Hide 'n Seek

Marco Sternini
Riccardo Fragozzi
Tiziano Vuksan
Marco Sangiorgi

25 Ottobre 2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il progetto consiste in un gioco di nome "Hide 'n Seek", in cui il giocatore deve raccogliere delle chiavi disseminate nella stanza per procedere alle stanze successive. All'interno della stanza vi è un mostro, il cui compito è quello di inseguire e ostacolare il giocatore. Sparsi nel mondo di gioco ci sono dei power-up che forniscono effetti al giocatore per aiutarlo nel suo compito. Il gioco finisce quando il giocatore riesce a raccogliere le chiavi dell'ultima stanza.

Requisiti funzionali

- Il giocatore si deve muovere in un ambiente bidimensionale in tempo reale.
- Il mostro/nemico deve essere guidato da un AI che riconosce gli ostacoli e si sa muovere all'interno del mondo di gioco.
- I power-up generati casualmente all'interno della mappa devono fornire potenziamenti temporanei al giocatore.
- Il gioco deve tenere traccia delle statistiche principali della sessione e queste devono essere consultabili dal menù principale.
- Durante una sessione di gioco, il giocatore può mettere in pausa e scegliere se tornare al menù principale o continuare la partita.
- Ogni mappa contiene uno specifico numero di chiavi che, una volta raccolte dal giocatore, stabiliscono il completamento del livello corrente.

Requisiti non funzionali

- Il gioco deve essere fluido e stabile anche su computer non troppo performanti.

1.2 Analisi e modello di dominio

Il mondo di gioco è composto da **entità** che interagiscono tra di loro e con l'ambiente. Le principali entità sono il **giocatore** e il **mostro**, ambedue possono muoversi e collidere con altre entità fisiche in gioco. Il giocatore sarà comandato da un input mentre il mostro da un'intelligenza artificiale che ha come obiettivo quello di inseguire il giocatore. Il mostro quando collide con il giocatore lo elimina, quindi fa terminare la partita. All'interno del mondo di gioco vi sono altre entità statiche che modellano l'ambiente. I **power-up** sono entità statiche che forniscono un potenziamento al giocatore una volta ottenuti, ovvero una volta colliso con essi. Le **chiavi** sono entità statiche collezionabili dal giocatore. Una volta ottenute tutte le chiavi presenti nel livello, questo risulta completato e si procede con il successivo. Infine i muri sono entità che delimitano le zone accessibili del mondo di gioco. Ogni livello è caratterizzato da una disposizione differente delle entità.

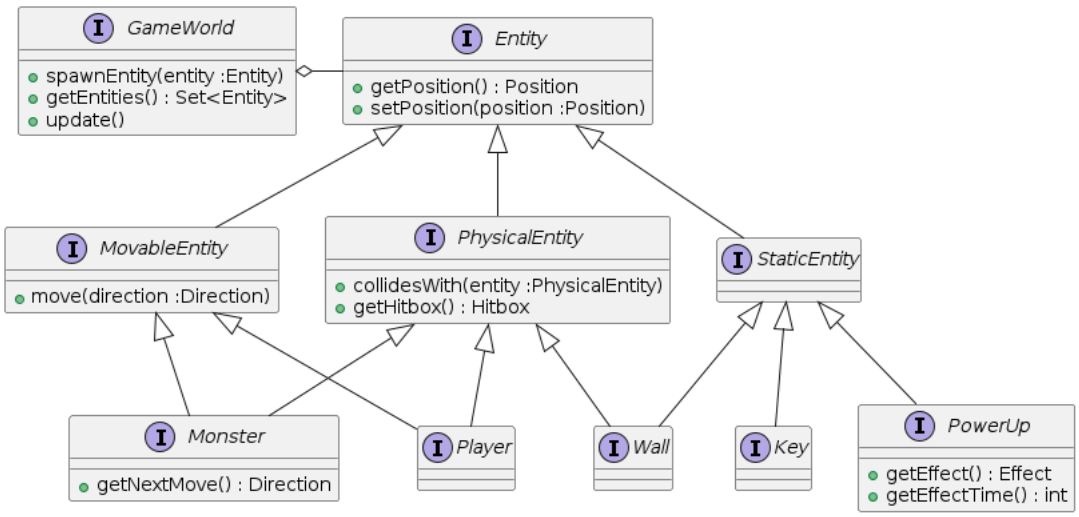

Figura 1.1: Schema UML del dominio

Capitolo 2

Design

2.1 Architettura

Abbiamo scelto di utilizzare il pattern architetturale MVC. L'interfaccia **GameSceneController** funziona da controller generale delle interfacce e da `entry-point` per l'applicazione. Il vero motore di gioco viene istanziato una volta che l'interfaccia GameGui viene caricata. **GameWorldController** gestisce l'avanzamento della logica del gioco attraverso il **GameWorld** e aggiorna la visualizzazione delle entità attraverso i relativi **EntityController**. Ogni EntityController viene istanziato associando un Entity (model) e un EntityView (view). Il Renderer si occupa successivamente di recuperare le view delle entity e gestire la loro rappresentazione.

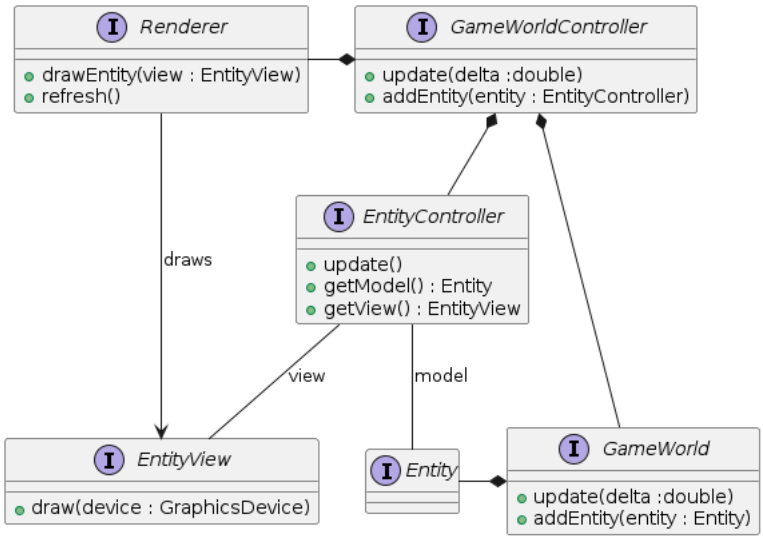

Figura 2.1: Schema UML MVC

Model

Il model segue il pattern architetturale ECS (Entity-Component-System), ma nella sua variazione semplificata **Entity-Component**. Ogni oggetto di gioco è un **Entity** alla quale vengono "attaccati" oggetti del tipo **Component** che ne definiscono il suo comportamento. L'architettura segue il principio di *composition over inheritance*, ogni entità è definita dai suoi componenti associati e non dalla sua gerarchia. L'interfaccia principale è **Entity** alla quale possono essere associati diversi **Component**. Per semplificare il pattern abbiamo scelto di mantenere un vincolo, ogni entità può possedere un unico componente della stessa tipologia.

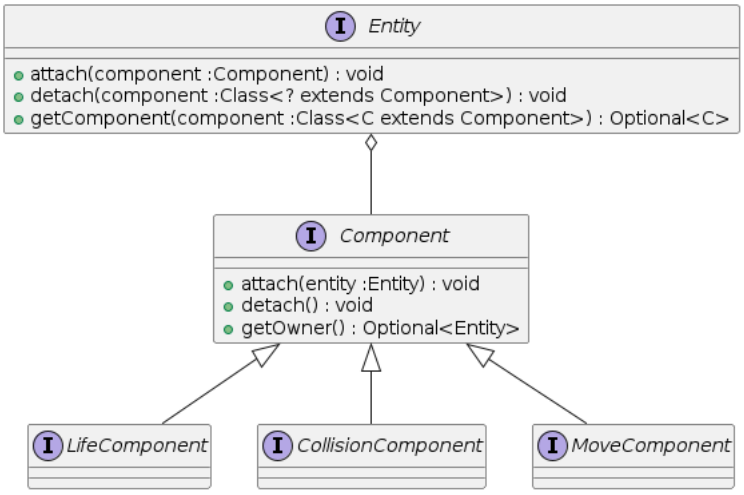

Figura 2.2: Schema UML Entity Component con componenti di esempio

2.2 Design dettagliato

2.2.1 Marco Sternini

La progettazione da me svolta riguarda principalmente:

- Gameloop e rendering.
- PowerUp.

Gameloop e rendering

Il core, o motore di gioco, si basa su un classico sistema con game loop. Il loop svolge continuamente una serie di processi che gestiscono l'avanza-

mento, le interazioni e il rendering della scena di gioco. L'interfaccia base è **GameLoop** che permette di gestire l'avvio e lo stop di esso. La classe astratta **GameLoopFXImpl** fornisce un metodo astratto che viene gestito ed eseguito 60 volte al secondo.

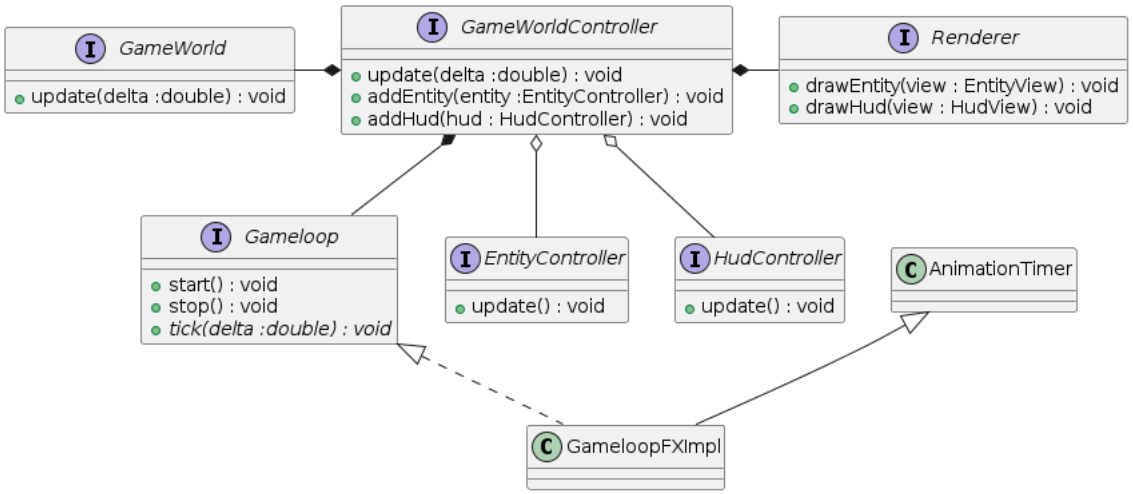

Figura 2.3: Schema UML Gameloop e GameWorld

Il flusso comincia passando al **GameWorld** l'input corrente. Il **GameWorld** viene poi aggiornato: viene assegnato un vettore alle entità comandate da input a seconda di esso, viene calcolato il vettore delle entità comandate da un AI e successivamente vengono mosse tutte le entità in gioco. Il controller aggiorna la view di ogni entità in gioco e fornisce al **Renderer** le view.

Problema: Mantenere la possibilità di cambiare componente grafica facilmente.

Soluzione: Per permettere, in ottica futura, di cambiare libreria grafica ho utilizzato il pattern **Strategy**. Il Renderer disegna la scena di gioco utilizzando un'interfaccia **GraphicsDevice** (la strategia) che viene assegnata. GraphicsDevice espone i principali metodi che le entità utilizzano per disegnarsi.

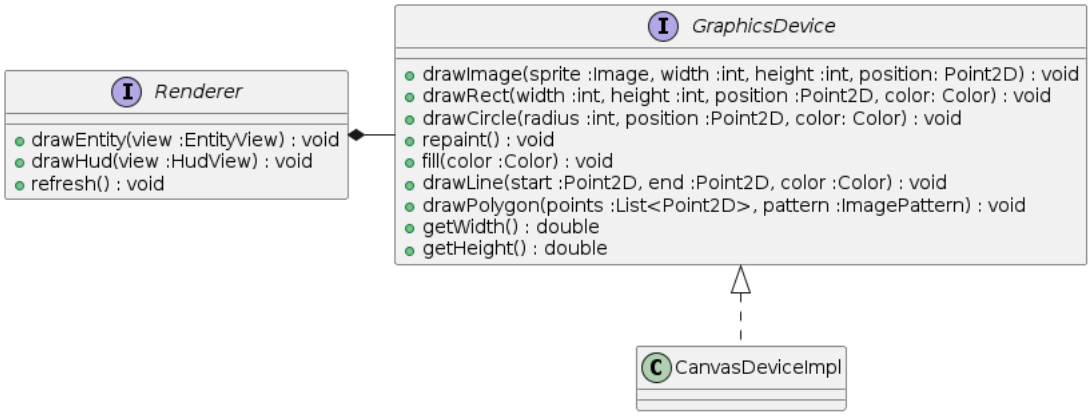

Figura 2.4: Schema UML Renderer

Power-up

I power-up non sono altro che entità di gioco che a contatto con un giocatore forniscono un effetto prolungato per un certo numero di secondi.

Problema: Il power-up deve costantemente ascoltare se un giocatore si trova nel proprio raggio di azione.

Soluzione: Per gestire l'interazione tra il giocatore e il Power-Up ho quindi introdotto i **Trigger** utilizzando il pattern **Observer**. I **Trigger** sono dei subscriber che restano in ascolto di eventi mandati da uno specifico componente osservabile. Gli **ObservableComponent** permettono di "iscrivere" più trigger. Ogni trigger ascolta una particolare tipologia di evento ed ha assegnato a sé stesso una funzione callback che viene eseguita alla ricezione dell'evento. Ho cercato di rendere questo sistema ad eventi il più generico possibile per essere utilizzabile anche dai miei colleghi per altre funzionalità. Estendendo **AbstractEvent** si può creare un evento specifico che fornisce informazioni aggiuntive riguardo quello che è accaduto. Il componente osservabile può quindi mandare questo evento e sollecitare i **Trigger** in ascolto ad eseguire il proprio callback.

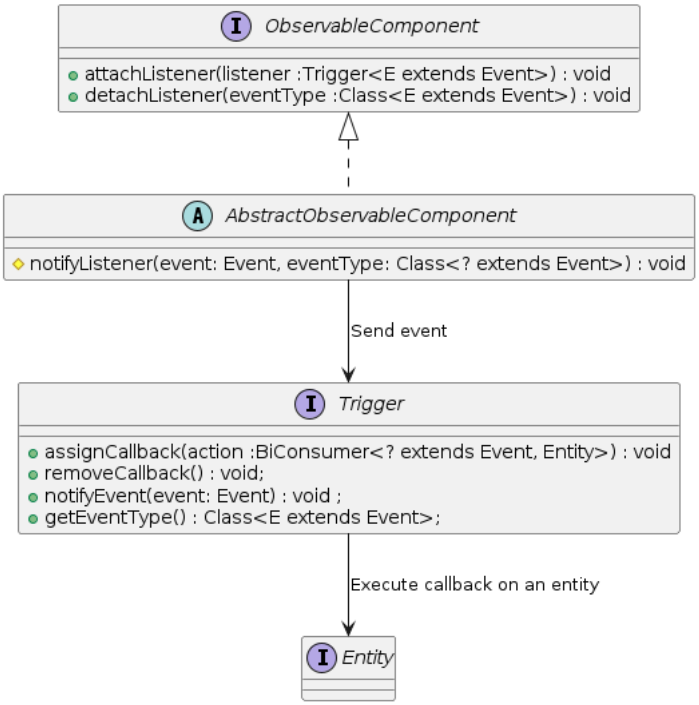

Figura 2.5: Schema UML dell'interazione fra ObservableComponent e Trigger

I power-up sono composti da un **LifeComponent**, **CollisionComponent** ed un **PositionComponent**. Il **Trigger**, che osserva il **CollisionComponent**, una volta sollecitato elimina il Power-up togliendo la vita ad esso, ed esegue il proprio effetto sull'entità con cui ha colliso. L'effetto modifica dei parametri di componenti, come per esempio la velocità nel **MoveComponent**.

Problema: Reset dei parametri di un componente modificato da un Power-up.

Soluzione: Una volta ottenuto un Power-up l'effetto viene immediatamente attivato e dopo un certo numero di secondi viene eliminato. Per ovviare a questo problema ho introdotto un'interfaccia **UpgradableComponent**. Quest'interfaccia permette al sistema di resettare i parametri del componente e controllare se questo è già stato alterato o no. In questo modo il giocatore non può raccogliere più Power-up dello stesso tipo per ottenere un effetto multiplo e rovinare l'esperienza di gioco.

2.2.2 Riccardo Fragozzi

Il mio contributo al progetto consiste nello sviluppo delle seguenti tematiche:

- Gestione del movimento
- Gestione delle collisioni
- Statistiche di gioco
- Creazione dei livelli e disegno dei principali asset di gioco

Gestione del movimento

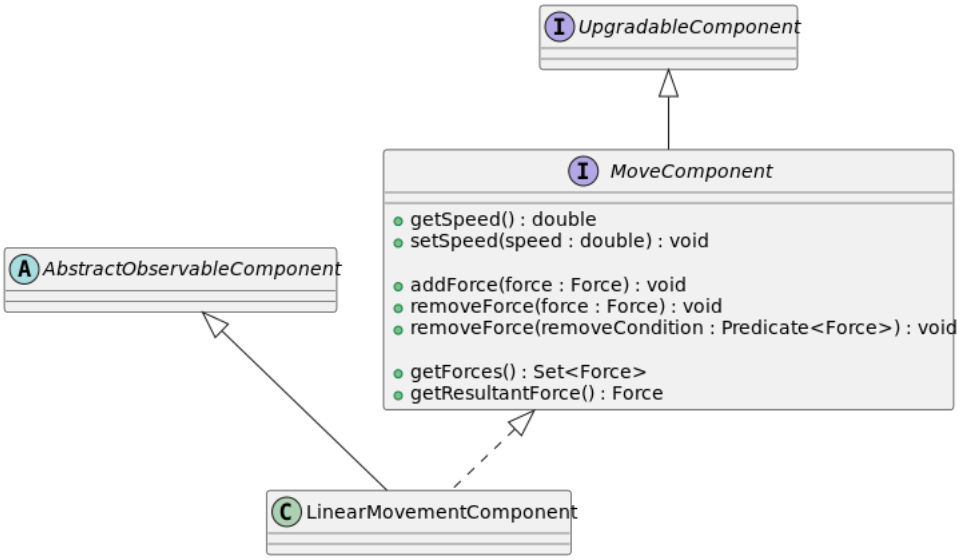

Figura 2.6: Schema del movimento

L'area di gioco è composta da varie entità, alle quali sono assegnati vari componenti in funzione del loro ruolo attribuito nel gioco. Alle entità che devono muoversi all'interno dell'area di gioco viene aggiunto il componente **MoveComponent**, che espone i metodi necessari per determinare univocamente la direzione e la velocità istantanea dell'entità in movimento. L'interfaccia **MoveComponent** è implementata unicamente dalla classe **LinearMovementComponent** che, come spiegato di seguito, produce un movimento privo di accelerazione. In ottica futura è possibile creare l'effetto dell'accelerazione inserendo nel metodo **getResultantForce** un moltiplicatore proporzionale a quanto tempo prima è stata applicata la forza.

Problema: In virtù della progettazione dei muri (e delle entità in generale) sottoforma di poligoni, è indispensabile gestire in modo efficiente i vincoli del movimento dovuti alla collisione, in quanto si potrebbero riscontrare casi in cui il muro non ha un'inclinazione di 0° oppure 90°, ma può assumere qualsiasi angolazione.

Soluzione: Ho pensato di modellare il movimento attraverso l'aggiunta di una o più Forze all'entità, ognuna delle quali ha un'intensità e una direzione: in questo modo è possibile gestire in modo più realistico una collisione che vincola il corpo in una determinata direzione, lasciando libero il movimento in tutte le altre direzioni non vincolate.

Per la semplificazione dello sviluppo ho progettato un motore della fisica utopico, sviluppato nella classe **DinamicsWorldImpl**, nel quale le forze non producono un'accelerazione ma un movimento lineare. Il prodotto delle forze fa sì che la posizione dell'entità venga aggiornata, tramite il componente **PositionComponent**. Ogni forza può essere scomposta nelle due componenti X e Y, e ha un identificatore per permettere di essere trovata ed eliminata facilmente quando non è più in uso. Esempio: se l'utente sta premendo contemporaneamente W e D, al player vengono aggiunte due forze perpendicolari distinte di uguale intensità. Se il player trova un muro in alto che ne vincola il movimento, la forza che muove il giocatore in alto viene annullata dall'effetto della collisione, mentre quella verso destra continua a produrre un effetto di movimento verso destra.

Gestione delle collisioni

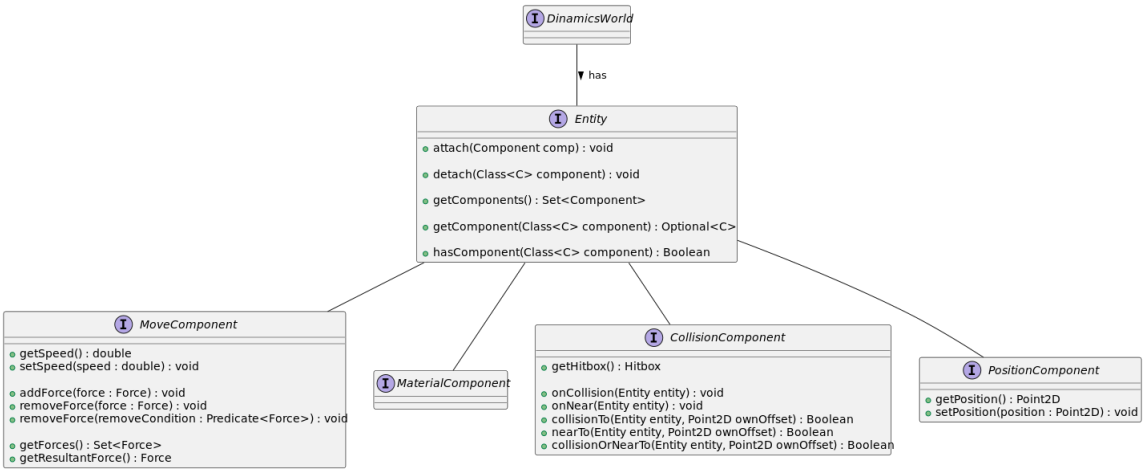

Figura 2.7: Schema di rilevamento delle collisioni

Ad ogni entità può venire assegnato un componente **CollisionComponent**, che dispone di una **Hitbox** (ovvero un Set di Point2D che modellano il poligono di collisione) e fornisce i metodi **collisionTo** e **nearTo** illustrati di seguito. Se due componenti che si scontrano hanno entrambi un **MaterialComponent**, vengono gestiti come oggetti aventi una massa e il motore del gioco ne rende impossibile l'intersezione (ad esempio il player ed il muro). Allo stesso modo, se almeno uno dei due oggetti non ha il **MaterialComponent**, l'intersezione è possibile (ad esempio il player ed un PowerUp). Il componente **MaterialComponent** attualmente non presenta metodi ed è utile solamente a fornire la propria istanza. In futuro si potrebbe facilmente introdurre ad esempio la rottura del muro in funzione della velocità del giocatore e della resistenza del materiale.

Problema: Proprio perchè il motore del gioco rende impossibile la collisione tra due entità materiali, la collisione tra il player ed il monster era impossibile e l'evento non veniva mai scatenato.

Soluzione: Per ovviare questo problema ho introdotto l'evento **nearTo**: esso si verifica quando due oggetti sono adiacenti ma non intersecati (ad esempio il player appoggiato al muro), che è distinto dall'evento **collisionTo**, il quale si verifica quando due oggetti sono in intersezione. Ho introdotto

inoltre **collisionOrNearTo**, che si verifica in condizione or tra i due eventi.

Il **DinamicsWorld**, per ogni entità, ottiene il vettore di forza risultante tramite il **MoveComponent**, controlla che abbia un **MaterialComponent**, interroga il **CollisionComponent** per assicurarsi che a seguito del movimento non si verifichino intersezioni con altre entità e, in caso negativo, aggiorna la posizione al componente.

Statistiche di gioco

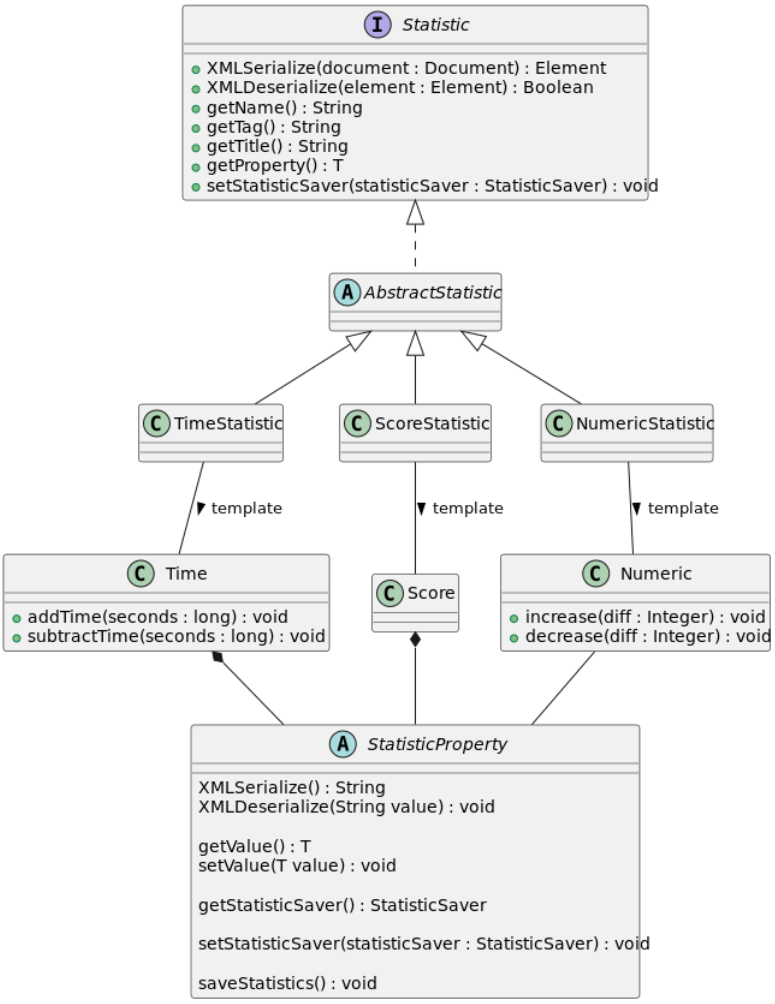

Figura 2.8: Relazione tra il componente Statistic e la proprietà StatisticProperty

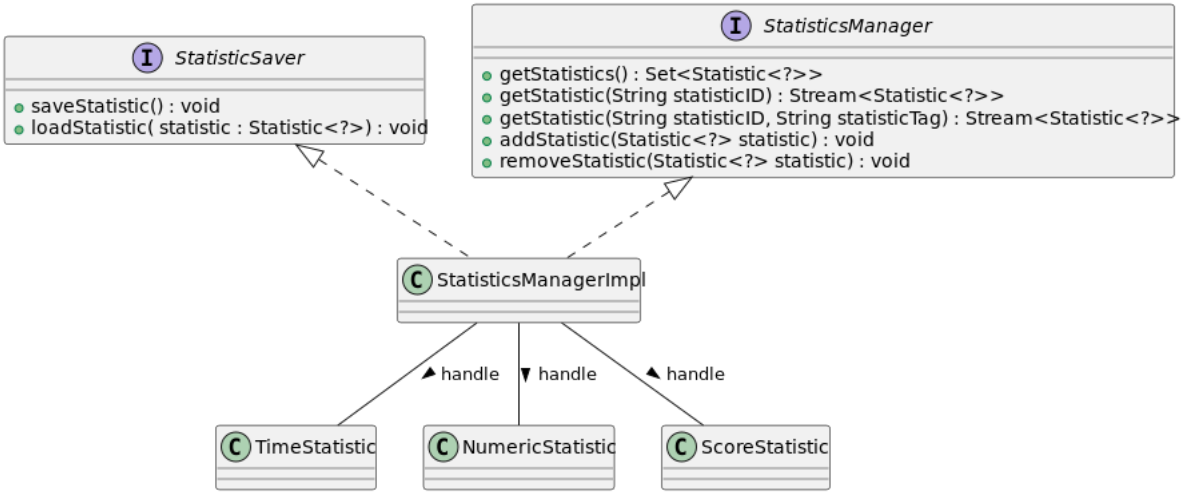

Figura 2.9: Relazione tra StatisticManagerImpl e le statistiche

Problema: Per realizzare un gestore delle statistiche, è necessario gestire gli eventi di superamento del livello, perdita del livello, raccoglimento di una chiave, osservare il tempo di superamento di ogni livello, e salvare queste informazioni su disco.

Soluzione: Il salvataggio e la lettura da disco sono state realizzate tramite l'utilizzo del parsing XML, per creare facilmente una struttura dati adeguata. Ogni statistica è identificata da **name** e **tag**, che indicano rispettivamente il nome e il livello alla quale si riferiscono (ad esempio `bestScore` e `level1`). Ogni statistica è composta da un valore, modellato da una classe che estende **StatisticProperty**: quest'ultima contiene le funzioni per il parsing e il salvataggio XML e i getter/setter del valore, che utilizzano un **StatisticSaver** per scrivere su disco ogni qualvolta il valore viene modificato.

Esistono tre tipi di **StatisticProperty**: **Numeric**, **Score** e **Time**. Il componente **StatisticManager** gestisce tutte le statistiche e fornisce i metodi per crearle, rimuoverle e filtrarle in modo semplice. A livello di view, ho previsto due soli componenti: **ScoreStatisticView**, un Canvas che riempie le 5 stelle dello score in funzione del punteggio, e **TextStatisticView**, che estende il componente Label per mostrare le statistiche di tipo **Numeric** e **Time**.

Creazione dei livelli e disegno dei principali asset di gioco

I livelli sono composti da entità prefissate in determinati punti, di cui i muri sono caratterizzati dal poligono di delimitazione. Poichè ogni muro è caratterizzato da una posizione e un set di vertici, la creazione manuale dei livelli prevede la scrittura manuale di tutti i punti, ed è un lavoro oneroso. Per questo mi sono occupato dello sviluppo di un tool separato che permette di disegnare i livelli col trascinamento del mouse e semplifica l'esportazione del file di livello in formato xml, e della conseguente creazione dei quattro livelli di gioco.

2.2.3 Tiziano Vuksan

Il lavoro svolto da me riguarda:

- Menù principale e di pausa
- Gestione delle mappe

Menù principale e di pausa

Questa sezione comprende la realizzazione di ogni singolo menù visualizzato dall'utente. Tutto si concentra sull'interfaccia **GameSceneController** che viene implementata dalla sua relativa classe **GameSceneControllerImpl**. Questa classe, oltre a essere l'unico oggetto istanziato all'avvio del programma (attraverso il metodo start di **Launcher**), permette di navigare tra le diverse interfacce grafiche. Per poter mostrare a video una interfaccia grafica diversa ogni volta mi sono avvalso di **SceneManager** che mi permette di tenere in memoria ogni singola interfaccia. **SceneManager** quindi espone un metodo (`activateInterface()`) che cambia il contesto - quindi l'oggetto `root` principale - della scena che viene disegnata nella finestra del programma. Ad ogni interfaccia grafica è associato un suo controller a cui vengono delegate le azioni da svolgere nel caso in cui l'utente interagisce con elementi di tipo `"control"`, in questo caso pulsanti. Inoltre è presente un riferimento all'oggetto **GameSceneControllerImpl** che permette di poter passare ad altre interfacce direttamente. Quello che si ritroverà l'utente all'avvio del programma sarà il menu principale.

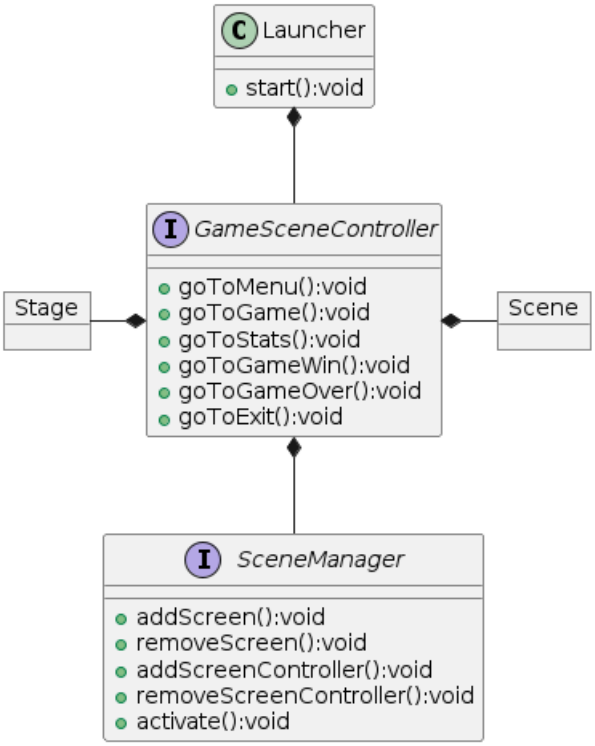

Figura 2.10: Schema UML GameSceneController e SceneManager

Problema: Cambio di interfacce grafiche. L'idea iniziale era stata quella di cambiare **Scene** allo **Stage** (finestra) principale, e quindi di caricare sul momento ogni singola interfaccia. La cosa non si è prestata favorevole per diversi motivi quali anche la performance.

Soluzione: Le UI vengono caricate tutte all'inizio del programma. Oltre a queste vengono caricate, e quindi associate a loro, anche un file di styling CSS. Ho creato un manager di scene (**SceneManager**), o più nello specifico un manager di oggetti `root` (ovvero i contenitori principali di ogni file `.fxml`). Questo è servito perché ho usato una sola scena alla quale viene cambiato contesto impostando un oggetto `root` diverso ogni volta. Il manager associa a ogni nome di file univoco il suo oggetto `root`, e al momento di cambio d' interfaccia viene "attivata" l'interfaccia desiderata.

Gestione delle mappe

Il gioco, come menzionato prima nel paragrafo 1.2, consiste in diversi li-

velli, che possono essere creati a partire da dei file in formato XML. Dentro a questi file ci sono diversi elementi che identificano le entità presenti in un livello di gioco come per esempio i muri, il giocatore e il mostro. **GameLevel** permette di accedere a informazioni importanti di ogni livello come il suo nome, identificativo, le entità in gioco etc. Queste informazioni sono rese disponibili da **GameLevelLoader**. Per poter quindi leggere le informazioni da file e tradurle, istanziando i rispettivi elementi, mi sono avvalso di quest'ultima classe. Da qui avviene la lettura e parsing dei singoli livelli. Inoltre **GameLevelLoader** espone a **GameLevel** metodi che restituiscono le entità e, nello specifico, anche gli HUD di gioco.

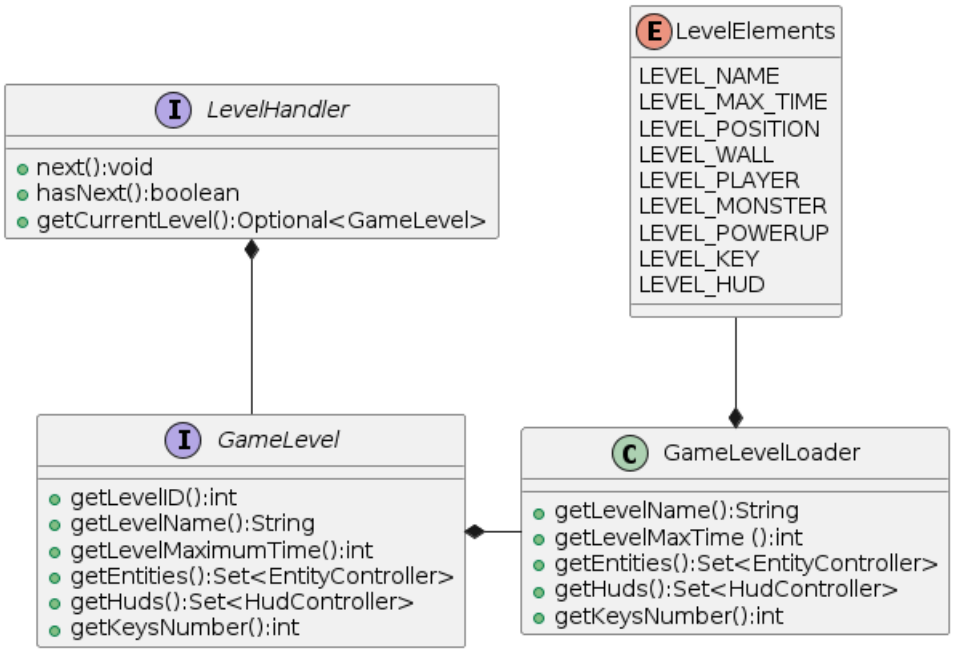

Figura 2.11: Schema UML GameLevel, LevelHandler, GameLevelLoader

Problema: Serviva uno standard per poter definire le entità e gli HUD di gioco all'interno di un file

Soluzione: Ho usato l' XML perché mi sembrava il più adatto e flessibile. Ogni file che rappresenta un livello ha degli elementi strettamente legati all'entità che vengono usate nel motore di gioco. Tra questi vi sono (entità): `wall`, `player`, `monster`, `powerup`, `key`, `hud`. È quindi il compito di **GameLevelLoader** di leggere da file questi elementi e tradurli in oggetti.

2.2.4 Marco Sangiorgi

La progettazione da me svolta riguarda principalmente:

- AI di gioco.
- Sistema di illuminazione.

AI di gioco

L'AI di gioco consiste in un componente in grado di sfruttare altre componenti per conoscere lo stato attuale di gioco e reagire di conseguenza. Tale componente è modellato dall'interfaccia **BrainComponent** che espone un metodo per attivare il processo di analisi e conseguente reazione.

Problema Riconoscere lo stato di gioco, è necessario poter reperire le entità che sono in gioco.

Soluzione L'idea è quella di creare un componente in grado di comportarsi come i sensi umani, il cui compito è di interrogare le entità circostanti per crearsi un'idea attraverso le caratteristiche del senso stesso (vista, udito, ...).
L'interfaccia base è **SenseComponent** ed espone un metodo (**World()**) per reperire l'insieme di entità che finora sono state 'percepite' attraverso il metodo che le interroga (**Feel(Entity)**).
Il pattern **Strategy** è stato utilizzato per modellare la logica generale di creazione del mondo 'percepito' nella classe astratta **AbstractSenseComponent**, esponendo il metodo principale per verificare i requisiti di ogni entità interrogata.
Sempre per avvicinarsi all'idea di senso umano, l'enumerazione **SenseConfidence** è stata introdotta per dare la possibilità di gestire diversi livelli di certezza di quanto percepito riguardo un'entità.

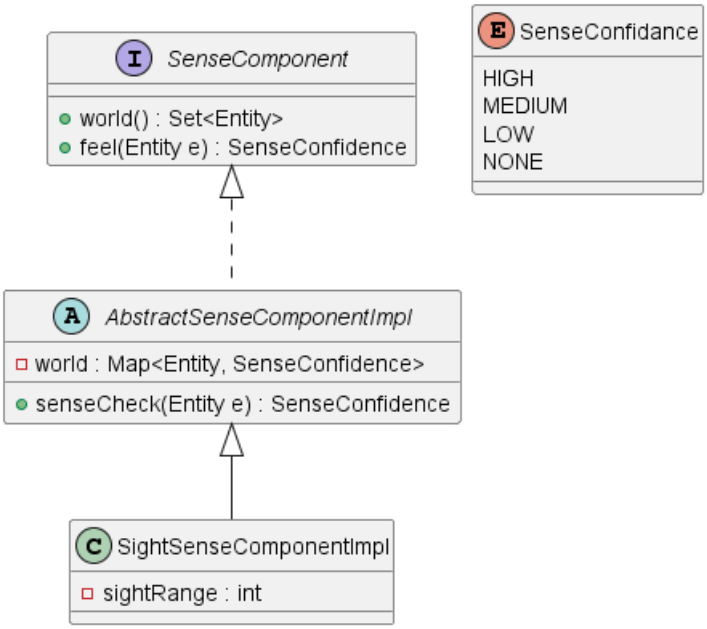

Figura 2.12: SenseComponent Schema

Problema Gestire i sensi di tutte le entità in gioco.

Soluzione Per suddividere la gestione delle diverse caratteristiche di ogni entità, e quindi anche della componente sensoriale, è stato introdotto una generalizzazione del concetto di mondo.
Partendo dall'idea di **World** come insieme di oggetti, specializzandolo ad un **EntityWorld** insieme di Entity, fino ai più capillari:

- **GameWorldImpl**, per gestire tutte le entità di gioco
- **SenseWorldImpl**, per gestire la parte sensiorale e intellettiva delle entità di gioco
- **DynamicsWorldImpl**, per gestire la fisica delle entità di gioco

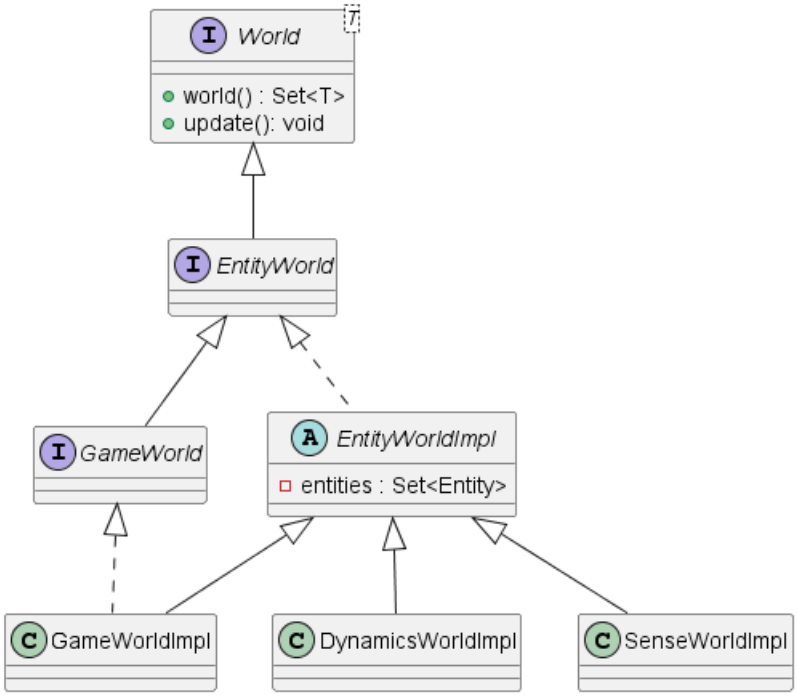

Figura 2.13: Worlds Schema

Problema Capire se un'entità è buona o cattiva

Soluzione Per differenziare le entità sul loro ruolo nel gioco è stata introdotta la caratteristica del 'cuore'. L'interfaccia base è **HeartComponent**, in grado di essere interrogato per capire cosa pensa di un altra entità.

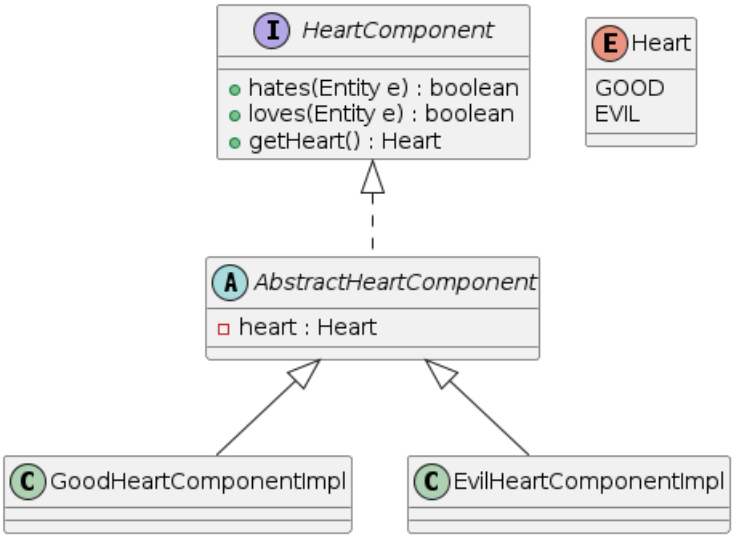

Figura 2.14: HeartComponent Schema

Problema Algoritmi di Path-Finding per evitare gli ostacoli

Soluzione L'idea è quella di simulare un moderno GPS su una mappa che viene espansa incrementalmente man mano che si scopre il mondo di gioco. L'interfaccia che modella questa idea è **MapComponent**, in grado di crearsi una mappatura interna di tutte le entità e capace di ritornare il percoso minore tra due punti (se esiste) quando interrogata.
L'implementazione **GPSMapComponentImpl** fa uso dell'algoritmo **Floyd-Warshall** per avere il percorso minore tra tutte le coppie di punti percorribili della mappa.
A tale scopo è stata utilizzata la libreria esterna **JGraphT** per delegare la parte algoritmica sui grafi.

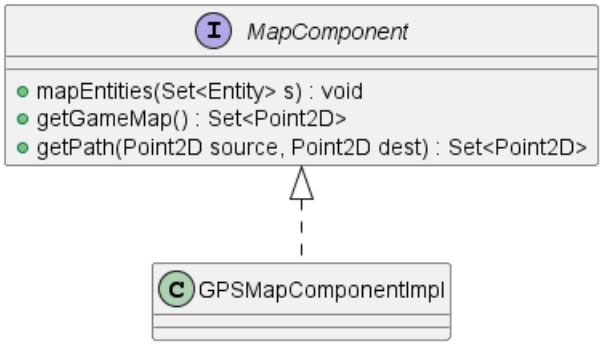

Figura 2.15: MapComponent Schema

Problema Analisi di gioco e conseguente reazione

Soluzione Come il nostro cervello utilizza ciò che ha a disposizione per interagire col mondo e reagire, allo stesso modo è necessario un componente che sappia sfruttare gli altri componenti. Questa idea è stata dall'interfaccia **BrainComponent**, classe core dell'AI, in grado di saper sfruttare gli altri componenti in modo più (**ExpertBrainComponentImpl**) o meno (**NaiveBrainComponentImpl**) furbo.
Sempre sull'onda di ricalcare la realtà, il comportamento che viene adottato dal BrainComponent dipende dal HeartComponent dell'entità che lo possiede. Pertanto, è stata adottata una mappatura di comportamenti diversificata sulla base del 'cuore'.
Il pattern **Strategy** è stato utilizzato per rendere trasparente la gestione delle mappature tramite la classe astratta **AbstractBrainComponent**.

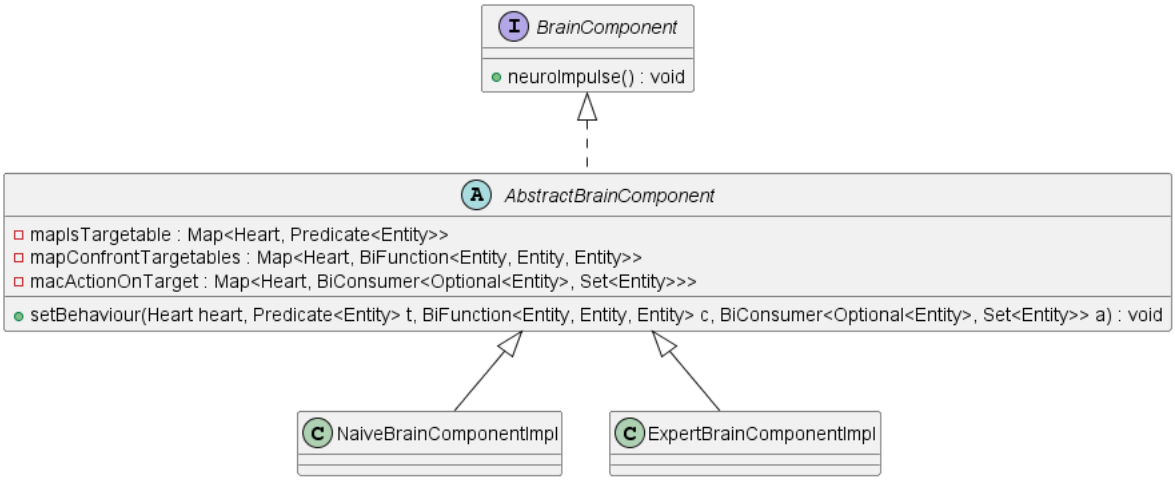

Figura 2.16: BrainComponent schema

Sistema di illuminazione
L'illuminazione del player è una delle cataratteristiche chiave del gioco.

Problema Rendering del punto di luce

Soluzione Si è scelto di utilizzare un'immagine .png con sfumatura al centro che venisse spostata ogni frame con il player.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per mantenere sotto controllo il funzionamento di diverse parti del progetto, abbiamo realizzato specifici test utilizzando la suite JUnit. I test sono per la maggioranza relativi a componenti fondamentali dell'architettura. Sono presenti:

- Una classe di test che controlla il funzionamento dell'architettura Entity-Components.
- Due classi di test per controllare il funzionamento dell'input sia dalla parte view che dal componente.
- Una classe di test per le collisioni nel mondo di gioco.
- Una classe di test per i Trigger e il sistema ad eventi.
- Due classi di test per verificare il corretto comportamento dell'intelligenza artificiale.
- Una classe di test per controllare i requisiti minimi di un livello.

3.2 Metodologia di lavoro

Per realizzare il progetto abbiamo utilizzato Git e GitHub per l'hosting in rete. Il branch *main* creato automaticamente da GitHub non lo abbiamo utilizzato particolarmente perché non vi era la necessità di creare più release. In generale, il branch principale era *develop* dove abbiamo cercato comunque di mantenere sempre versioni stabili, ovvero eseguibili. Ogni membro del team

creava un feature-branch specifico e proseguiva il suo sviluppo all'interno di esso. Una volta ritenuta pronta la funzionalità, si procedeva con un merge su *develop*. A volte abbiamo anche creato dei commit direttamente su *develop* per sistemare piccoli bug o aggiungere funzionalità che richiedevano poche righe di codice.

3.2.1 Marco Sternini

- Gameloop, renderer e relative implementazioni (**controller.GameloopFXImpl, view.RendererImpl**);
- Realizzazione di interfaccia a un generico device per la rappresentazione grafica e implementazione specifica per il canvas (**view.CanvasDeviceImpl**);
- Realizzazione di due classi astratte per i componenti (**model.components.AbstractComponent**, **model.components.AbstractObservableComponent**);
- Creazione di diversi componenti utilizzati da varie entità (**model.components.InputHandlerComponentImpl**, **model.components.InventoryComponentImpl**, **model.components.lives.LifeComponentImpl**);
- Progettazione ed implementazione dell'Hud sia view che controller (**view.huds.HudViewImpl**, **controller.huds.KeyHudControllerImpl**);
- Realizzazione di un oggetto per gestire l'input da tastiera (**controller.InputSchemeImpl**);
- Progettazione e realizzazione del sistema ad eventi (**model.TriggerImpl**, **model.events.***);
- Implementazione dei PowerUp (**model.entities.PowerUp**, **view.entities.PowerUpViewImpl**) e delle tipologie/effetti (**model.enums.PowerUpTypes**);
- Realizzazione di una classe astratta generica per gli oggetti collezionabili (**model.entities.AbstractCollectableEntity**) e la sua implementazione principale (**model.entities.Key**);
- **InputSchemeTest**, **EntityTest**, **InputHandlerTest**, **TriggerTest**;

Ho contribuito anche alla realizzazione di:

- Controller generici per entità (**controller.entities.***);
- Controller principale del mondo di gioco (**controller.GameWorldControllerImpl**);
- View generiche per le entità (**view.entities.AbstractEntityMovableView**, **view.entities.AbstractEntityView**);

3.2.2 Riccardo Fragozzi

- Sviluppo di un componente che gestisce le collisioni (**hidenseek.model.components.physics.CollisionComponent** e l'implementazione **CollisionComponentImpl**)
- Sviluppo di un componente che gestisce il movimento (**hidenseek.model.components.physics.MovementComponent** e l'implementazione **MovementComponentImpl**)
- Sviluppo di un componente che gestisce il materiale (**hidenseek.model.components.physics.MaterialComponent** e l'implementazione **MaterialComponentImpl**)
- Sviluppo di un componente che gestisce la posizione (**hidenseek.model.components.physics.PositionComponent** e l'implementazione **PositionComponentImpl**)
- Sviluppo di una classe che modella un segmento che compone un lato di una Hitbox (**hidenseek.model.components.physics.Segment** e l'implementazione **SegmentImpl**)
- Sviluppo di una classe che modella una forza applicabile ad un entity (**hidenseek.model.components.physics.Force** e l'implementazione **ForceImpl**)
- Sviluppo di una classe che modella una corpo avente una forma poligonale (**hidenseek.model.components.physics.Hitbox** e l'implementazione **HitboxImpl**)
- Sviluppo di una classe che modella il motore della fisica, determinando gli spostamenti causati dalle forze e le collisioni (**hidenseek.model.worlds.DinamicsWorld**)

- Sviluppo del controller (**hidenseek.controller.StatisticsController** associato all'intefaccia **GameStatsGui.fxml**)
- Sviluppo del model del sistema delle statistiche **hidenseek.model.statistic.***, **hidenseek.model.statistic.numeric***, **hidenseek.model.statistic.score***, **hidenseek.model.statistic.time***,
- Sviluppo della view del sistema delle statistiche **hidenseek.view.statistics.***,

Ho contribuito allo sviluppo delle seguenti classi per le parti riguardanti l'uso delle statistiche:

- **hidenseek.controller.GameSceneControllerImpl**,
- **hidenseek.controller.GameWorldControllerImpl**,

3.2.3 Tiziano Vuksan

- Realizzazione di un controller per la navigazione tra interfacce (**hidenseek.controller.GameSceneController** e relativa implementazione **hidenseek.controller.GameSceneControllerImpl**)
- Realizzazione di un oggetto per la gestione delle interfacce (**hidenseek.model.GameSceneController** e relativa implementazione **hidenseek.model.GameSceneControllerImpl**)
- Creazione di tutte le interfacce grafiche in formato FXML (proprietario della libreria JavaFX): **layouts.GameGui**, **layouts.GameOverGui**, **layouts.GameStatsGui**, **layouts.GameWinGui**, **layouts.MainMenuGui**
- Creazione della maggior parte dei controller associati alle interfacce: **hidenseek.controller.fxml.GameGuiController**, **hidenseek.controller.fxml.GameOverMenuController**, **hidenseek.controller.fxml.GameWinController**, **hidenseek.controller.fxml.MainMenuController**

- Creazione del file CSS per lo styling di tutti i menu (**stylesheets.MenuStyle**)

  N.B. : Ho creato un'unico file CSS per lo styling di tutti i menu. Non era necessario crearne uno per interfaccia, ma nel caso servisse basterebbe una piccola modifica nella parte di caricamento delle interfacce dentro alla classe **hidenseek.controller.GameSceneControllerImpl**

- Creazione di un oggetto che legge un file di livello in formato xml **hidenseek.model.GameLevelLoader**

- Creazione di un oggetto che rappresenta un singolo livello (**hidenseek.model.GameLevel** e sua implementazione **hidenseek.model.GameLevelImpl**)

- Un gestore di livelli di gioco (**hidenseek.model.LevelHandler** e sua implementazione **hidenseek.model.LevelHandlerImpl**);

3.2.4 Marco Sangiorgi

- Interfaccia ed implementazione base per i mondi (**hidenseek.model.worlds.AbstractEntityWorldImpl**, **hidenseek.model.worlds.World**)

- Gestione della parte AI (**hidenseek.model.components.brains.AbstractBrainComponent**, **hidenseek.model.components.brains.NaiveBrainComponentImpl**, **hidenseek.model.components.brains.ExpertBrainComponentImpl**)

- Algoritmi di pathfinding (**hidenseek.model.components.MapComponentImpl**)

- Componenti vari utilizzati da altre entità (**hidenseek.model.components.RewardCompoenentImpl**)

- Identificazione delle entità di gioco secondo il loro scopo nel gioco (**hidenseek.model.components.hearts.AbstractHeartComponent**, **hidenseek.model.components.hearts.GoodHeartComponentImpl**, **hidenseek.model.components.hearts.EvilHeartComponentImpl)**

- Gestione delle percezione delle altre entità (**hidenseek.model.components.senses.AbstractSenseComponent**, **hidenseek.model.components.senses.SightSenseComponentImpl**)

- Generalizzazione dell'idea di insieme di entità omogeneo (**hidenseek.model.wordls.World**, **hidenseek.model.wordls.EntityWorld**, **hidenseek.model.wordls.SenseWorldImpl**)

- Creazione di una piccola libreria di utility usata da alcune classi (**hidenseek.model.Utils**)
- Illuminazione di gioco (**hidenseek.model.components.physicsLightComponent**)

Ho contribuito anche alla realizzazione di:

- **GameWorldControllerImpl**
- **DynamicsWorldImpl**
- **GameWorldImpl**
- **Player, Monster**
- **PlayerViewImpl**

3.3 Note di sviluppo

3.3.1 Marco Sternini

- Optional;
- Lambda;
- Stream;
- Reflection;
- Generici;
- Wildcards;
- JavaFX;

Per la progettazione e sviluppo generale dell'architettura ECS ho seguito un progetto di anni scorsi (OOP17-ga-game). In particolare l'interfaccia **Entity** che è il core del sistema ha funzionato da base di partenza per poi sviluppare la nostra architettura.

3.3.2 Riccardo Fragozzi

- Stream;
- Reflection;
- Generics;
- Wildcards;
- Lambda;
- Optional;
- JavaFX;
- API **org.w3c.dom** per la serializzazione e deserializzazione in formato XML.
- AppDirs per il salvataggio delle statistiche cross-platform

Mi sono occupato inoltre del design dei principali pattern di gioco (texture di sfondo, asset del player, maschere di opacità del LightComponent e **ScoreStatisticView**), utilizzando la suite Adobe Illustrator e Adobe Photoshop.

3.3.3 Tiziano Vuksan

- Stream;
- Lambda;
- Optional;
- JavaFX;
- API **org.w3c.dom** per il DOM (parsing di file XML).
- FXML per la creazione di interfacce grafiche

Per la progettazione generale delle interfacce grafiche ho usato SceneBuilder, un tool che mi permette di strutturare le interfacce, per poi dare come risultato un file in formato FXML.
In genere per l'utilizzo corretto delle classi e interfacce della libreria JavaFx mi sono riferito alle documentazioni ufficiali (sezione "Documentation").

3.3.4 Marco Sangiorgi

- Stream;
- Lambda;
- Optional;
- JavaFX;
- API **org.jgrapht:jgrapht-core** per il pathfinding;

La documentazione consultata per l'implementazione degli algoritmi sui grafi è la seguente: JGraphT link

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Marco Sternini

Mi ritengo abbastanza soddisfatto del lavoro da me svolto. Credo che complessivamente il progetto possa essere utile come mockup per la realizzazione del gioco vero e proprio, magari con un motore di gioco esterno o utilizzando librerie grafiche avanzate. Buona parte del codice, sia mio che degli altri membri del gruppo, ritengo sia strutturato bene in modo che sia facilmente estendibile. L'aspetto del quale non sono pienamente soddisfatto è l'utilizzo del pattern MVC. Solo durante le fasi finali ho notato certe scelte progettuali discutibili, ma per essere la prima volta lo reputo comunque un discreto risultato. Nel mio attuale lavoro da programmatore sviluppo piccole parti di un framework molto grande e ben testato. Per questo motivo le fasi di progettazione non risultano troppo complesse o prolungate. Questo progetto rappresenta la mia prima esperienza di sviluppo completo, dal brainstorming fino al rilascio. Lo svolgimento di questa prova mi ha inoltre fatto comprendere quanto sia importante lo unit testing e l'utilizzo dei pattern. Questi aspetti non vengono per niente citati negli istituti tecnici informatici e la trovo una grande mancanza.

Riccardo Fragozzi

Sono soddisfatto del lavoro svolto, nel quale ho messo in pratica tutte le mie conoscenze e ne ho acquisite alcune nuove, come ad esempio la capacità di sviluppare in gruppo e ideare il gioco in brainstorming. Questo può essere considerato un importante obiettivo che è stato raggiunto sia da me personalmente che da tutti gli altri componenti del gruppo. Anche il successo

riscontrato nel coordinare il poco tempo a disposizione a causa di vari impegni lavorativi ha dimostrato una buona capacità organizzativa da parte di tutti noi, e di questo mi ritengo pienamente soddisfatto.

Tiziano Vuksan

Per quanto riguarda il lavoro da me eseguito sono abbastanza soddisfatto. Dico "abbastanza" perché sono sicuro che il codice che ho scritto si può sicuramente migliorare. Personalmente non sono mai soddisfatto sempre al 100% perché sono dell'opinione che su qualsiasi cosa c'è sempre margine per un miglioramento. Premetto che i membri di questo gruppo (tutti e quattro) sono studenti lavoratori (tra l'altro dipendenti della stessa azienda). Questo ovviamente non è da intendersi come giustificazione per incoerenze tra i lavori assegnati ed eventuali noncuranze nella qualità della progettazione e sviluppo del programma (il tempo è stato vitale per la realizzazione di questo progetto). Detto ciò, per motivi di lavoro improrogabili, siamo stati costretti a cambiare certe parti assegnate. Il rischio è stato infatti, nel mio caso, che la "parte nuova" proposta a me non avesse a che fare con lo sviluppo del `model` del software. Mi sono comunque occupato alla fine della gestione delle mappe (inteso come parsing e caricamento delle entità definite da file). Nonostante ciò sento comunque di non aver contribuito veramente a qualcosa che io potessi definire abbastanza "challenging" per me, ma soprattutto che appartenesse al `model`. Malgrado queste discordanze di organizzazione fra di noi, posso comunque dichiarare che sono contento del risultato finale. Finisco dicendo che il gioco in sé può essere esteso con diverse features che ci eravamo imposti (a partire dalle features opzionali), e mi farebbe piacere vedere il risultato come lo avevamo immaginato all'inizio.

Marco Sangiorgi

Ritengo che il progetto abbia raggiunto un risultato più che accettabile, sia sotto il livello della complessità, che sotto quello grafico. La parte di sviluppo è stata vissuta in maniera tranquilla in concomitanza con le idee di tutti i membri del gruppo. Gli scambi di opinioni e le ricerche fatte assieme hanno contribuito a migliorare la leggibilità e l'efficacia del codice, perciò ritengo che il gruppo abbia lavorato complessivamente bene. Per quanto riguarda la mia parte, mi sono divertito nel modellare i problemi a me assegnati ispirandomi alla realtà, applicando quanto appreso a lezione. Mi ritengo più che soddisfatto della grado di progettualità raggiunto, dalla struttura delle classi alla logica con le quali e per le quali sono state create. Non ritengo che sia perfetto, anzi alcuni aspetti per motivi di tempo e complessità non

sono stati affrontati nel modo migliore, ma tutto alla fine ha voluto essere funzionale al risultato finale. Sottolineo come la parte di illuminazione non è oggettivamente scalabile e ritengo che sia una buona soluzione nell'ottica dei casi d'uso che abbiamo pensato e progettato per il progetto. Infine per migliorare, è necessario approfondire maggiormente la fase progettuale nella quale non siamo riusciti a coprire tutte le problematiche riscontrate nello sviluppo. Ritengo ci sia solo bisogno di più esperienza, perché la qualità nei singoli non manca.

Appendice A

Guida utente

All'avvio del programma appare il menù principale di gioco. Quando si entra nel gioco tramite il pulsante "START NEW GAME" inizierà il gioco vero e proprio. I comandi di utilizzo all'interno dello scenario:

- W - muove il giocatore in alto
- A - muove il giocatore a sinistra
- S - muove il giocatore in basso
- D - muove il giocatore a destra
- Esc - mette in pausa il gioco

All'interno del gioco si possono raccogliere le chiavi e i power-up passandoci sopra con il personaggio.

Figura A.1: Collezionabili in Hide 'n' Seek

In alto a destra è presente un hud che mostra le chiavi attualmente in possesso e quelle presenti nel livello corrente.

Figura A.2: HUD delle chiavi possedute

Appendice B

Esercitazioni di laboratorio

Nessun membro del gruppo ha consegnato esercitazioni di laboratorio.