenti e la loro interazione con gli eventi. In particolare, dato un GameObject avente un *HealthIntComponent* a cui si sottrae un vita ogni *frame*, si controlla la corretta chiamata ad un *GameOverEvent* quando il suo "trigger" viene attivato.

3.2 Metodologia di lavoro

Il lavoro sin dal suo stato embrionale è stato pensato per essere suddiviso in parti eque ai membri del progetto.
Ogni membro ha svolto in autonomia la propria parte in modo da limitare il lavoro "collettivo" e lavorare sulla nostra coordinazione a simulazione di un vero ambiente lavorativo, riuscendo così a sviluppare alcune funzionalità ritenute opzionali. Per quest'ultime si è deciso di collaborare per il raggiungimento di una visione comune
Tutto il progetto è stato svolto grazie all'integrazione di GitHub, consentendo ad ognuno di poter lavorare e contribuire autonomamente al progetto, contornate da videochiamate per il controllo della visione d'insieme.
A lavoro finito possiamo constatare che i ruoli sono stati rispettati come concordato dal piano illustrato.

- **Francesco Magnani**
  - *Game Engine e Game State*: Package controller.core
    - GameEngine (scheletro della classe più varie aggiunte)
    - GameState (scheletro della classe)
    - Entity
  - *GameObjects*: Package model
    - GameObjectImpl
    - ShootingComponent
    - PlayerComponent
    - HitBox (più classi correlate es. Rectangle2d e Circle2d)
    - World (scheletro della classe più varie aggiunte)
  - *Generazione di Entità*: Package controller.core
    - EntityBuilder
  - *Menù*: Package view.menu
    - RylehPauseMenu (menù + funzione di pausa del GameEngine)
    - RylehGameOverMenu (menù + chiamata di GameOver)
  - In comune con il collega Ciprian Stricescu:
    - *Gestione di un nuovo livello*: Package controller.levels
- **Thomas Capelli**
  - *InputControllerImpl e InputController*: Package controller
  - Events: Package controller.events
    - Event
    - EventHandler
    - EventListener
    - BulletSpawnEvent
    - EnemyCollisionEvent
    - GameOverEvent
    - ItemPickUpEvent
    - NewLevelEvent
    - RemoveEntityEvent
  - *Items*: Package model.items
    - Interfaccia Item
    - FireSpeedItem
    - HealItem

        * MaxHealthItem
    - *BulletComponent*: Package model.components
    - *GUI*: Package view.menu
        * RylehMainMenu
        * MenuFactory e MenuFactoryImpl
    - *Timer*: Package common
        * Timer e TimerImpl
    - In comune con il collega Lorenzo Cavallucci:
        * LevelDesigner: Package model
    - Modifiche a classi dei colleghi:
        * HealthIntComponent: per lanciare l'evento di game Over o di rimozione di una entità, e per aggiungere l'immortalità del giocatore
        * ViewHandlerImpl e tutte le componenti grafiche: per rendere resizeable la finestra e tutto ciò che si trova al suo interno

- **Lorenzo Cavallucci**
    - *View* : Package view
        * Textures
        * AnimationLoop
    - *GraphicComponents* : Package view.graphics
        * GraphicComponent
        * PlayerGraphicComponent
        * package graphics.other
        * package graphic.enemies (in comune col collega Ciprian Stricescu)
    - *Comportamento entità Door* : Package model.components
        * DoorComponent
    - *Factory relativa a porta e fuoco* : Package controller.factories
        * basicFactory.createDoor
        * basicFactory.createFire
    - *Ricerca e realizzazione assets grafici* : Folder resources
        * Package assets.texture
    - In comune con il collega Thomas Capelli
        * LevelDesigner : Package model
    - Modifiche a classi dei colleghi:

        * GameStateImpl: sort delle entità secondo il loro zindex per un corretto rendering
        * ItemPickUpEvent: interazione con l' animazione della Componente Grafica dell'Item

- **Ciprian Stricescu**
  - *Component e nemici*: Package model.components
    * DrunkComponent
    * LurkerComponent
    * ShooterComponent
    * SpinnerComponent
    * CollisionComponent
  - *Raggio dell'Hit Box delle entità*: Package model.physics Ma salve, come la va?
    * HitBoxType
  - *Game UI*: Package view
  - *Gestione update UI*: Package controller.events
  - *Ricerca e predisposizione dei font di gioco*: Folder resources
    * Package assets.fonts
  - In comune con il collega Francesco Magnani:
    * *Gestione di un nuovo livello*: Package controller.levels
  - In comune con il collega Lorenzo Cavallucci:
    * *Gestione pattern grafico dei nemici*: Package view.graphics.enemies
  - In comune con il team:
    * *Generazione delle entità*: Package controller.core.factories

3.3 Note di sviluppo

3.3.1 Francesco Magnani

- *JavaFX*: elementi grafici di base (bottoni, testi ecc) per il menù di pausa e di game over + la classe Timeline per l'implementazione del GameEngine.
- *Optional*: utilizzato soprattutto nei GameObject per ottenerne i componenti.
- *Reflection*: sempre all'interno dell'interfaccia GameObject ho utilizzato la reflection per determinare la classe dei Componenti.

- *Math*: la libreria Math è risultata molto utile in svariate classi, tra cui il calcolo della distanza nel LevelHandler.
- *Stream e Lambda*: utilizzati in quantità per interfacce funzionali e in molte classi tra cui GameObject o GameEngine nel *debugger*.
- *Algoritmi*: realizzazione del movimento per il player, che impedisce movimenti "illegali" e collide con ostacoli. Gestione dello spawn nel LevelHandler basato sulla distanza dal player.

3.3.2 Thomas Capelli

- *JavaFx*: Uso di alcune funzionalità di base, senza scene builder, per creare il menù principale in modo più simile al metodo visto a lezione
- *Optional*: Utilizzo molto leggero di Optional soprattutto per il check delle collisioni dei bullet
- *Stream e Lambda*: Utilizzo di lambda e stream sempre per quanto riguarda le collisioni e in altri parti del codice per renderlo chiaro e coinciso
- *Algoritmi*: Un algoritmo interessante potrebbe essere quello pensato insieme a Lorenzo Cavallucci, e contenuto in LevelDesigner, per la gestione dello spwan di entità basato sul numero di livelli superati. Esso utilizza funzioni matematiche accompagnate da curve Gaussiane per il calcolo della probabilità. Un'altra classe che si è rivelata molto utile è stata TimerImpl che realizza un semplice ma efficace timer, distaccato dal Game Loop, che aspetta un unità di tempo a scelta. E' stato particolarmente utile per lo sviluppo dell'immortalità temporanea del giocatore e per il delay dello sparo.

3.3.3 Lorenzo Cavallucci

- *JavaFX* Studio della libreria **JavaFx**, più nello specifico per l'utilizzo di *Rectangle*, *Scene*, transizioni tra cui *FadeTransition*;
- *Stream e Lambda* Utilizzo di Stream e Lambda, soprattutto nella gestione della DoorComponent e il suo comportamento alla collisione col Player;
- *Algoritmi* Un algoritmo interessante potrebbe essere quello pensato insieme a Thomas Capelli, e contenuto in LevelDesigne, per la gestione dello spwan di entità basato sul numero di livelli superati. Esso utilizza funzioni matematiche accompagnate da curve Gaussiane per il calcolo della probabilità.

3.3.4 Ciprian Stricescu

- *JavaFX*: uso della libreria per lo sviluppo della classe GameUI, in particolare *Text* e *FadeTransition*.
- *Stream e Lambda*: utilizzo ogni qualvolta possibile, soprattutto nella gestione delle collisioni nelle componenti dei nemici e in *CollisionHandler*.
- *Optional*: utilizzo leggero nella gestione delle collisioni.
- *Algoritmi*: un utilizzo interessante è nella gestione del pattern di movimento di *DrunkComponent* basato su *Perlin noise generation*, trasposto in un ambiente 1d.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Francesco Magnani

Personalmente ho trovato in questo progetto una grossa sfida: fin dall'inizio ero convinto che avremmo, come gruppo, commesso molti errori e che il risultato finale non sarebbe stato esattamente ciò che ci aspettavamo di creare. Ho sempre considerato l'inesperienza, prima e dopo il lavoro svolto su questo progetto, come un gradino troppo ripido per creare qualcosa di veramente "solido". Nonostante tutto ciò, ritengo di aver conseguito insieme agli altri membri del gruppo un risultato di cui essere soddisfatti, che ha insegnato molto a ciascuno di noi circa le dinamiche del lavoro in gruppo, non solo nell'ambito della progettazione software. Da parte mia, non ritengo sia ancora il caso di lavorare troppo su un unico progetto: sono molto più aperto alla partecipazione a progetti di natura diversa rispetto a "Ryleh's Call", o Game developing in generale, ma non escludo di ritornare su questo lavoro in un futuro per ampliarlo e modificarlo.

4.1.2 Thomas Capelli

Giunto alla conclusione del progetto mi ritengo abbastanza soddisfatto del risultato in generale e della mia parte, specialmente considerando il fatto che sia il lavoro in team, sia la costruzione di un progetto di tale rilevanza erano due aspetti per me totalemente nuovi. Lavorare in questo nuovo modo è stato sicuramente di aiuto per crescere a livello personale per quanto riguarda le conoscenze e inoltre ha confermato, e anzi aumentato, la voglia di addentrarmi in questo mondo. L'impegno da parte mia e del team in generale c'è stato, ogni membro ha sempre cercato di dare il massimo ed essere presente nei momenti di bisogno; ovviamente l'inesperienza dovuta alla novità mi porta a dire che tante cose potrebbero essere migliorate e aggiunte in futuro ma, considerando anche il fatto che abbiamo

dovuto iniziare da capo a poche settimane dalla consegna a causa di una scelta sbagliata in fase di progettazione, mi ritengo personalmente soddisfatto da tutto ciò che abbiamo fatto per rimediare all'errore. Il lavoro con i miei colleghi è stato gratificante e mi ha spinto a prendere in cosiderazione l'idea di lavorare in futuro nuovamente su questo progetto per fare tutte le migliorie del caso.

4.1.3 Lorenzo Cavallucci

Una volta concluso il progetto, non posso che ammettere la mia positività nei confronti di ciò che ho appreso, sia da un punto di vista della programmazione, sia per quanto riguarda la mia persona: se da un lato, dopo diverse difficoltà, la realizzazione e la scrittura del codice mi ha portato a comprendere più a fondo il linguaggio utiizzato e le capacità anche creative che possono essere utilizzate, dall'altro è innegabile quanto importante sia stato tutto ciò per la crescita personale. Il lavorare in un team, rispettando deadline interne e magari interfacciandoci con gli altri colleghi in modo autonomo, ha reso il progetto una sorta di immersione in un mondo sicuramente più vicino all'ambito lavorativo. In più, difficile risulta non accennare anche i rapporti che si sono formati internamente al team, al punto da rendere quasi certo un ritorno su tale progetto, magari in un futuro, per continuare a lavorarci, con l'obiettivo, un giorno, di migliorarlo al punto da poterlo, eventualmente, pubblicare.

4.1.4 Ciprian Stricescu

A conclusione del progetto ritengo soddisfacente il nostro percorso, seppur con alti e bassi. La parte iniziale di macro-definizione del gioco è stata la parte più critica. Nelle parti finali ho decisamente notato la nostra crescita come team, temprati da un minimo di esperienza.
L'impresa di progettare un'applicazione di una certa difficoltà è stato un momento formativo che mi ha insegnato soprattutto quanto sia importante una corretta organizzazione del proprio tempo. Poter lavorare da vicino a parti diverse ritengo mi abbia dato la possibilità di approfondire tante nozioni di game design.
L'aver lavorato per la prima volta in team per un tempo considerevole mi ha fatto apprezzare l'esperienza a tal punto da voler continuare a lavorare a R'lyeh's Call o progetti futuri con i miei colleghi.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Francesco Magnani

Una delle difficoltà incontrate nello sviluppo di Ryleh's Call è stata un errore di calcolo da parte mia e del gruppo, quando si è scelto, almeno inizialmente, di utilizzare la libreria FXGL: essendoci "allargati" sulla raccomandazione circa l'utilizzo di librerie esterne, abbiamo scelto una libreria che non permetteva una solida progettazione e che forzava la dipendenza agli strumenti forniti da essa. Successivamente ad un confronto con i docenti, abbiamo quindi deciso di fare marcia indietro ed utilizzare un'altra libreria consigliata durante il corso, JavaFX, scartando del tutto FXGL e costringendoci a rivalutare tutto il progetto da capo, il che ha stretto notevolmente i tempi disponibili.
Nonostante questo, durante lo sviluppo di questo progetto ho trovato difficoltà principalmente nel costruire una visione di insieme che riuscisse a tenere conto in maniera efficace di tutti gli "ingranaggi" all'interno del progetto. Mentre ritengo che in buona parte questo sia stato dovuto a mancanza di esperienza nell'ambito di sviluppo e progettazione di software, giudico comunque scarsa l'attenzione che ho dedicato alle parti preliminari di design e in generale di massima astrazione. In compenso, questo progetto mi ha aiutato a comprenderne la vera importanza e di conseguenza il ruolo fondamentale che possiedono lungo lo sviluppo del prodotto finito.

4.2.2 Thomas Capelli

Penso che una delle difficoltà principali dell'intero gruppo sia stata la fase di progettazione iniziale, forse sottovalutata o forse più semplicemente per mancanza di esperienza ci siamo trovati "costretti" a svolgere diverse riunioni per capire come far funzionare il tutto. Inoltre sempre per i motivi elencati abbiamo sbagliato a considerare un framework preso in cosiderazione inizialmente che putroppo ci ha fatto perdere tempo che poteva essere dedicato diversamente al miglioramento del progetto.
Per quanto mi riguarda penso che il corso in generale sia stato molto interessante e utile, anche se alcuni aspetti piuttosto funzionali in fase di progetto sono stati purtroppo trattati davvero in fretta e furia a lezione e di conseguenza è stato necessario capire in autonomia e da capo il loro funzionamento (un esempio è lo strumento Git per quanto riguarda il lavoro in gruppo o Gradle per la gestione delle dipendenze).

4.2.3 Lorenzo Cavallucci

Nel corso dello sviluppo del progetto, non ho trovato vere difficoltà che possano aver impattato negativamente sul suo svolgimento nel tempo; in compenso, però, la parte iniziale dedicata a setup di progetto e progettazione concettuale, sono state più ostiche di quanto aspettato. In più, lo spostamento verso una nuovo FrameWork ha portato via tempo importante che avremmo volentieri speso in un miglioramento del progetto. Ciò nonostante, una volta risolti tutti i problemi del caso, devo ammettere di aver guadagnato sufficiente confidenza con diversi strumenti.

4.2.4 Ciprian Stricescu

La maggior difficoltà è stato di sicuro dover riscrivere l'intero progetto dopo una prima versione sbagliata. Tutto il percorso è stato un continuo trial and error ma grazie anche ai miei colleghi siamo riusciti a superare ogni ostacolo. Un'altra difficoltà che mi ha colpito personalmente è stata tutta la parte di algebra dei vettori e la sua implementazione in un linguaggio di programmazione. La confusione di un primo tentativo fallito alla fine si è tramutata in una sfida, oltre che formativa, divertente.

Appendice A

Guida utente

R'lyeh's Call è stato pensato fin dall'inizio come gioco user-friendly. Per guidare un nuovo giocatore, nella prima stanza che apparirà una volta premuto il tasto start vi è un semplice tutorial che offrirà tutte le informazioni necessarie.