ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Triennale in Ingegneria e Scienze Informatiche

Godot: I videogiochi come medium artistico nell'ambito dello sviluppo indie

Tesi di laurea in:
Programmazione a Oggetti

Relatore
Prof. Danilo Pianini

Candidato
Miriana Ascenzo

III Sessione di Laurea
Anno Accademico 2023-2024

Sommario

La seguente dissertazione tratta il tema dei videogiochi esplorati come medium artistico. A seguito di una comparazione delle possibili opzioni di linguaggio di programmazione e/o Engine di Gioco, si sceglie di implementare l'applicazione "Cards Againsts the Dungeon" nel Godot Engine, un motore di gioco che soddisfa i requisiti funzionali e non funzionali esplorati in questa analisi, tenendo conto di vari aspetti sia tecnici che culturali.

Indice

Capitolo 1

Introduzione

Ci sono tanti modi di fare arte: realizzare un dipinto, comporre un brano musicale, scrivere un libro, recitare un'opera teatrale, cucinare un piatto, girare un film. Uno dei molteplici obiettivi dell'arte è quello di fare esperienza di qualcosa, così come offrire un'esperienza alle persone che ci circondano; realizzare un gioco, e quindi per estensione un videogioco, è anch'esso un modo di creare e offrire un'esperienza agli altri, che sia un gioco di logica, un'avventura grafica, un picchiaduro o un farming game: nelle parole di Jesse Schell tratte dal libro *The Art of Game Design*, "i giochi non sono altro che mezzi per raggiungere un fine. Quando le persone giocano, fanno un'esperienza: creare questa esperienza è il fine del game designer" ([Jes14]).

Alcuni tipi di esperienza sono spesso possibili solo attraverso un videogioco: si pensi a giochi horror che sfruttano comandi da console per muovere, creare e modificare file sul desktop per indurre nel giocatore stupore o paura, come nei casi dei giochi *Doki Doki Literature Club!* (Team Salvato, 2017), *Undertale* (Toby Fox, 2015), *OneShot* (Future Car LLC, 2016), oppure avventure che permettono al giocatore di forgiare la propria storia tramite la selezione di opzioni di dialogo, la somma delle quali conduce ad uno di tanti possibili finali, come nei casi dei giochi *Detroit Become Human* (Quantic Dream, 2018), *Life Is Strange* (Dontnod Entertainment, 2015), *Nine Hours, Nine Persons, Nine Doors* (Spike Chunsoft, 2009), o ancora, giochi in cui l'utente è completamente libero di agire, prendere decisioni, avere un certo stile di vita, come nei casi di *The Sims* (EA, 2000-2025),

Stardew Valley (ConcernedApe, 2016), *Elden Ring* (FromSoftware, 2022).

Chiunque può essere un Game Designer, a prescindere dalle proprie abilità e conoscenze: l'importante è desiderare di esserlo e impegnarsi per creare l'esperienza immaginata ([Jes14]). In questa dissertazione si esplora come l'uso di specifici *Game Engine*, a supporto di sviluppatori alle prime armi con la programmazione, possa abilitare aspiranti Game Designer e artisti a usare il videogioco come mezzo per esporre la propria arte e le proprie idee, proponendo soluzioni a problemi di tipo tecnico, di budget e di forza lavoro. A dimostrazione delle possibilità esplorate, viene illustrata la progettazione e lo sviluppo del gioco "Cards Against the Dungeon" a partire da una iniziale ideazione e oportuna schematizzazione, fino a giungere ai dettagli di programmazione e implementazione delle funzioni di gioco.

Capitolo 2

Motivazioni e Background

2.1 Videogiochi come arte: panoramica culturale

Oggi il videogioco è riconosciuto come medium artistico, anche se non da tutti, solo in risposta a numerosi dibattiti portati avanti da critici, studiosi di media, giornalisti e dagli stessi artisti. Nel libro *Per una cultura del videogames*, lo scrittore e docente Matteo Bittanti richiama all'attenzione dei lettori il modo in cui i recensori di videogiochi non prestano alcuna attenzione alle implicazioni ideologiche, politiche, artistiche e culturali degli stessi ([Bit04]).

"l video games sono arte - un'arte popolare, un'arte emergente, un'arte spesso non riconosciuta, ma ciò nonostante un'arte. [..] È giunto il momento di considerare seriamente giochi come una nuova arte popolare che dà forma alla sensibilità estetica del ventunesimo secolo" ([Jen00]).

Lo studioso americano Henry Jenkins, professore in *Communication, Journalism and Cinematic Arts* alla *University of Southern California*, é considerato da Bittanti colui che riesce a dare una soluzione alla questione di videogioco come legittima espressione artistica, in quanto definisce il videogioco uno strumento per superare limiti, prevedibilità e vuotezze causate dal costringersi in uno dei medium artistici più comunemente affermati. Bittanti riporta nel suo libro i dati raccolti dagli studi di settore, che vedono crescere il numero di studenti intenzionati a di-

ventare game designer piuttosto che registi ([Cap15]).

In molti pensano che un videogioco sia da considerarsi artistico solo quando mette piede in territorio cinematografico, ovvero il medium che più si avvicina a quello del videogioco stilisticamente e tecnicamente parlando ([CM07]); tuttavia un gioco non mostra la sua arte solo all'interno di una elaborata cutscene, con giochi di telecamera e luci ben studiati. Ogni aspetto di un videogioco è spesso realizzato proprio a partire da artisti: le grandi aziende si compongono di team di design, a cui appartengono figure come l'artista concettuale, che realizza le bozze e le idee per personaggi, luoghi, oggetti di scena; il musicista, che scrive e compone brani musicali specifici in base all'ambientazione e alla sensazione che si vuole evocare; il modellatore 3D, che realizza gli asset di gioco a partire dalle bozze create; il narratore, che si occupa di creare una storia da raccontare, o anche solo di come si succedono gli eventi all'interno di un'esperienza di gioco; il level designer, figura di rilievo che si occupa della logica e della presentazione visiva di una certa ambientazione o livello di gioco. Tutte queste figure sono sempre presenti e sono fondamentali alla creazione di un gioco, che difatti racchiude molteplici medium in uno, offrendo una nuova esperienza unica e inevitabilmente diversa da un film o un libro. Un gioco permette di esplorare un universo secondo le proprie regole e i propri tempi, permettendo a chi gioca di immergersi completamente nell'esperienza ed eventualmente immedesimarsi in personaggi o situazioni proposte.

2.2 Engine di sviluppo: quale scegliere

Prima di iniziare a scrivere del codice al fine di sviluppare un videogioco, è indispensabile fare un'analisi qualitativa della propria idea, la quale va sviluppata in modo più dettagliato possibile a priori. Lo sviluppatore deve prendere determinate decisioni, come ad esempio: se il gioco è pensato per essere 2D o 3D, su quale piattaforma verrà pubblicato, se il gioco ha funzioni single-player o multi-player, su quale genere e temi si basa, quali particolari calcoli computazionali deve eseguire, quali sono il budget e il tempo a disposizione dello sviluppatore.
Una volta prese queste decisioni, allo sviluppatore spetta fare un'ultima ed impor-

tante scelta, ovvero su quale *Game Engine* realizzare il proprio gioco.
Il Game Engine è un framework software che può semplificare il lavoro di un programmatore. Solitamente fornisce tool pronti all'uso per la gestione degli input, per la computazione matematica, o anche dei motori integrati che si occupano della riproduzione quanto più fedele possibile della fisica del mondo reale, oppure ancora strumenti di animazione e la gestione autonoma dei processi in parallelo ([And15]). Sebbene non esista una vera e propria definizione di "Game Engine", è possibile visualizzarlo come uno strumento che fornisce funzioni comuni alla maggior parte dei videogiochi, permettendo l'uso o la definizione di asset di gioco riutilizzabili quante più volte possibile.

Usare un Game Engine non è obbligatorio: uno sviluppatore può scegliere di programmare direttamente in uno dei linguaggi di programmazione ad uso generale quali C, C++, Java, e così via. L'uso di questi linguaggi comporta tempi di sviluppo maggiori, in quanto implica la costruzione da zero di un effettivo motore di gioco. Il programmatore dovrà quindi occuparsi di reperire delle librerie grafiche (*OpenGL*, *DirectX*, *Vulkan*, ecc...) indispensabili per poter visualizzare a schermo un qualunque elemento di gioco. È a carico del programmatore la costruzione di opportune funzioni necessarie a implementare la logica di interazione tra gli elementi grafici, ad esempio le collisioni, oltre che la definizione di come il giocatore deve interfacciarsi con tali elementi, quindi ad esempio gli input di gioco. Tuttavia, rimane il vantaggio della versatilità di questi linguaggi di programmazione: lo sviluppatore non è vincolato da limiti imposti da un Engine preesistente, e può creare funzioni mirate che si occupano di uno specifico aspetto del gioco che intende sviluppare usando il linguaggio che preferisce.

Scegliere di usare un Game Engine ed evitare lo sviluppo di un intero motore di gioco è spesso la via intrapresa dagli sviluppatori indie con un team ristretto, con limiti di tempo e di budget, o semplicemente con limiti di esperienza pratica a livello di programmazione informatica.

Tra gli Engine più popolari, **Unity** è sicuramente uno dei motori più usati, sia a livello aziendale che indipendente, per produrre giochi 2D o 3D. L'Engine permette

di esportare lo stesso gioco in 15 piattaforme diverse e fornisce tool per l'implementazione del *cross-platforming*, ovvero la possibilità di sincronizzare lo stesso file di salvataggio su molteplici dispositivi e console. Unity dispone di un proprio linguaggio di programmazione, simile a JavaScript, denominato *UnityScript*, ma permette l'utilizzo di altri linguaggi come C#. La community dell'Engine è piuttosto grande ed è inoltre disponibile un esteso marketplace dove gli utenti possono acquistare o vendere asset di gioco. Unity dispone di un piano gratuito e di uno a pagamento, consentendo ai piccoli sviluppatori di iniziare a programmare senza barriere economiche; tuttavia un guadagno annuale superiore ai 200'000$ comporta l'applicazione di una piccola tassa periodica per l'uso dell'Engine.

Di eguale popolarità è **Unreal Engine**, un motore particolarmente potente in ambito di rendering grafico. Il linguaggio predefinito dell'Engine è C++, un linguaggio più complesso in linea con la ripida curva di apprendimento proposta. L'IDE di Unreal Engine è ricco di funzioni, ma poco intuitivo; di recente è stato tuttavia introdotto un sistema di scripting visivo denominato *Blueprint Visual Scripting*, che permette agli utenti di creare un'applicazione tramite l'istaurazione di specifiche connessioni tra oggetti, tutto a livello visivo, riducendone di gran lunga la difficoltà ([Gam25]). Unreal Engine è uno dei più elaborati in ambito di programmazione 3D, ma per questo stesso motivo risulta anche molto pesante e dunque poco performante su computer di fascia bassa. Il motore offre una licenza gratuita a scopo non commerciale, mentre la pubblicazione di giochi realizzati con l'Engine richiede la sottoscrizione al piano pro offerto.

Altre proposte di Engine sono: **GameMaker**, motore che permette la creazione di giochi per desktop, mobile e PlayStation, pensato per chi ha poca dimestichezza con la programmazione, ma che è piuttosto indirizzato alla sola programmazione di giochi 2D; **Construct** e **G Develop**, Engine completamente esenti dalla scrittura di codice e basati sulla definizione di oggetti a cui assegnare comportamenti predefiniti, e per questo comunque limitati nelle funzionalità implementabili; **RPG Maker**, un programma piuttosto popolare, ma limitato al genere RPG con sistema di combattimento a turni e un party da gestire.

Per la realizzazione del gioco preso in esame, Cards Against the Dungeon, si

cerca un Game Engine che possa essere quanto più di supporto a un programmatore con un team ristretto o nullo, con un budget limitato e con una certa predisposizione all'integrazione dell'arte quanto più diretta e intuitiva possibile. Uno dei principali obiettivi è quello di permettere agli artisti che vogliono usare il videogioco come medium artistico di lavorare in modo naturale senza dover affrontare particolari barriere tecniche. Il Game Engine scelto, a fronte di un'analisi dei requisiti funzionali e non, è il **Godot Engine**.

2.3 Godot come scelta di engine di sviluppo

Godot è un motore di gioco open source pubblicato su GitHub con licenza MIT (licenza di software libero creato dal *Massachusetts Institute of Technology*) nel 2014. L'Engine di Godot permette la realizzazione di giochi sia 2D che 3D, che possono poi essere esportati per PC, per mobile e per piattaforme web. Nel 2017 riceve un importante finanziamento da parte di Microsoft, con la richiesta della stessa dell'inclusione di C# come linguaggio di programmazione base ([Etc17]); nel 2020 riceve invece ulteriori finanziamenti da Epic Games, con l'obiettivo di permettere al giovane Engine di espandere il proprio repertorio 3D e di rifinire il rendering grafico ([Lin20]), dimostrando come anche aziende più grandi siano interessate al potenziale di Godot come Game Engine.
Sebbene nato nel 2014, Godot vede un aumento del suo utilizzo solo a partire dal 2019. La *Game Developer Conference* è una conferenza annuale per sviluppatori di videogiochi solita condurre un sondaggio denominato *State of the Game Industry*, con l'obiettivo di delineare lo stato dell'industria, le tendenze, i punti di forza e di debolezza. Questi sondaggi rilevano Godot Engine come uno dei motori con maggiore crescita tra quelli usati per la realizzazione dei giochi pubblicati sulla piattaforma di Steam, in particolare negli anni 2022, in cui l'1,15% dei giochi di Steam sono realizzati con Godot, e 2023, che vede questa stima salire a 1,44% ([Con23]); l'Engine si piazza dunque al sesto posto nella classifica degli engine più utilizzati tra i 65 motori riconosciuti su SteamDB. Molti giochi creati con Godot sono inoltre pubblicati sulla piattaforma itch.io, dove l'Engine in questione raggiunge il terzo posto tra i più usati. Su itch.io prende luogo il GMTK Game Jam,

una competizione per sviluppatori che si sfidano nella creazione di un gioco nel minor tempo possibile seguendo determinati limiti di design; è in queste competizioni che Godot supera di gran lunga le altre opzioni ([Hol24]).

Sebbene non sia possibile definire con accuratezza i motivi dell'aumento di popolarità di Godot come Engine di sviluppo, è comunque lecito fare delle supposizioni basate sui fattori che meglio distinguono questo motore di gioco, in particolare: intuitività, chiarezza dell'interfaccia, curva d'apprendimento del linguaggio di programmazione, customizzazione, estendibilità, compatibilità, costo e community ([LM25b]).
Godot si presenta come un Engine molto leggero, e quindi supportato da un gran numero di computer anche di fascia bassa. Presenta un'interfaccia molto intuitiva, con un design basato su blocchi di costruzione, chiamati nodi, che possono essere disposti ad albero all'interno di quelle che Godot denomina scene. Per ogni nodo, così come per ogni scena, può essere definito uno script, e ogni scena, nodo e script può essere riutilizzato all'interno del progetto come si preferisce. Proprietà di nodi e scene possono essere impostate e modificate direttamente nell'interfaccia, che presenta sulla sinistra la gerarchia dei nodi, e sulla destra un *Inspector* con varie opzioni impostabili nel pannello stesso.
L'engine mette a disposizione molteplici tipi di nodi, ma l'utente è libero di definire ulteriori oggetti personalizzati; sono quelli che Godot chiama *Resources*, oggetti di cui si possono specificare determinate proprietà e funzioni, le stesse che poi compariranno nell'Inspector una volta definite. Le Resources agevolano la creazione di asset di gioco specifici, come ad esempio un inventario per un gioco rpg classico, un tipo di oggetto come ad esempio una spada, o anche un soggetto nemico, permettendo all'utente di istanziare più volte lo stesso tipo di risorsa senza doverla ridefinire ogni volta.
Tra i nodi predefiniti dall'Engine, se ne enfatizzano alcuni di particolare rilevanza ai fini del presente studio; si tratta di nodi che permettono l'immediata importazione di texture e animazioni, che siano due dimensionali o tre dimensionali, di cui si possono specificare determinati comportamenti in base agli input. Sia per i nodi di tipo 2D che per quelli 3D è inoltre presente un pannello di animazione, in cui è possibile realizzare le stesse direttamente nell'Engine qualora si avessero a

disposizione i singoli frame. A partire dall'ultima versione rilasciata al momento della scrittura di questa dissertazione, è inoltre disponibile un *Movie Maker* che permette di registrare momenti di gameplay e scene pre-impostate, che potranno essere usate direttamente come cutscene all'interno del gioco. Un ulteriore elemento di rilievo artistico è il *Tilemap Editor*, che permette di creare in modo rapido mappe di gioco a partire da un set di caselle, che Godot importa e converte in un database di blocchi di costruzione.

Il linguaggio predefinito di Godot è un linguaggio nativo denominato *GDScript*, principalmente ispirato da Python e dunque molto simile nella presentazione e nella curva di apprendimento. È possibile in alternativa scegliere di programmare in C# come precedentemente trattato, nel qual caso è possibile importare le rispettive librerie, oltre che sfruttare le risorse del .NET framework distribuito da Microsoft. Essendo open source, Godot rimane un motore infinitamente estendibile: nella community esistono diverse estensioni messe a disposizione dagli utenti stessi di Godot, che permettono l'utilizzo di svariati linguaggi, quali Rust, Nim, Python, JavaScript, C++. Inoltre nel suo essere open source permette di ridurre di gran lunga i costi di produzione: l'Engine mette a disposizione una repository nota come *AssetLib*, ovvero una raccolta gratuita di asset disponibile a tutti, distribuiti sempre con licenza gratuita MIT, a cui possono contribuire anche gli utenti stessi ([LM25a]).

Uno dei pregi di Godot è dunque l'estesa community che la circorda. Esiste infatti un forum ufficiale creato sulla falsa riga di siti più frequentati e noti come *StackOverflow*. Gli utenti sono molto attivi anche su altre piattaforme quali Reddit, Youtube, itch.io, dove sono messe a disposizione, per la maggior parte gratuitamente, innumerevoli risorse quali tutorial, asset grafici, librerie o interi giochi e/o codici da poter usare ed estendere. In questo ambito si sottolinea il valore di Godot come Engine di gioco per gli sviluppatori sprovvisti di team o comunque con un team ristretto, in quanto la community di Godot è molto ampia e in continua crescita, emergendo negli ultimi anni come uno degli Engine con più supporto ([Hol24]).

Capitolo 3

Analisi

3.1 "Cards Against the Dungeon": panoramica e obiettivi

Viene preso in esame il videogioco sviluppato interamente sul Game Engine Godot denominato "Cards Against the Dungeon". Questo gioco è un *roguelike*, ovvero un gioco in cui la maggior parte degli elementi, quali ambienti, nemici e ricompense, viene generata in modo casuale per ogni partita.
Quando un giocatore inizia una nuova partita, si troverà in una stanza (selezionata casualmente) a cui è associato un nemico da affrontare. Il combattimento con i nemici avviene tramite un sistema a turni: il giocatore avrà a disposizione un certo numero di mosse e un mazzo di carte di diverso effetto, quali attacco, cura, aumento della difesa o dell'attacco. Vincere una battaglia significa ottenere delle nuove carte come ricompensa, oltre che l'accesso alla stanza seguente. Proseguire verso una nuova stanza significa aumentare il proprio livello di un'unità, e un livello più alto comporta un aumento della difficoltà nelle battaglie successive, che vedranno protagonisti nemici più resistenti e con maggiore potenza d'attacco.
In ogni stanza sarà possibile modificare il proprio deck di carte da usare per la battaglia imminente prima che questa inizi tramite l'interazione del giocatore con il corrispondente *trigger*.

Ambienti, design dei personaggi e dei nemici, design di oggetti quali armi, arma-

ture, scudi o grafiche di interfaccia utente e attacchi a disposizione del giocatore, sono realizzati specificamente per questo gioco. Il progetto di Cards Against the Dungeon viene sviluppato avendo tra gli obiettivi principali quello di creare una esperienza di gioco non troppo ardua, oltre che quello di esibire gli artwork creati su misura, in linea con il tema dei videogiochi come medium artistico.

3.2 Requisiti funzionali e non funzionali

Requisiti funzionali Il gioco in esame deve presentare un menu principale all'avvio, che permette al giocatore di iniziare una nuova partita, di riprenderne una già in corso e/o uscire dal gioco e tornare al dekstop con conseguente chiusura dell'applicazione.
Devono essere presenti svariate risorse di gioco:

- un set di carte divise per tipo, ovvero armi, armature, scudi, pozioni e attacchi, ognuna con il proprio effetto e logica;
- un set di nemici con delle stastistiche di base, modificatori per la difficoltà, una IA e un pool di mosse tra cui scegliere;
- un set di stanze divise per elemento, che determinano le caratteristiche dei nemici da affrontare e le carte che si possono ottenere come ricompensa in caso di vittoria;
- lo sprite del giocatore per permettere l'interazione all'interno del Dungeon, provvisto di animazioni e reazioni agli input;
- gli elementi di UI che tengono traccia delle stastistiche, rispettivamente del giocatore con i suoi equipaggiamenti e del nemico.

Se si decide di iniziare una nuova partita, deve essere inizializzato un nuovo mazzo di carte creando una copia del deck iniziale messo a disposizione nelle risorse di gioco; deve inoltre essere impostato il livello del giocatore a uno, e i nemici devono essere inizializzati nelle loro statistiche di base senza modificatori.
Il giocatore, in una fase di "preparazione" antecedente una battaglia, deve poter camminare all'interno della stanza per ispezionare i vari punti di interesse, quali

il mazzo, con cui interagire al fine di poterlo modificare, e il punto di innesco per l'inizio di una battaglia. Interagendo con il deck si deve aprire un sotto menù dedicato all'organizzazione del proprio deck, permettendo l'aggiunta e/o la rimozione di carte a piacere, con conseguente salvataggio o annullamento delle modifiche effettuate.
Durante una battaglia viene pescata una mano iniziale di cinque carte, ognuna delle quali può essere trascinata dal giocatore all'interno dello schermo e successivamente rilasciata nei vari punti di interazione messi a disposizione dall'interfaccia utente, quali slot di equipaggiamento o la zona di attacco al nemico. Per ogni turno devono essere rispettate le mosse disponibili; esaurite tutte le mosse, le carte rimaste nella mano del giocatore devono essere correttamente disabilitate. In un qualunque momento, il giocatore può scegliere di terminare il proprio turno e quindi passare il comando al nemico; quando il nemico avrà terminato il suo turno, il comando tornerà al giocatore, che pescherà le carte mancanti per raggiungere il limite di cinque carte, e tutte le carte (nuove e vecchie) verranno correttamente riabilitate.
Il nemico, dotato di una certa IA, potrà fare durante il proprio turno un'unica mossa. Potrà attaccare, curarsi o aumentare le proprie statistiche, nei limiti del numero di incrementi possibili che saranno definiti in base alla difficoltà della stanza.
In qualunque momento durante la partita, il giocatore può richiamare il menù di pausa, che permette di uscire dal gioco (tornando al menù iniziale o chiudendo l'applicazione) ed eventualmente di salvare o meno i progressi. Il file di salvataggio mantiene le informazioni corrispondenti alla stanza attuale, al livello raggiunto e al deck del giocatore, ma non della battaglia in corso. Se il giocatore dovesse scegliere di riprendere una partita salvata in precedenza, il gioco verrà caricato nella fase di preparazione raggiunta nella sessione precedente; se una battaglia era in corso, questa verrà ricominciata da capo.

Requisiti non funzionali Il gioco in esame ha come pubblico target quelle persone che cercano un'esperienza di gioco poco impegnativa ma che comunque richiede una certa manualità nella gestione delle azioni possibili (definite dalle carte) e del numero di mosse. Il giocatore deve sapersi adattare alla situazione

senza conoscere a priori quali siano gli avversari o le risorse a propria disposizione. Si sceglie un approccio due dimensionale, al fine di rendere il gioco quanto più leggero possibile e quindi eseguibile anche su computer di fascia bassa, permettendo a un pubblico più ampio di farne esperienza.
Il budget a disposizione è nullo: lo sviluppatore è il solo responsabile di codice e grafica, e sarà l'Engine scelto, la documentazione e la community a fare le veci di un ipotetico team di sviluppo.

3.3 Modello Domain

Viene proposto un primo schema generale (fig. 3.1) che definisce la macro struttura del progetto nei suoi elementi distintivi. Vengono dunque illustrati:

- il **menù iniziale**, dal quale scegliere di iniziare una nuova partita, di continuarne una sospesa in precedenza, o di uscire dal gioco per poter tornare al desktop;
- l'oggetto che rappresenta il **ciclo di gioco**, che si compone di una fase di preparazione per modificare il proprio **deck** di carte, e di una fase di battaglia;
- il **dungeon**, ovvero la mappa di gioco in cui si muovono giocatore e nemici, che si compone di vari elementi di **UI** che si aggiornano durante la lotta;
- la classe **Entity**, che può rappresentare il personaggio giocante o il personaggio nemico;
- le classi **Player** e **Enemy** che ereditano da Entity, il primo con un certo numero di mosse (ovvero numero di carte giocabili), il secondo con un certo pool di azioni eseguibili durante la lotta;
- l'oggetto **Deck**, usato dal giocatore in entrambe le fasi di preparazione e battaglia del gioco.

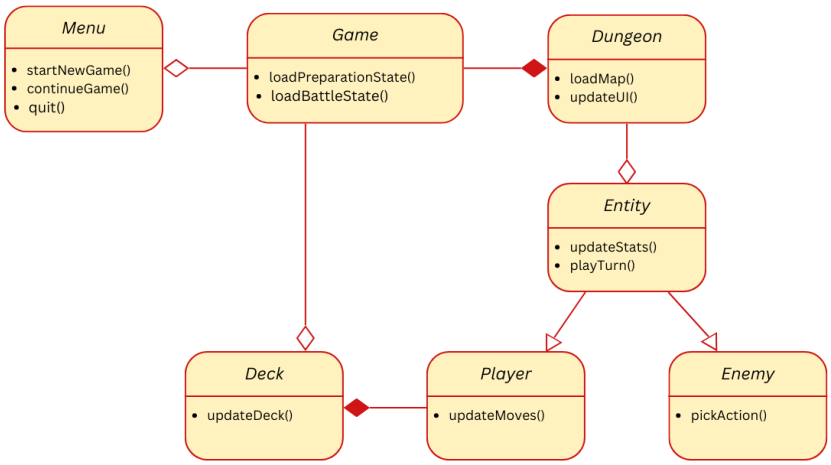

Figura 3.1: Schema UML del domain model

Capitolo 4

Design

4.1 Architettura

L'architettura del software preso in esame è realizzata seguendo il pattern **Composite**, ovvero un *Design Pattern* di tipo strutturale tra quelli descritti dai *Gangs of Four* (GoF).
Il Composite Pattern permette di trattare nodi e composizioni di nodi in modo equivalente, ognuno con il proprio pattern, interfaccia e modo di interpretare gli input, permettendo una facile strutturazione ad albero degli elementi di un progetto ([EGV94]). Godot stesso propone questo tipo di approccio gerarchico ai suoi utenti; l'Engine mette a disposizione delle classi, appunto chiamate Nodi, che l'utente può estendere e poi utilizzare all'interno di una composizione di nodi denominata Scena. Ogni Scena può poi essere usata come Nodo all'interno di altre Scene, generando dunque una struttura ad albero. Ad ogni Nodo o Scena può essere legato uno script, permettendo l'uso degli stessi in modo intercambiabile.

Nella figura fig. 4.1 viene proposto un design che evolve i concetti descritti nel domain model, ponendoli in una struttura gerarchica di nodi in varie relazioni di aggregazione e/o composizione. Per l'elaborazione di questo schema si prendono in considerazione due particolari elementi di Godot: i **Container**, classi di **Controllo** specifici dell'Engine che hanno lo scopo di moderare alcuni comportamenti dei nodi figli, per esempio da un punto di vista di posizionamento nello schermo

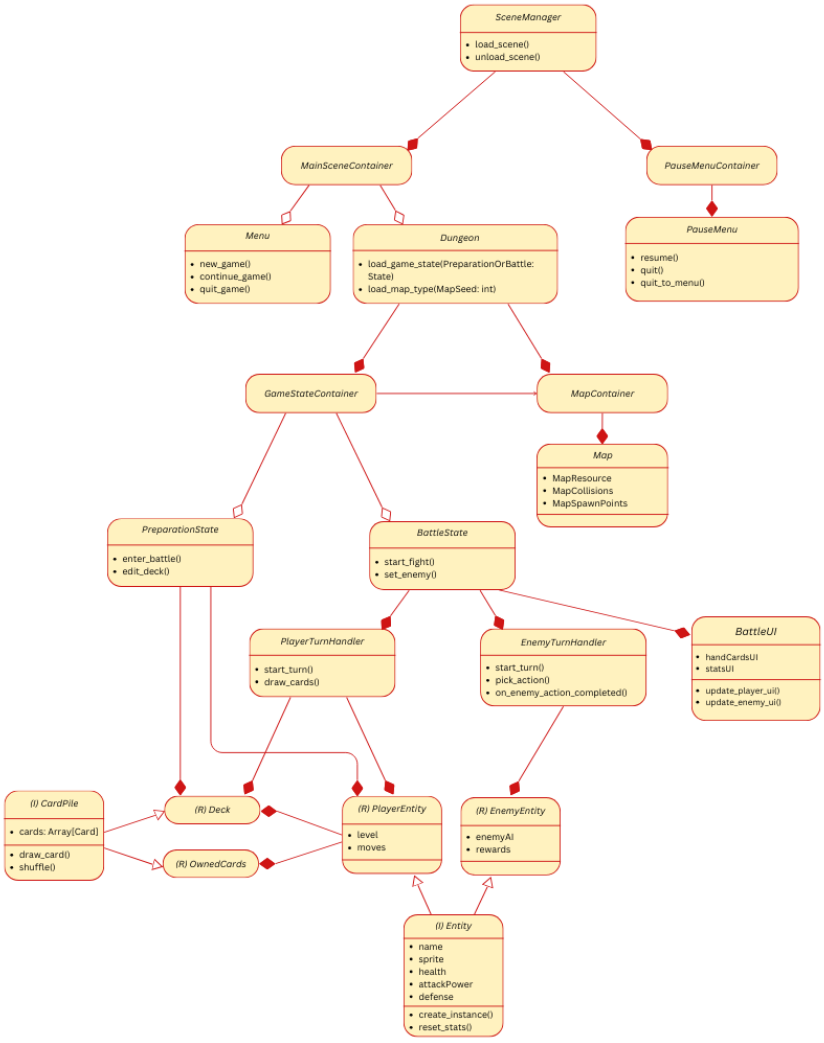

Figura 4.1: Schema UML dell'architettura ad albero del progetto

e ordine di presentazione, e sono dunque usati come veri e propri contenitori; le **Resources**, indicate nello schema con una R, che definiscono tipi di dati creati ad hoc per il progetto, utilizzabili alla pari di array e mappe in un qualunque punto del codice.

- Lo **Scene Manager** rappresenta la radice dell'albero, ovvero la scena che assume il ruolo di main. Tra i nodi figli dello Scene Manager abbiamo due contenitori, uno per la scena corrente e uno per il menù di pausa. Lo Scene Manager si occupa di caricare o rimuovere determinate scene dal **MainSceneContainer** in base al momento di gioco, e di rendere visibile il menù di pausa quando richiesto.

- Le scene caricabili nel MainSceneContainer sono il **Menù Principale**, caricato al primo avvio di gioco, e il **Dungeon**. Quest'ultimo, similmente allo Scene Manager, ha la funzione di caricare la mappa del dungeon su cui il giocatore può muoversi e agire in un **MapContainer**, e di caricare la fase attuale del livello di gioco (quindi la fase di preparazione o la fase di battaglia) in un **GameStateContainer**.

- La fase di preparazione, caricata nel GameStateContainer quando richiesto, si compone della risorsa **PlayerEntity**, ovvero il personaggio giocante, permettendo al giocatore di muoversi tramite esso all'interno del dungeon. La fase di preparazione estrapola poi le informazioni del **Deck** dal PlayerEntity, mantenendone un riferimento ai fini di apportare modifiche a priori di una battaglia.

- La fase di battaglia, anch'essa caricata su richiesta nel GameStateContainer, si compone di nodi che hanno lo scopo di gestire le informazioni e le azioni riguardanti il giocatore e il nemico durante una battaglia, agendo da mediatore tra i due: **PlayerTurnHandler** gestisce le azioni del giocatore, come la pescata di carte o l'uso di un effetto, mentre **EnemyTurnHandler** si occupa delle azioni del nemico del dungeon, quali la scelta del prossimo attacco. Entrambi gli handler fungono anche da *observer*, preoccupandosi di tenere sotto controllo le statistiche di giocatore e nemici e segnalando le modifiche delle stesse ai nodi che gestiscono tali informazioni.

- **PlayerEntity** e **EnemyEntity** sono risorse che ereditano da una interfaccia di risorsa denominata **Entity**: quest'ultima definisce tratti comuni ai due avversari, quali vita, difesa, potere d'attacco, nome e sprite di gioco, mentre le risorse ereditanti definiscono ulteriori informazioni come le mosse disponibili in ogni turno per il giocatore o L'IA e le ricompense per i nemici.
- Viene definita un'ulteriore risorsa di gioco, ovvero la **CardPile**, da cui ereditano le risorse **Deck** e **OwnedCards** (ovvero le carte possedute e che non sono usate nel deck). La pila di carte si compone quindi di ulteriori risorse di tipo **Card** disposte ad array, e per ogni pila sono definite funzioni classiche come la pesca o la mischiata.
- Sempre nella fase di battaglia è definita la scena di **Battle User Interface**, la quale si occupa di rappresentare a schermo la mano del giocatore, il suo equipaggiamento e le statistiche di entrambi giocatore e nemico.

4.2 Design dettagliato

Nella figura fig. 4.2 viene approfondito lo schema precedentemente elaborato. Viene aggiunto un elemento di fondamentale importanza al funzionamento del gioco, ovvero l'**Event Bus**. Quello dell'Event Bus è un pattern relativamente nuovo e dunque non presente fra i pattern classici definiti dal Gangs of Four; esso permette ai vari elementi dell'applicazione di comunicare in *loose coupling* (accoppiamento debole), dunque senza necessità di scambiare informazioni di implementazione o funzionamento ([Che18]).
Tra le varie funzioni di Godot, è importante notare quella dei così detti **signal**, veri e propri segnali che qualunque elemento, che sia un nodo, una resource o una scena, può emettere per comunicare una certa informazione ad altri nodi. L'Event Bus diventa quindi un pattern rilevante in questo Engine, realizzato come un nodo indipendente che raccoglie in sé vari signal per poterli poi comunicare al resto del progetto, senza dover necessariamente essere collegato agli stessi ([Lov21]).
L'Event Bus in Godot è più comunemente implementato tramite Singleton, chiamato anche *Autoload* nella documentazione ufficiale dell'Engine in esame. Godot mette a disposizione questo tipo di script speciale per rimediare alla mancanza di

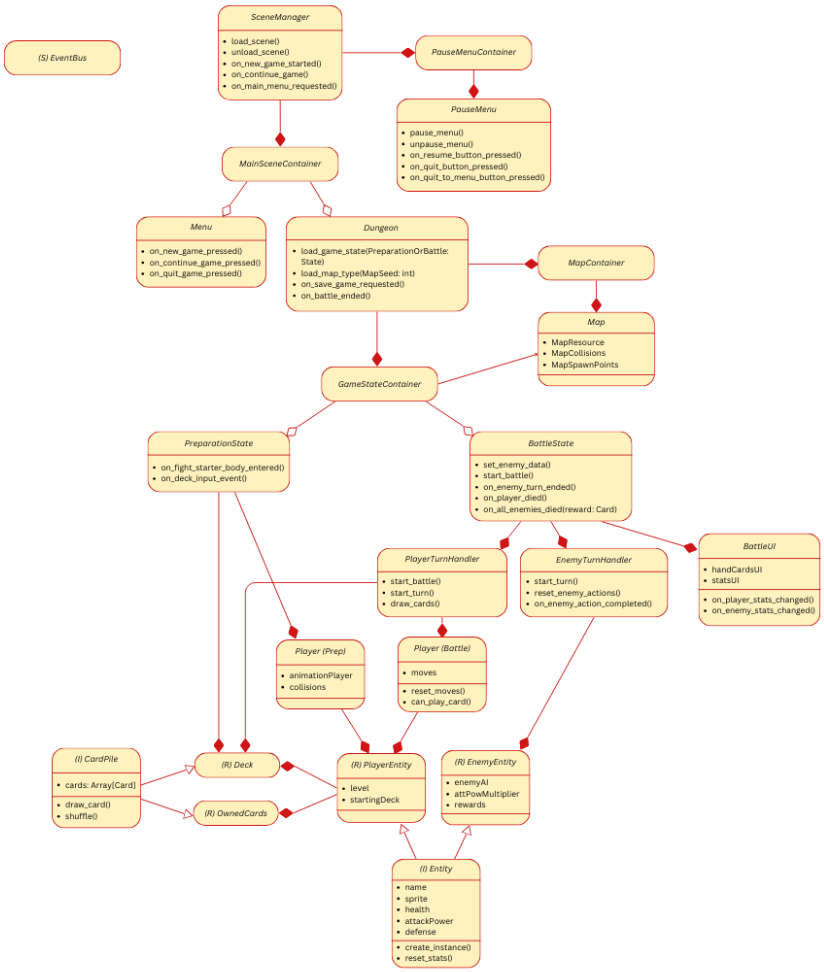

Figura 4.2: Schema UML del design dettagliato del progetto

veri e propri metodi che permettono il mantenimento di informazioni che permangono durante l'intera esecuzione dell'applicazione ([LM25d]). Come suggerisce il termine, un Autoload viene automaticamente caricato in ogni momento a prescindere dalla scena corrente in esecuzione, permettendo di immagazzinare variabili globali e di gestire eventuali transizioni (ad esempio quelle realizzate tramite signal), ma può anche essere usato per implementare veri e propri pattern Singleton. Vari Autoload possono essere creati e poi assegnati dalle impostazioni del progetto; tutti gli Autoload creati vengono costruiti a runtime come nodi, che vengono poi inseriti come figli della radice del progetto prima di ogni altro elemento figlio previsto.

Nel progetto discusso in questa dissertazione, viene usato il pattern Singleton esclusivamente ai fini dell'implementazione dell'Event Bus, così da poter sfruttare al meglio la meccanica dei signal. All'interno dello script associato all'Event Bus non è presente alcuna funzione o logica; esso si limita ad elencare i vari tipi di signal e a comunicare al resto dei nodi del progetto (o perlomeno quelli interessati ai signal in questione) il momento in cui vengono emessi. Tramite Event Bus, nodi distanti tra loro nella gerarchia del progetto hanno modo di sapere quando compiere determinate azioni. Un esempio concreto è la necessità di richiamare il menu iniziale in qualunque momento del gioco: alla pressione del bottone "torna al menù" disponibile nel menù di pausa, viene emesso un segnale specifico che viene raccolto dal Bus e propagato ai nodi interessati a tale richiesta, come ad esempio il dungeon, che ha necessità di salvare i progressi di gioco prima della chiusura della sessione, e lo scene manager, che deve poter sostituire la scena principale di gioco.

Nello schema dettagliato proposto in fig. 4.2 vengono quindi aggiunte varie funzioni di semantica *on_signal_requested* al fine di gestire le azioni da eseguire in risposta ai vari signal qualora emessi.

Viene inoltre sviluppato tramite *Decorator Pattern* la situazione del Player, che ha necessità di differenziare le sue funzioni in base alla fase di gioco corrente. Si definiscono dunque i nodi: **PlayerPrep**, il quale vede una aggiunta di dati relativi al movimento all'interno del dungeon e delle opportune collisioni da gesti-

re; **PlayerBattle**, che tiene conto di dettagli utili alla battaglia, quali le mosse disponibili in un certo turno e la conseguente gestione delle stesse.

Capitolo 5

Implementazione

Di seguito vengono illustrate le diverse scene di gioco e i nodi o risorse di cui si compongono a partire dal punto più alto della gerarchia illustrata in fig. 4.2, dunque dalla radice dell'albero, fino a scendere ai nodi foglia dello schema. Negli schemi proposti in questa dissertazione sono illustrati nodi, attributi e funzioni creati su misura ai fini della realizzazione del progetto, mentre sono omessi segnali e funzioni nativi dell'Engine di Godot, che veranno però trattati in maggior dettaglio nei capitoli a venire.

5.1 Scene Manager

Lo Scene Manager (fig. 5.1) è la scena impostata come radice del gioco nelle impostazioni di Godot, agendo dunque come main. Tra le sue funzioni sono presenti due metodi privati denominati rispettivamente *ready* e *process*: queste sono funzioni predefinite dell'Engine, presenti per ogni classe di Godot. La funzione *ready* si occupa di preparare il nodo prima che venga usato, per esempio per impostare delle costanti o fare delle azioni precedenti all'inserimento della scena nell'albero di gioco; la funzione *process* invece è simile ad un ciclo *while* ed è richiamata dall'Engine ad ogni frame di gioco, aggiornandone ogni volta lo stato. La funzione process è dunque quella che si occupa di processare il modo in cui i nodi si aggiornano durante il ciclo di gioco, oltre che definire la fisica delle interazioni tra oggetti, come ad esempio muovere uno sprite o gestire le collisioni con l'am-

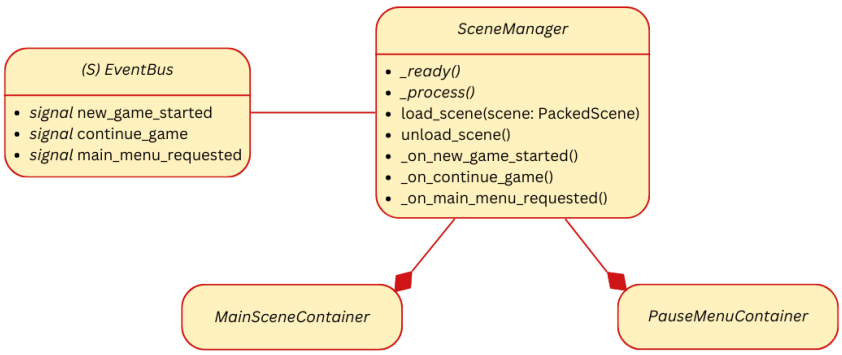

Figura 5.1: Schema UML dello Scene Manager

biente ([LM25c]), tutto ad una frequenza di default di 60 fotogrammi al secondo (comunque modificabile dalle impostazioni di progetto). Alla funzione di process viene inoltre passata una variabile *delta*, la quale definisce il tempo, in secondi, trascorso dall'ultima chiamata alla stessa funzione. Il parametro delta permette al programmatore di fare determinati calcoli in modo indipendente dalla frequenza di fotogrammi: per esempio, per definire la velocità di movimento di un oggetto è necessario moltiplicare una certa velocità per il parametro.

Nel caso specifico dello Scene Manager, la funzione **ready** si occupa di collegare alcuni segnali dell'EventBus a delle funzioni di callback. Si tratta dei segnali corrispondenti a: inizio di una nuova partita, continuo di una partita precedente, richiesta di tornare al menu principale. A seguito degli opportuni collegamenti, la funzione di ready termina le sue operazioni caricando la prima scena, ovvero il menù principale, tramite la funzione *load_scene(scene: PackedScene)*.

La funzione **load_scene(scene: PackedScene)** consiste in un iniziale **unload_scene()**, che rimuove tutti i nodi figli presenti in *MainSceneContainer*, e successivamente aggiunge un nuovo nodo figlio, creato tramite istanziazione della scena di tipo PackedScene (definito da Godot) passata come parametro alla funzione. Quando si parla di rimozione di nodi figli, si differenziano due diversi metodi proposti dall'Engine:

- **node.child.queue_free()** libera definitivamente dalla memoria il nodo figlio indicato, il quale dovrà quindi essere ricreato da zero qualora dovesse servire nuovamente;

- **node.remove_child(child)** si limita a staccare un figlio da un nodo, che rimane però in memoria mantenendo i dati che esso conteneva al momento della rimozione, e potrà essere richiamato in seguito senza perdere l'istanza e le informazioni al suo interno.

Per il caricamento e la rimozione di nodi nel MainSceneContainer, è necessario usare la funzione queue_free(), in quanto, per design di gioco, ogni livello richiede la creazione di un nuovo Dungeon secondo parametri generati in modo casuale, quindi la montatura sul momento di una mappa di gioco, del nemico da affrontare e delle modifiche da fare alle statistiche del nemico in base al livello del giocatore.

Le funzioni di callback dei segnali sfruttano le funzioni di caricamento e rimozione, precedentemente trattate, in base alla situazione:

- **on_new_game_started()** è la funzione di callback che reagisce all'emissione da parte del menu principale del segnale *new_game_started*. Se richiamata, lo Scene Manager rimuove il menù principale dal MainSceneContainer, istanzia un nuovo Dungeon e lo aggiunge come nuovo figlio del contenitore. Istanziato il nuovo dungeon, viene emesso anche un segnale *new_dungeon_entered* per richiedere al gioco l'inizializzazione e caricamento del primo livello della partita.

- **on_continue_game()** reagisce al segnale *continue_game* che parte sempre dal menù principale; similmente alla nuova partita, la funzione rimuove il menù principale e carica un Dungeon con parametri specifichi che vengono estratti dal salvataggio di gioco e assegnati dopo l'esecuzione della funzione ready del Dungeon.

- **on_main_menu_requested()** reagisce al segnale *main_menu_requested* che può essere emesso dal menù di pausa, o anche dallo schermo di sconfitta qualora il giocatore dovesse perdere tutti i punti vita durante una battaglia.

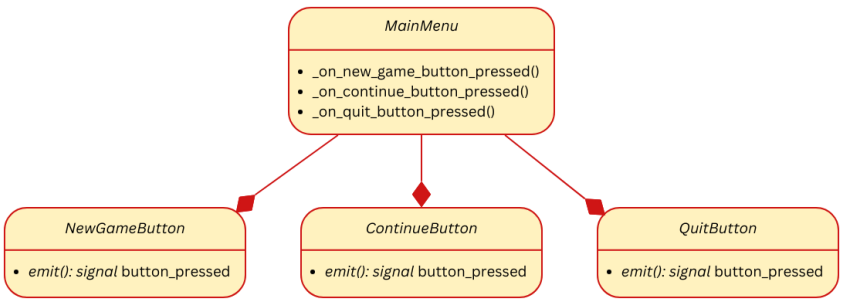

Figura 5.2: Schema UML del menu iniziale di gioco

La funzione toglie la scena del Dungeon dal contenitore per sostituirla con un nuovo menù iniziale.

Infine, la funzione **process** definita nello Scene Manager si occupa di richiamare il menu di pausa ogni volta che il giocatore preme il pulsante "Esc" (operazione disabilitata se situati nel menu iniziale di gioco). Il pulsante Esc, così come altri pulsanti della tastiera e del mouse, sono configurati nella mappa di input direttamente nelle impostazioni del progetto. Funzioni native di Godot permettono di rilevare gli input definiti nella mappa così da poter facilmente gestire determinati comportamenti di gioco; finchè due input vengono definiti nella mappa con lo stesso nome, ad essi viene attribuito lo stesso comportamento, permettendo al programmatore di rendere il proprio gioco compatibile con molteplici hardware di input come tastiera, mouse, controller e altri.

5.2 Menu

Il menu iniziale di gioco (fig. 5.2) presenta anch'esso una funzione **ready** che si occupa di rilevare un file di salvataggio. Se tale file fosse inesistente, o se dovesse presentare al suo interno un valore booleano *saveExists* impostato su false, la funzione disabilita il pulsante "Continue".
La funzione **on_new_game_button_pressed()** è una semplice funzione di callback in risposta alla pressione del bottone corrispondente nel menù, così come anche **on_continue_button_pressed()**. Tali bottoni non hanno uno script, in

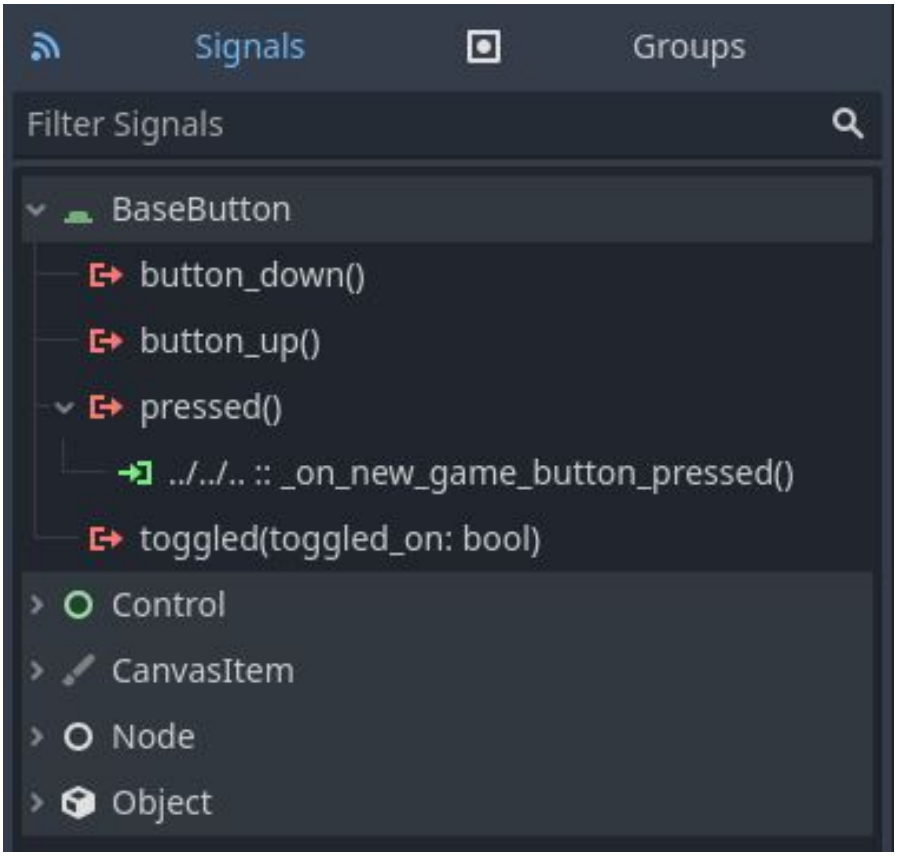

Figura 5.3: Finestra dei signals nell'Inspector di Godot. Nell'immagine, al signal pressed() emesso dal bottone è associata la funzione di callback _on_new_game_button_pressed().

quanto la gestione dei segnali che possono emettere e quali funzioni devono rispondere a tali segnali sono configurabili direttamente dall'Inspector dell'interfaccia di Godot, posta sulla destra del programma: ciò è possibile fin quando tali nodi e tali funzioni appartengono tutti alla stessa scena (fig. 5.3). Le funzioni di callback definite si limitano ad emettere i corrispondenti segnali *new_game_started* e *continue_game* che vengono poi rilevati dall'EventBus.

Il menu di pausa (fig. 5.4), così come quello iniziale, si compone di vari bottoni senza script e che emettono determinati segnali quando premuti, con le rispettive funzioni di callback definite nello script associato al nodo padre. In questa scena sono definite le funzioni **pause_menu** e **unpause_menu**, le quali si occupano di

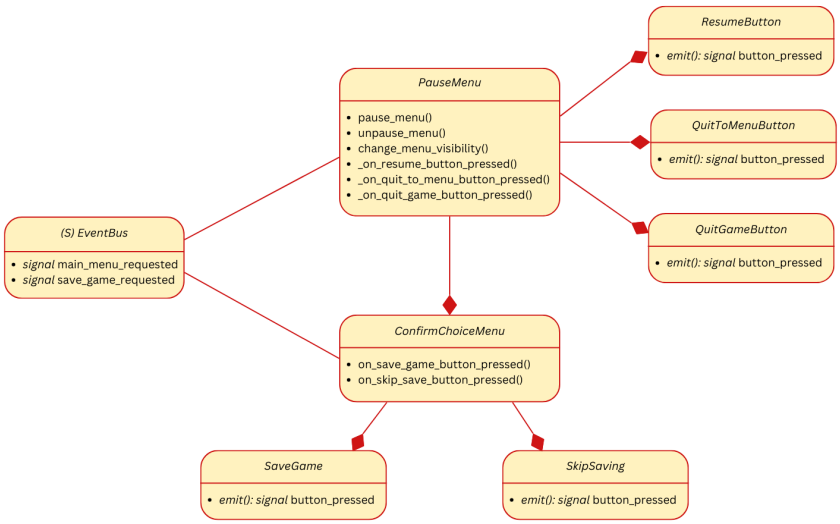

Figura 5.4: Schema UML del menu di pausa

rendere visibile il menu e di mettere in pausa l'intero processo di gioco. Il processo di gioco, richiamabile tramite funzione nativa di Godot *get_tree()*, ha un attributo di tipo booleano denominato *paused*; inoltre, tutti i nodi hanno un attributo di tipo speciale *process_mode* che può assumere valori "pausable", "when paused", "always", "disabled" e "inherited". Quando il valore "paused" del processo di gioco è impostato su true, tutte le scene, e i rispettivi figli con processo in modalità "pausable" vengono fermate, mentre tutte le scene e nodi con processo in modalità "when paused" diventano attive e interagibili; il contrario avviene quando paused è impostato su false. Lo stesso sistema viene usato anche per altri sotto menu che richiedono un momento di stasi da parte del gioco, per esempio nel menu di modifica del deck.

La funzione di callback alla pressione del bottone per tornare al menu principale, **on_quit_to_menu_pressed()**, cambia la visibilità degli elementi del menu, nascondendo i bottoni resume e uscita dal gioco e mostrando nuovi bottoni di conferma di uscita e di richiesta di salvataggio del gioco. Premendo il bottone "save game" viene emesso il segnale *save_game_requested* da mandare all'EventBus, altrimenti viene emessa la sola richiesta di uscire dal gioco o di tornare al menu

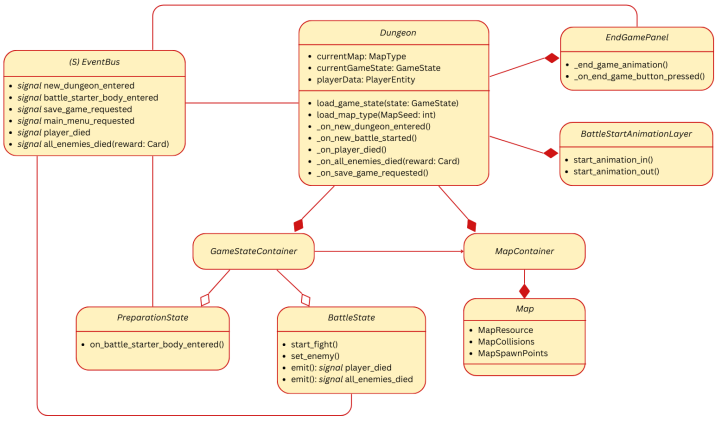

Figura 5.5: Schema UML del Dungeon

iniziale tramite emissione del segnale *main_menu_requested*.

5.3 Dungeon

La scena del Dungeon (fig. 5.5) è quella più centrale dell'applicazione: essa prevede di gestire la maggior parte degli aspetti del ciclo di gioco, come ad esempio la scelta della mappa, il cambio della fase tra preparazione e battaglia, la gestione della vittoria o della sconfitta.

Essa si compone di due contenitori, uno per la gestione del GameState corrente, l'altro per la gestione della Mappa del dungeon corrente. Tra i figli del Dungeon vediamo: un *Panel*, classe di Godot che agisce in modo simile a un container ma a cui può essere associato un tema (ovvero una risorsa di tipo *Theme* assegnabile e personalizzabile direttamente dall'Inspector); un *CanvasLayer*, ulteriore nodo specifico dell'Engine che crea un vero e proprio livello posto al di sopra (o al di sotto) del resto degli elementi della scena, in modo da ignorare vari posizionamenti di camera se previsti, o anche solo per richiamare il focus su di esso e impedire interazioni con elementi sottostanti qualora fosse necessario. Il Panel serve a visualizzare e gestire la fine della battaglia, mentre il CanvasLayer è usato per una semplice animazione di inizio battaglia, reso visibile durante la transizione tra fase

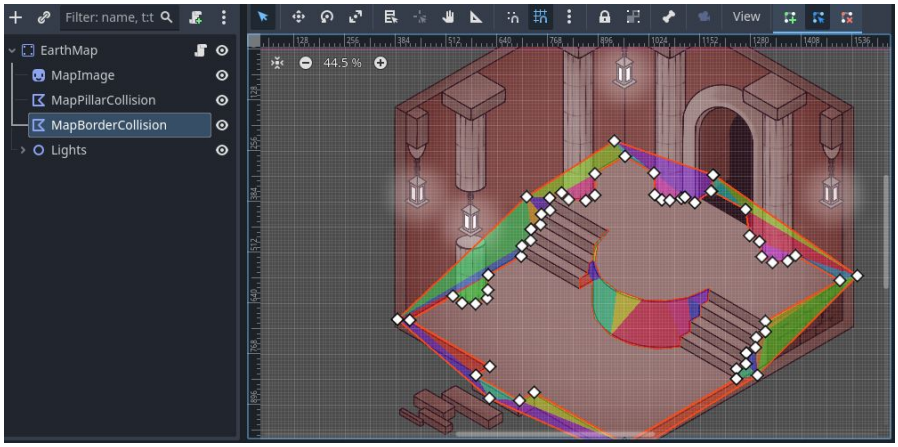

Figura 5.6: poligono di delinazione collisioni di una delle mappe di gioco.

di preparazione e fase di battaglia.

Le diverse mappe di gioco sono implementate in scene separate per permettere una maggiore personalizzazione dell'ambiente. Hanno in comune un nodo di texture per l'immagine della mappa, un nodo di collisioni e delle coordinate di *spawn*, ovvero i punti in cui posizionare il giocatore, il nemico, il deck e il punto d'innesco per l'inizio della battaglia. Le collisioni vengono realizzate tramite editor centrale del Godot Engine, che permette di creare dei poligoni di delineazione dei punti della mappa oltre cui il giocatore non può andare, per esempio sui muri o sulla colonna centrale presente in una delle mappe (fig. 5.6).

La funzione **ready** del Dungeon inizializza per la prima volta i dati del giocatore in caso di inizio di una nuova partita, oppure resetta le statistiche in caso di passaggio a un nuovo livello di una partita in corso; collega inoltre le opportune funzioni di callback a determinati segnali di interesse al Dungeon. A partire dalla funzione di callback **on_new_dungeon_entered()**, che risponde al primo segnale *new_dungeon_entered* emesso dallo Scene Manager alla pressione del bottone di nuova partita, si crea un effetto a cascata di segnali, la cui risposta ne richiama altri, generando quindi la sequenza di eventi che costituiscono il ciclo di gioco. Si esaminano le funzioni di callback definite nella scena del Dungeon:

- **on_new_dungeon_entered()** si assicura di nascondere il Panel di fine bat-

taglia qualora fosse ancora visibile dal livello precedente; successivamente genera, con una funzione *rand()*, un numero casuale intero che definisce il tipo di mappa da selezionare per il livello corrente. Generato un numero, ovvero il *MapSeed*, questo viene passato come parametro alla funzione **load_map(MapSeed: int)** che associa ad ogni numero una risorsa di mappa, e la carica nel MapContainer appena creata una nuova istanza **Map**. A seguire, viene caricata la fase di preparazione nel GameStateContainer tramite funzione **load_game_state()**, che, similmente al caricamento della mappa, carica la scena corrispondente nel GameStateContainer dopo aver creato una nuova istanza **GameState**. Entrambe le istanze di mappa e fase di gioco vengono inserite nei rispettivi contenitori solo dopo una pulizia di eventuali figli già presenti tramite funzione di Godot *queue_free()*.

- La funzione **on_new_battle_started()** è richiamata quando il giocatore dà inizio a una nuova battaglia entrando nel punto di innesco presente in mappa durante la fase di preparazione, il quale emette il segnale *battle_starter_body_entered*. La battaglia comincia solo dopo un'animazione richiamata sul CanvasLayer precedentemente illustrato, finita la quale viene caricata, tramite load_game_state(), la fase di battaglia.

- In caso di sconfitta durante la fase di battaglia, viene richiamata la funzione **on_player_died()** in risposta all'omonimo segnale. In questo caso, viene reso visibile il pannello di EndGame con una piccola animazione, e su di esso viene impostato il bottone per tornare al menu principale tramite emissione del segnale *main_menu_requested* a cui risponde lo Scene Manager. Una situazione analoga si verifica in caso di vittoria in risposta al segnale *all_enemies_died(reward: Card)*, per cui nella funzione **on_all_enemies_died(reward: Card)** si rende visibile il pannello, ma questa volta con un bottone che richiede l'entrata in un nuovo livello, e quindi un nuovo dungeon emettendo il segnale *new_dungeon_entered*. La funzione di vittoria prende in input la ricompensa della battaglia, ovvero una carta, che verrà inserita nella lista di carte a disposizione del giocatore; in segno di vittoria vengono inoltre aumentati i punti vita massimi.

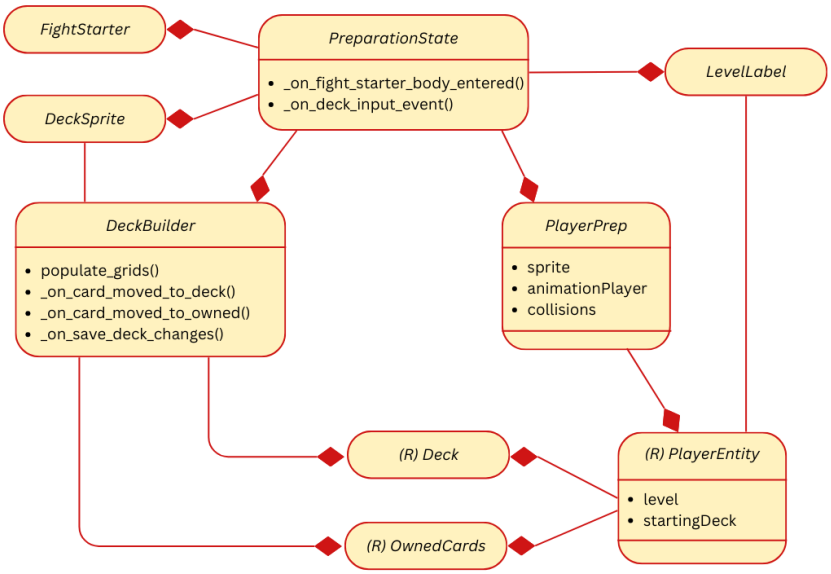

Figura 5.7: Schema UML del Preparation State

- **on_save_game_requested()** è la funzione di callback che risponde al segnale di richiesta salvataggio da parte del menu di pausa; questa funzione raccoglie tutti i dati presenti nel Dungeon corrente, quindi dati del giocatore, mappa corrente e livello corrente, salvandoli all'interno di una risorsa di gioco *SaveGame*.

5.4 Fase di Preparazione

La fase di preparazione (fig. 5.7), al momento dell'istanziazione, prende in input un riferimento ai dati del giocatore impostati nel Dungeon tramite la funzione di build di Godot denominata **init()**. La scena PreparationState estrapola dai dati del giocatore delle informazioni quali il livello, che imposta in un elemento di Godot di tipo *Label* posto in alto alla scena, e il deck di carte, così da poter creare in modo opportuno il sotto menu di modifica del mazzo.

Questa scena si compone di vari elementi, che siano ulteriori scene figlie o semplici nodi:

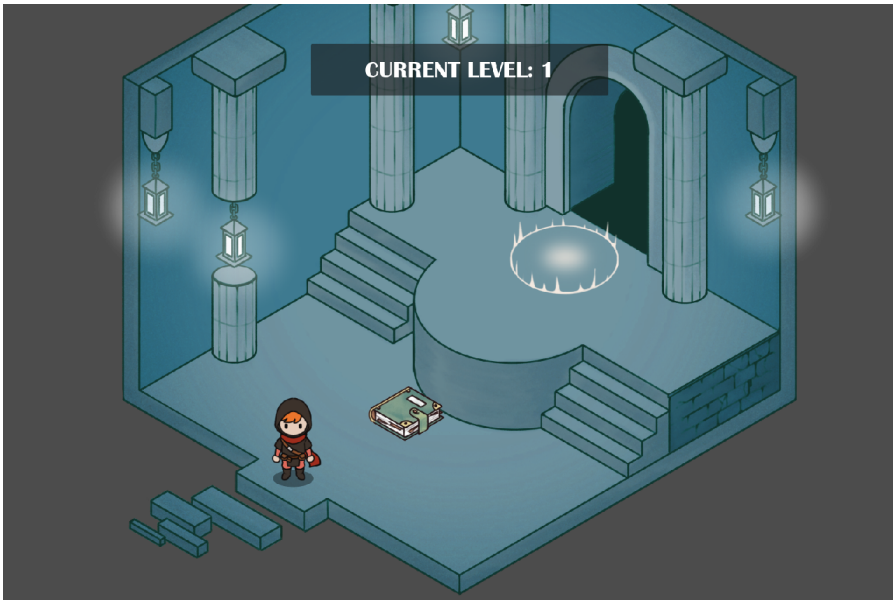

Figura 5.8: Screenshot di gioco durante la fase di preparazione antecedente una battaglia.

- La scena **PlayerPrep** è una scena che si compone del **PlayerEntity**, ovvero della risorsa di gioco nella quale sono raccolti tutti i dati del giocatore. Arrichisce le informazioni del giocatore con una scena in cui sono definite animazioni di movimento e area di collisione: ai piedi dello sprite è presente un cerchio blu (fig. 5.13), invisibile durante l'esecuzione di gioco, che rappresenta i punti dello sprite che possono interagire con altre zone di collisione, per esempio quelle della mappa (fig. 5.6).

- I nodi **FightStarter** e **DeckSprite** sono semplici sprite che reagiscono ad input tramite segnali impostati nell'Inspector a destra dell'interfaccia. In particolare, FightStarter è uno sprite animato che riproduce un'animazione di default in loop, dunque non necessita di un più complesso AnimationPlayer: poichè dotato di animazione unica, essa può essere riprodotta tramite funzione *play* associata ai nodi di tipo *AnimatedSprite*, eseguita nella funzione process() del PreparationState. Lo sprite emette segnale *battle_starter_body_entered* quando il giocatore ci passa sopra con il PlayerPrep, avviando la fase di battaglia. In particolare, l'interazione tra PlayerPrep e

FightStarter è resa possibile tramite la fisica nativa di Godot delle collisioni: finchè a due nodi viene assegnata una *CollisionShape* (l'area blu in fig. 5.13), questi possono emettere un segnale di default di tipo *on_area_entered* qualora dovessero entrare in contatto l'uno con l'altro. Il DeckSprite possiede anch'esso un nodo di collisione per evitare che il player possa camminarci sopra.

- Cliccare con il mouse sullo sprite del Deck richiama invece il sotto menu **DeckBuilder**, una scena inizialmente invisibile e poi resa visibile con assegnazione del corrispondente valore booleano tramite funzione **on_deck_input_event()**. Il DeckBuilder si compone di due nodi di Godot di tipo *GridContainer*, quindi due diverse griglie, una contenente le carte del mazzo del giocatore, l'altra le carte che il giocatore possiede ma che non ha inserito nel mazzo. Ogni carta in qualunque griglia è cliccabile, e se cliccata questa si sposta nella griglia opposta tramite un aggiornamento visivo che avviene richiamando la funzione **populate_grid()**, aggiornando di conseguenza le liste di carte corrispondenti. Se si decide di salvare le modifiche apportate, viene chiuso il menù DeckBuilder e la modifica viene salvata nel PlayerEntity. Sempre in questa scena è presente un piccolo riquadro che mostra a schermo la descrizione di una carta qualora il giocatore dovesse passarci sopra con il mouse (fig. 5.9); questa meccanica è resa possibile sempre tramite signal di default, anch'esso impostato nell'Inspector a destra dell'interfaccia di Godot (*mouse_entered signal*).

5.5 Fase di Combattimento

La fase di combattimento (fig. 5.10), similmente alla fase di preparazione, riceve un riferimento ai dati del giocatore e alla mappa di gioco corrente al momento dell'istanziazione. Nella funzione **ready()**, il BattleState imposta vari dati a partire dai parametri ricevuti in **init()**:

- A livello locale viene mantenuto il riferimento al **PlayerEntity**, così da poter aggiornare le statistiche e il deck in base a come evolve la battaglia.

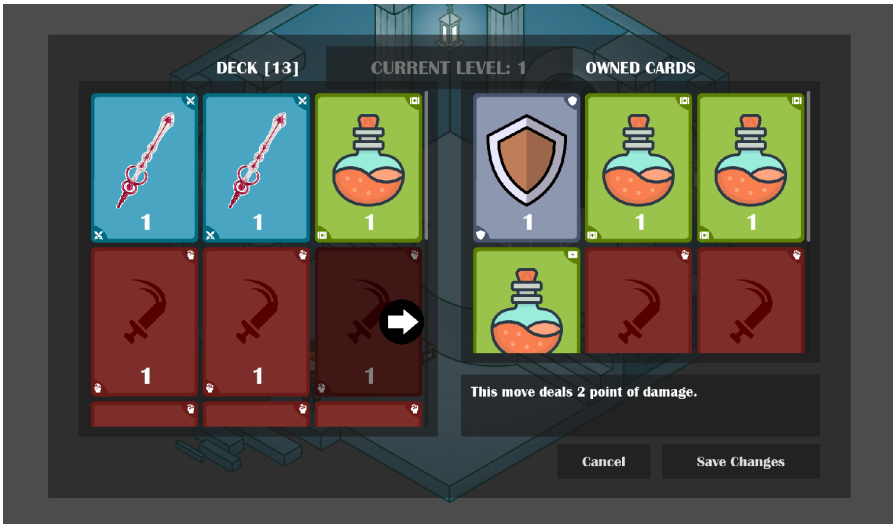

Figura 5.9: Screenshot di gioco del sotto menù di modifica del deck.

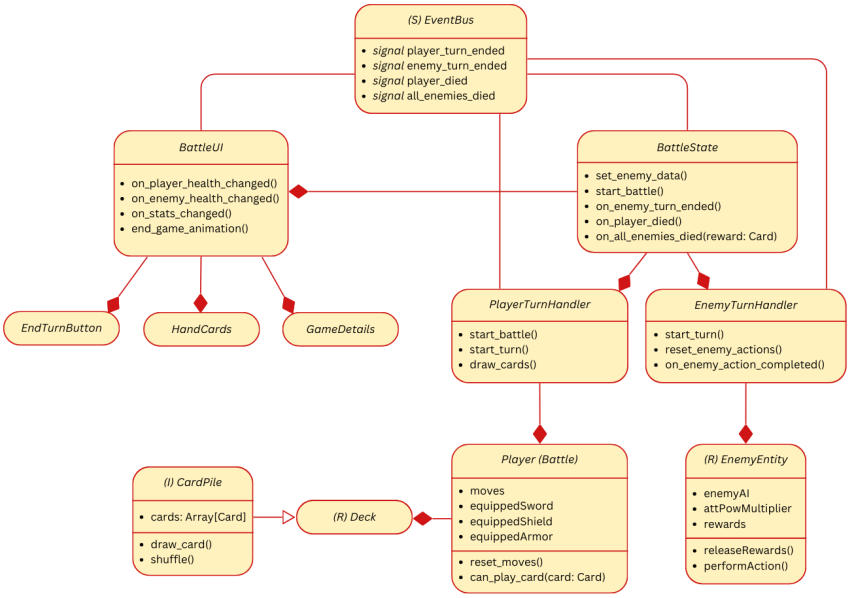

Figura 5.10: Schema UML della fase di battaglia

- A partire dai dati della mappa di gioco, il BattleState imposta le proprietà del nemico da affrontare tramite funzione **set_enemy_data()**. I nemici sono istanziati in base al tipo della mappa, per esempio se la mappa è di tipo acqua verrà istanziato un nemico di tipo acqua. Le statistiche dei nemici vengono inoltre modificate in base al livello del dungeon: ogni tre livelli la vita del nemico aumenta di tre unità e il suo potere d'attacco aumenta di un terzo del livello, così da regolare la difficoltà della partita man mano che i mostri vengono sconfitti.

- Una volta impostati i riferimenti al **PlayerEntity** e al **EnemyEntity**, gli stessi vengono passati alla **BattleUI**, così che quest'ultima venga inizializzata e possa correttamente mostrare a schermo i dati di gioco man mano che vengono aggiornati.

- Vengono infine collegati alcuni signal alle rispettive funzioni di callback, in particolare quelli di fine turno (sia per giocatore che nemico), di vittoria, di sconfitta e di richiesta di uscita dal gioco.

Il BattleState si compone di due nodi di script: **PlayerTurnHandler** e **EnemyTurnHandler**. Questi nodi sono i principali attori della scena, ognuno con la propria funzione **start_turn()** e le eventuali funzioni di gestione del turno. Nella funzione di ready del BattleState viene richiamata la funzione **start_battle()**: essa si occupa di resettare l'IA del nemico tramite funzione **reset_enemy_actions()** dell'EnemyTurnHandler e di iniziare il turno del giocatore richiamando la funzione start_turn() del PlayerTurnHandler, che ha quindi priorità di turno.

Il turno del giocatore consiste in un iniziale **reset_moves()**, così da riportare il numero di mosse disponibili nel proprio turno a quattro, ovvero il valore scelto per la versione di gioco trattata in questa dissertazione. Una volta ripristinate le mosse, il mazzo del giocatore viene mescolato tramite metodo **shuffle()**, e vengono pescate delle carte tramite funzione **draw_cards(cardsToDraw)**. Il giocatore può tenere in mano fino a un massimo di cinque carte: prima did pescare, viene svolto un calcolo per ottenere il parametro *cardsToDraw*, ovvero le carte da pescare per arrivare ad avere esattamente cinque carte in mano. Il calcolo è possibile

tramite funzione di Godot **get_child_count()** che permette di contare i figli di un nodo, in questo caso del nodo **HandCards**, figlio della BattleUI.
Le carte che vengono pescate sono istanziate sul momento, creando un nodo di decorazione **CardUI** che si compone dell'entità **Card**; le carte vengono poi aggiunte alla mano tramite metodo **add_card()** definita per HandCards, nodo di tipo *VBoxContainer* che dispone in automatico i suoi figli in riga e in ordine di aggiunta. Se nel deck non sono più presenti carte da pescare, vengono generate delle carte di attacco più deboli per permettere al giocatore di finire la partita.
Durante un turno il giocatore ha quattro mosse disponibili ed ogni carta ha uno specifico costo in mosse. Se il giocatore non ha abbastanza mosse per giocare una carta, quest'ultima viene disattivata, e la sua grafica diventa trasparente con delle scritte rosse a segnalare la sua disattivazione. Finite le mosse, tutte le carte ancora in mano non saranno più interagibili e al giocatore non rimarrà altro che dover terminare il turno tramite l'opportuno bottone **EndTurn**. Tale bottone emette un segnale *player_turn_ended* che permette al nemico di iniziare il suo turno.

Il turno del nemico consiste in una funzione **perform_action()** di EnemyEntity, la quale produce un singolo effetto di attacco, potenziamento o cura, scelta in modo casuale tra le azioni disponibili nell'IA del nemico. Quando l'azione è conclusa, viene chiamata **reset_enemy_actions()**, una funzione che genera una nuova azione da usare nel turno successivo, concludendo con l'emissione del segnale *enemy_turn_ended* che passa il turno al giocatore.

Quando il giocatore o il nemico vengono sconfitti, ovvero quando uno dei due avversari esaurisce i punti di vita, l'Entity corrispondente emette un segnale di tipo *player_died* o *all_enemies_died(reward: Card)*, che verranno rilevati dal Dungeon con conseguente animazione di fine partita o livello. Nel caso di vittoria, viene eseguita la funzione **release_rewards()** di EnemyEntity al fine di scegliere una carta ricompensa casuale tra quelle definite nell'array di possibili carte ricompensa del nemico; questa viene passata tramite signal alla scena del Dungeon, responsabile dell'assegnazione del premio al giocatore in fase di gestione della vittoria.

Il nodo **BattleUI** è un nodo di tipo *CanvasLayer*, dunque un elemento che

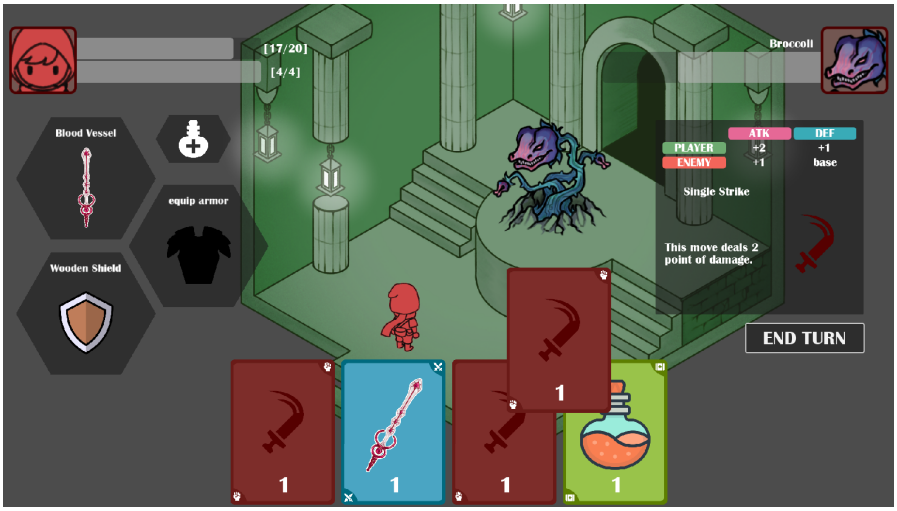

Figura 5.11: Screenshot di gioco della fase di battaglia. Nella figura, il giocatore trascina una carta attacco all'interno della scena.

sovrasta tutti gli altri all'interno della scena, con priorità di focus per ogni input rilevato. Esso si compone di vari elementi:

- In alto sono presenti la barra della vita e le mosse disponibili del giocatore a sinistra, e la barra della vita del nemico a destra, che si aggiornano ad ogni effetto applicato durante la battaglia tramite funzione **on_stats_changed()** in risposta ai segnali *entity_changed* emessi da PlayerEntity e EnemyEntity.

- Sulla sinistra sono presenti gli equipaggiamenti del giocatore, ovvero tre slot per assegnare una spada, uno scudo e un'armatura, oltre che uno slot aggiuntivo per applicare eventuali carte di tipo cura. Gli slot di equipaggiamento mantengono un riferimento alla rispettiva carta assegnata; l'assegnazione delle carte equipaggiamento è gestito tramite la meccanica delle collisioni, dunque quando una carta collide con lo slot di equipaggiamento viene mandato un segnale rilevato dal PlayerEntity, il quale aggiorna di conseguenza le statistiche di attacco e/o difesa.

- Sulla destra sono presenti un Panel di informazioni sull'attacco e la difesa di entrambi gli avversari che aggiornano le informazioni in risposta ad ogni segnale di tipo *entity_changed*, seguito dal bottone per terminare il turno. Quando il giocatore muove il puntatore del mouse sopra una carta della

```
func move_hand_cards(hiding: bool) -> void:
  if not is_inside_tree():
    await ready

  var positionB = Vector2(384, 761)

  var tween := create_tween().set_trans(Tween.TRANS_QUINT)
  tween.tween_property(self, "global_position", positionB, 0.4)
  tween.tween_interval(HAND_HIDING_SPEED)
  tween.finished.connect(func(): Events.hand_cards_moved_down.emit())
```

mano, le relative informazioni sono mostrate (e poi nascoste) nella porzione di Panel sottostante alle statistiche, possibile tramite i due segnali nativi di Godot di tipo *mouse_entered* e *mouse_exited* emessi da ogni singola CardUI.

- In basso è presente la mano di carte, un container orizzontale che come anticipato dispone i suoi elementi in riga. Le carte al suo interno possono essere cliccate, trascinate e rilasciate finchè il giocatore dispone ancora di mosse per usarle, informazione verificabile tramite funzione **can_play_card(card: Card)** di PlayerEntity, mettendo a confronto il costo della carta e le mosse rimaste al giocatore. Le carte possono essere rilasciate sui vari slot di equipaggiamento oppure sul nemico stesso; tutti questi elementi hanno una *Area2D*, nodo di Godot che similmente alle CollisionShape2D verifica l'interazione tra collisioni, assegnata per rilevare quale collisione è avvenuta e dunque quale equipaggiamento assegnare.

Per tutti gli elementi di UI sono previste delle animazioni di fine partita che consistono nello spostamento di ogni nodo al di fuori dello schermo. Queste semplici animazioni di nodi sono realizzabili tramite la classe di Godot denominata *Tween*: un tween può essere istanziato con funzione **create_tween()**, il suo tipo (per esempio movimento o dissolvenza) può essere impostato con **set_trans()**, gli elementi da animare e in quale modo animarli possono essere impostati in **tween_property()**, e alla fine dell'animazione può essere associata una funzione di callback, per esempio l'emissione di un signal (sezione 5.5).

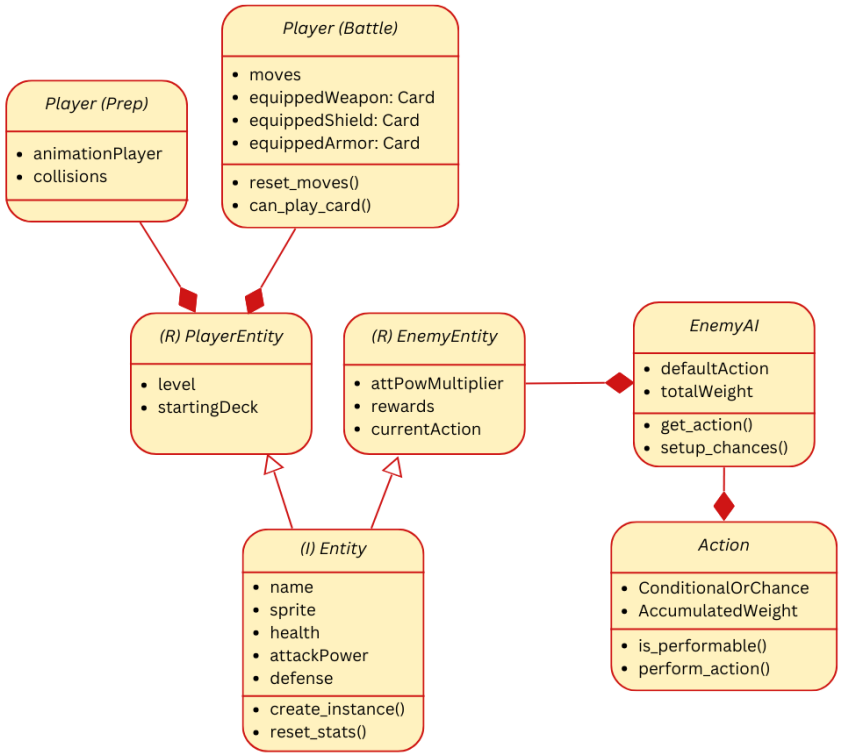

Figura 5.12: Schema UML delle Entities

5.6 Entity: Player e Enemy

Player ed Enemy sono oggetti di tipo Resource che ereditano da un'interfaccia di risorsa definita come **Entity**. Per ogni Entity sono definiti nome, punti vita, attacco, difesa e immagine di sprite, e per ogni attributo sono presenti funzioni di get() e set() per aggiornare le relative informazioni. Poichè le Entities hanno necessità di essere create su misura in base alla situazione e al momento di gioco, viene definita una funzione **create_instance()** con la quale è possibile istanziare una certa Entity con determinati valori iniziali: per esempio, un PlayerEntity implementa questa funzione per reinizializzare le informazioni del giocatore ogni volta che inizia una nuova partita, riportando la vita al valore di base e riassegnando il deck iniziale senza modifiche. Viene definita anche la funzione **reset_stats()**, in quanto è necessario resettare le sole statistiche di attacco e difesa all'inizio di ogni nuovo livello, evitando quindi di perdere le modifiche alla vita e al deck svolte nel corso della partita.

PlayerEntity estende la classe Entity aggiungendo le informazioni sul livello e sulle carte che il player ha a disposizione all'inizio di una partita, e viene poi approfondito tramite *Decorator Pattern* in due diversi oggetti, necessari a distinguere l'entità di gioco che un utente muove all'interno del dungeon durante la fase di preparazione e l'entità che partecipa alla lotta con un certo numero di mosse.

- **PlayerPrep** "decora" la risorsa del giocatore con uno *Sprite2D*, che prende in input un foglio di vari frame di animazione del personaggio, e un *AnimationPlayer*, un nodo di Godot che permette di usare i frame impostati nello sprite per creare varie animazioni in base agli input di gioco. L'assistente di animazione fornito da Godot in basso nell'interfaccia dell'Engine (fig. 5.13) è particolarmente profondo nelle sue funzioni: lo sviluppatore può creare diverse animazioni inserendo i frame all'interno della timeline e impostando parametri come gli fps o il looping; le transizioni fra varie animazioni, il loro inizio e la loro fine possono invece essere impostate tramite un sistema a grafo che gestisce in automatico tali transizioni. Ad ogni input o combinazione di input da pad direzionale viene associata una certa animazione; nell'interfaccia dell'Engine è possibile mappare tali input e scegliere quando

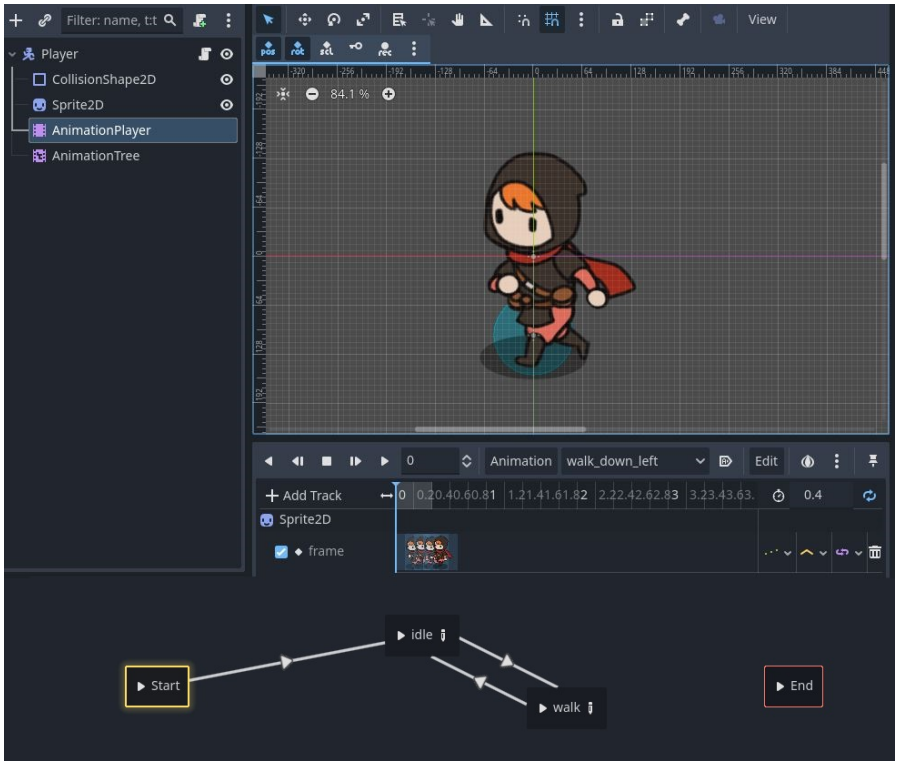

Figura 5.13: Tool di animazione dell'interfaccia di Godot. Il cerchio blu ai piedi dello sprite del giocatore è un nodo di tipo CollisionShape2D che definisce il punto di collisione del personaggio.

compiere la transizione da un'animazione all'altra. Nella figura fig. 5.14 per esempio sono definiti sei diverse animazioni, rappresentate dai punti bianchi sul bordo della finestra, che cambiano in base alla direzione premuta sul pad direzionale della tastiera, rappresentata dalla croce blu. Quando la croce blu si avvicina a uno dei punti definiti nella mappa, l'animazione transiziona in quella associata al punto corrispondente.

- **PlayerBattle** non ha bisogno di animazioni, in quanto il giocatore è fermo durante la battaglia. Questa decorazione mantiene piuttosto le informazioni relative allo scontro, in particolare alle carte equipaggiate e alle mosse disponibili durante il turno; questi attributi hanno tutti funzioni di get() e set() per permetterne la lettura e la modifica in fase di battaglia.

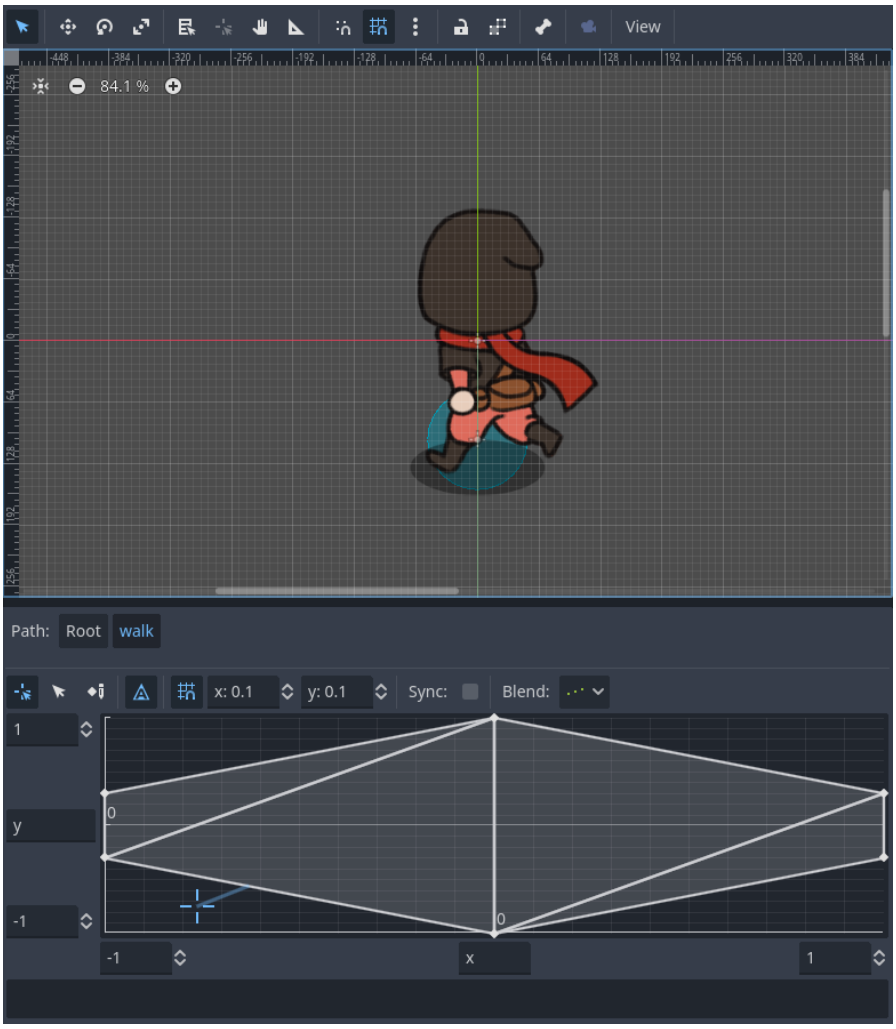

Figura 5.14: Sezione del tool di animazione per impostare la mappatura degli input. Ad ogni punto bianco che interseca i bordi del riquadro corrisponde un'animazione del player in una certa direzione.

L'EnemyEntity differisce dal Player in quanto si compone di una scena denominata **EnemyAI**, la quale raccoglie in sè vari nodi figli di tipo **Action**. Per ogni nemico viene quindi definita una IA con un certo numero di Action, e per ogni Action va definito un effetto da eseguire tramite funzione **perform_action()**. Le varie azioni sono scelte in modo perlopiù casuale tramite la funzione **get_action()** di EnemyAI, scegliendo quindi una Action tra i nodi figli della scena: la funzione get_action() fa un controllo di tipo tra le varie azioni, che possono essere condizionali o meno. Le azioni condizionali hanno priorità sulle altre: esse implementano la funzione **is_performable()** che restituisce un valore booleano uguale a true qualora le condizioni per essere eseguita fossero soddisfatte: per esempio l'azione di cura da parte del nemico è possibile solo quando il nemico raggiunge la metà dei suoi punti vita, ammesso che non sia già stata eseguita in precedenza. In caso di mancata rilevazione di una azione condizionale, viene scelta una Action di tipo Chance (sezione 5.6): per ogni Chance Action è definito un peso, così da poter specificare le probabilità che essa venga scelta, dopodichè EnemyAI genera un numero casuale tra zero e il peso totale delle chance per ottenere la mossa corrispondente ([Sar20]). La funzione get_action() viene usata all'interno dell'EnemyTurnHandler ad ogni reset delle azioni del nemico, conservando la prossima azione da eseguire nell'attributo **currentAction** di EnemyEntity.

Per questa versione di gioco, tra le azioni del nemico sono presenti un potenziamento dell'attacco e uno della difesa. Entrambi gli effetti implementano la funzione is_performable() al fine di limitare il numero di buff massimi che il nemico può effettuare. Raggiunto il limite, quindi se il nemico non può effettuare quella specifica azione di potenziamento, get_action() restituisce l'azione di default impostata sul semplice attacco. L'azione di attacco invece esegue un danno al personaggio del giocatore pari al valore statico assegnato all'azione sommato al moltiplicatore dei danni mantenuto all'interno di EnemyEntity.

5.7 Carte di gioco, Deck ed Effetti

Nel progetto viene definita un'iterfaccia di risorsa di tipo **Card**, la quale mantiene le informazioni per ogni carta di gioco come id, costo in mosse, immagine e

```
func setup_chances() -> void:
  for action: EnemyAction in get_children():
    if not action or action.type != EnemyAction.Type.CHANCE_BASED:
      continue

    total_weight += action.chance_weight
    action.accumulated_weight = total_weight

func get_chance_action() -> EnemyAction:
  var roll := randf_range(0.0, total_weight)

  for action: EnemyAction in get_children():
    if not action or action.type != EnemyAction.Type.CHANCE_BASED:
      continue
    if action.accumulated_weight > roll && action.is_performable():
      return action
  return defaultAction
```

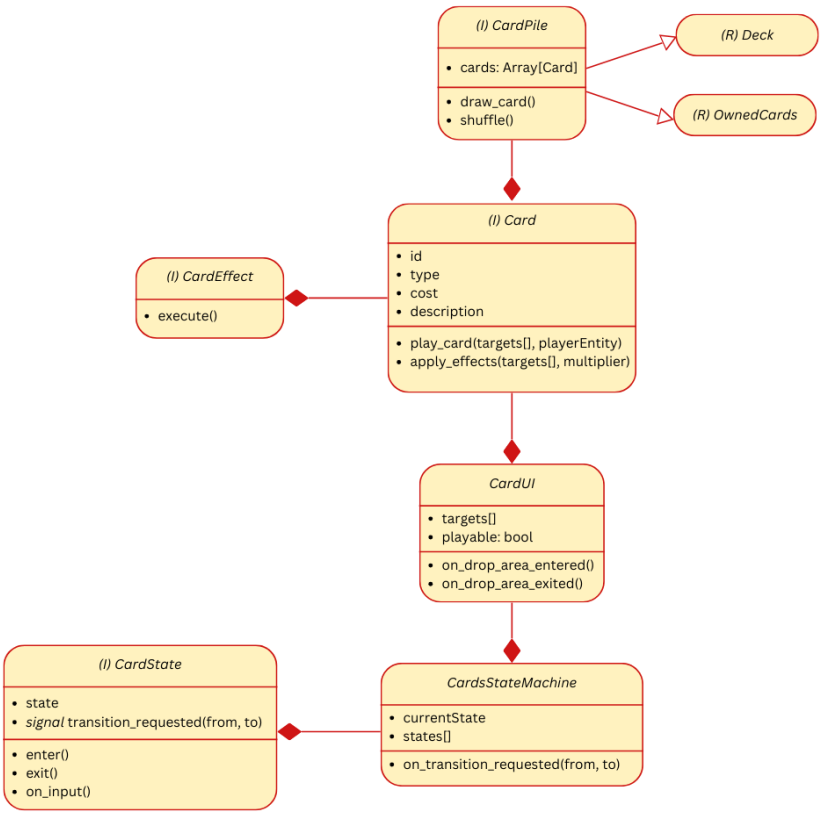

Figura 5.15: Schema UML delle Carte e dei nodi associati

descrizione. Nella risorsa è definita la funzione **play(targets[], playerEntity)** che prende in input un array di possibili obiettivi su cui applicare l'effetto della carta (che siano il giocatore o i nemici), e la risorsa PlayerEntity per poter correttamente aggiornare il numero di mosse rimaste per il turno in cui la carta viene usata; nella stessa funzione viene poi chiamato il metodo **apply_effect(targets[], multiplier)** che si occupa di eseguire la funzione **execute()** del rispettivo **CardEffect** implementato e associato alla carta in uso.
CardEffect è un'ulteriore interfaccia di risorsa, e da essa ereditano i vari effetti di gioco come l'aumento dell'attacco, della difesa, il recupero della vita o il semplice attacco all'avversario. Gli effetti vengono quindi applicati tramite il richiamo delle funzioni set() dei rispettivi valori da alterare per ogni target passato alla funzione execute(); se previsto, viene usato il valore *multiplier* passato insieme all'array di target per eventuali applicazioni di buff (ad esempio, un attacco al nemico viene applicato considerando il buff in attacco del giocatore).
Le carte sono raccolte in risorse di tipo **CardPile**, che rappresentano mazzi di carte da implementare per il deck e per le carte possedute e non in uso. CardPile definisce le funzioni di tipo **shuffle()** per mescolare le carte nella lista, e **draw_card()** per ottenere la prima carta in lista del mazzo.

Le risorse di carte non possono tuttavia essere usate così come sono: per essere interagibili all'interno del gioco devono essere avvolte in una scena, in questo caso denominata **CardUI**. Questa scena definisce il layout della carta a livello visivo, quindi assegnando un'immagine di base, lo sprite dell'oggetto definito e il costo in mosse, e ad ogni CardUI viene poi assegnato un **CardStateMachine**. Lo State Machine è un pattern di tipo comportamentale definito dal GoF, che permette ad un oggetto di alterare il proprio comportamento in base allo stato che gli viene assegnato; in Godot, questo pattern è implementabile tramite i signal, preparando una scena ricca di figli di tipo **CardState** ognuno con un signal di tipo **transition_requested**, una funzione di tipo **enter()** e una di tipo **exit()**.
Ogni carta parte dallo stato di base *HandState*. Quando in questo stato, se la CardUI dovesse rilevare un input da mouse *event.is_action_pressed("left_click")* tramite funzione **on_input()**, viene emesso il segnale transition_requested che richiama la funzione enter() dello stato *ClickedState*. Da questo stato può transizionare,

sempre tramite signal, al *DraggingState*, uno stato più complesso che si occupa di gestire le collisioni fra la CardUI e le varie zone di rilascio di equipaggiamento o di attacco al nemico. Trascinando una carta sullo schermo, ogni collisione rilevata con una certa area di rilascio risulta in un salvataggio della stessa in un array **targets**[]; l'area di rilascio viene rimossa dall'array se la collisione non è più rilevata. Al rilascio effettivo della carta (*event.is_action_released("left_click")*) in una delle zone, la carta viene rimossa dall'UI di gioco, viene eseguito il suo metodo play(targets[]) corrispondente e i dati vengono aggiornati di conseguenza in base alle azioni eseguite. Qualora la zona di rilascio non fosse quella corretta, o una carta venisse rilasciata in un punto diverso, questa viene rimandata allo stato *HandState* e riposizionata all'interno della mano del giocatore, dunque non viene utilizzata.

5.8 Salvataggio della partita

Come precedentemente accennato nella motivazione dell'uso del pattern Singleton in Godot, l'Engine non salva le modifiche alle risorse da un punto di vista di file di sitema durante l'esecuzione di un'applicazione. Per questo motivo, Godot mette a disposizione due classi: *ResourceSaver*, con la quale poter scrivere dati nella directory *user://*, e *ResourceLoader*, che permette di leggere un file e caricare le sue informazioni all'interno del codice.
I dati di salvataggio del gioco vengono raccolti all'interno di una Resource **SaveGame**, nella quale sono definiti i metodi **write_save_game()** e **load_save_game()**. La scrittura dei dati si limita a chiamare la funzione save() di ResourceSaver, che accetta come parametri la risorsa SaveGame e la directory in cui salvare il file; la lettura esegue invece la funzione load() di ResourceLoader, ma solo dopo aver verificato l'esistenza di un file SaveGame nella directory specificata tramite metodo exists(). Se quest'ultima funzione dovesse restituire un valore negativo, SaveGame si limita a impostare il proprio attributo *SaveExists* a false, restituendo una nuova risorsa di tipo SaveGame (sezione 5.8). Caricare una partita precedente significa istanziare un dungeon in fase di preparazione con le informazioni estratte dal SaveGame; verranno scartate dunque le informazioni relative ad eventuali battaglie interrotte durante il salvataggio che si sta caricando.

```
func write_save_game() -> void:
  if playerData.level > record:
    record = playerData.level
  ResourceSaver.save(self, SAVE_GAME_PATH)

func load_save_game() -> Resource:
  if ResourceLoader.exists(SAVE_GAME_PATH):
    var save_data = ResourceLoader.load(SAVE_GAME_PATH, "", ResourceLoader.
        CACHE_MODE_IGNORE).duplicate(true)
    saveExists = true
    playerData = save_data.playerData
    map = save_data.map
    return self

  else:
    saveExists = false
    return SaveGame.new()
```

Qualora si preferisse utilizzare altri metodi di salvataggio, Godot mette a disposizione varie funzioni di parsing per la creazione di documenti JSON o anche file binari, oltre che alla possibilità di combinare le tre tipologie di documento. Per il progetto Cards Against the Dungeon si è preferito sfruttare al meglio la flessibilità della classe Resource che permette la nidificazione di ulteriori Resources, come ad esempio PlayerEntity che si compone di più risorse CardPile.

Capitolo 6

Valutazione

6.1 Debugging, Test e Problemi Riscontrati

Godot è un Engine molto flessibile in fase di debug: l'IDE permette infatti di avviare in modo indipendente ogni singola scena così da poter isolare i test da eseguire su ogni elemento di gioco, mettendo a disposizione un sempre attivo Inspector che si aggiorna in tempo reale durante l'esecuzione dell'applicazione.
Fra le funzioni di Godot è presente la possibilità di assegnare la parola chiave *@export* ad un qualunque attributo in qualunque scena; usare questa annotazione rende visibile l'attributo nell'Inspector, agevolando la fase di debug: in corso di implementazione, ogni scena ha previsto l'utilizzo di dati e risorse creati su misura, assegnati tramite Inspector, eventualmente anche a runtime. Dall'Inspector, ogni attributo esportato può essere cliccato e visualizzato, permettendo allo sviluppatore una facile navigazione tra gli elementi di gioco.

Nella zona inferiore dell'interfaccia di Godot è presente una finestra che si compone di un Output e di un Debugger; durante l'implementazione sono stati eseguiti dei test di corretta esecuzione delle varie funzioni tramite delle stampe nella finestra di Output, mentre gli input sono stati verificati tramite la finestra di Debugger che rileva ogni singolo input e quale elemento ha registrato la sua esecuzione, consentendo ulteriori navigazioni agli elementi risultanti. L'Engine ha dunque permesso una modalità di testing puramente interattiva a runtime, seb-

bene lo sviluppo ha comunque previsto l'esecuzione di varie funzioni statiche di output per verificare la correttezza di calcoli e logica comportamentale.

La struttura ad albero e la creazione di nodi per ogni scena è sicuramente una pratica intuitiva e agevola lo sviluppo di videogiochi per chi ha poca pratica in materia, tuttavia presenta problemi di duplicazione in casi di scene che hanno necessità di essere diversificate anche se dotate di logica simile. Nel caso dell'applicazione implementata ai fini di questa dissertazione, la classe CardUI ha previsto una doppia implementazione di scena per distinguere le carte da usare nel menu di modifica del deck da quelle da usare durante la battaglia: qualora si tentasse di usare la stessa implementazione, si riscontrano problemi di conflitto di input, in particolare nel caso dell'implementazione dello stato di dragging delle carte. Le Resources possono essere usate più volte, ma lo stesso non è vero per Scene più complesse composte da vari nodi con specifici comportamenti legati alla scena corrente.

6.2 Valutazione

Godot non è un Engine infallibile: sebbene sia un motore particolarmente leggero, in caso di un numero elevato di informazioni da elaborare potrebbe riscontrare cali di frequenza, e quindi rallentamenti, all'interno dell'IDE stesso, vedendo spesso dei freeze che richiedono il riavvio dell'Engine. Godot è piuttosto consigliato per videogiochi con modelli e texture di piccole dimensioni, e immagini più grandi possono pesare sulla CPU del computer e rallentare l'esecuzione dell'applicazione in fase di sviluppo e debug.

Sono inoltre possibili diversi bug di IDE, come ad esempio avviene in alcuni casi di esecuzione della funzione queue_free(): tale funzione libera un oggetto non appena possibile, dunque appena eventuali funzioni ancora in corso terminano la loro esecuzione. Liberare un oggetto non corrisponde ad eguagliare lo stesso a un valore di tipo null. Queste situazioni causano errori di sincronismo, comparazione e assegnamento improprio di tali oggetti liberati, generando in alcuni casi errore di tipo *CALL_ERROR_INVALID_ARGUMENT* se usato come parametro di una

funzione statica, ma ignorando completamente l'errore qualora fosse usato come variabile di ritorno statica di una funzione ([Lnx23]).

Capitolo 7

Conclusioni

7.1 Espansioni o miglioramenti per il futuro

L'applicazione creata in visione di questa esamina è stata sviluppata tenendo in considerazione eventuali espansioni e miglioramenti. In particolare, l'EnemyTurnHandler è stato pensato per gestire molteplici nemici: la sua funzione *start_turn()* comincia con il turno del primo EnemyEntity scovato tra i nodi figli dell'Handler, e al suo termine va alla ricerca del prossimo figlio in lista, qualora esistente, così da poter avviare il suo turno. Il signal on_all_enemies_died viene emesso solo quando l'Handler non ha più nodi figli, gestendo correttamente la sequenza di vittoria del giocatore. Poste queste basi, in futuro sarà possibile aggiungere rapidamente più nemici in un dungeon partendo dalla scena BattleState, la quale dovrà però essere modificata per accogliere una visuale delle statistiche più profonda e accomodare il numero maggiore di entità in gioco.

La grafica di gioco è anch'essa piuttosto modificabile e aggiornabile; le immagini e i file possono essere sostituiti, senza ripercussioni, direttamente nei file di gioco, in quanto l'Inspector dell'Engine ha cura di assegnare automaticamente le risorse specificate in precedenza, anche in caso di modifica e refactor. Temi, Font, Layout di base possono essere creati, assegnati e modificati dalle impostazioni del progetto e dall'Inspector, con la possibilità di applicare lo stesso tema a tutti i nodi compatibili presenti nel progetto ed evitare di dover tornare manualmente su

ogni elemento.

Le risorse definite per il progetto inoltre possono essere infinitamente espanse grazie alla definizione di interfacce da cui estendere: effetti delle carte, nuove carte, nuove mappe e nuovi nemici possono essere inseriti nei file di sistema e aggiunti senza dover modificare il codice già scritto in precedenza. Persino la risorsa del giocatore può essere cambiata in fase di inizio gioco, consentendo una possibile scelta estetica del personaggio che si vuole giocare.

7.2 Godot: Limiti dell'engine e considerazioni finali

Lavorare con Godot è un'esperienza di tipo lineare; una volta definito un certo itinerario da seguire, è possibile programmare senza troppi ostacoli e barriere tecniche sia tramite approccio top-down che bottom-up. Il grande vantaggio di questo Engine è la chiarezza dei suoi elementi, con una documentazione interna esaustiva ma comunque approfondibile nella documentazione esterna ufficiale. Per chi volesse approcciarsi alla programmazione e usare i videogiochi come medium artistico, Godot è un ottimo strumento che permette una facile integrazione di asset grafici tramite l'Inspector, come ad esempio avviene per i nodi di tipo *TextureRect* che consentono l'assegnamento di un'immagine tramite un banale drag-and-drop del file scelto, la creazione di animazioni direttamente all'interno dell'IDE come precedentemente visto in fig. 5.13, ed infine la creazione di temi e stili come risorse di tipo *Theme* che possono essere create internamente all'Inspector e assegnabili ai vari nodi di controllo come contenitori, panel, label, ecc...
Godot è ben bilanciato tra sezioni dense di codice e sezioni che non ne prevedono affatto; in caso di necessità, è comunque possibile scaricare librerie e Asset, anche dall'IDE stesso tramite la sezione *AssetLib* in alto all'interfaccia, che forniscono funzioni pre-implementate. È comunque premura dello sviluppatore comprendere il funzionamento dei tool di Godot, come i fondamentali signal e le Resources. In caso di difficoltà e in assenza di un team, l'utente di Godot ha sempre a disposi-

zione una community vasta e disponibile ad aiutare chi è alle prime armi.

Godot ha i suoi limiti, che siano di grafica o di fisica, o anche solo di programmazione come visto in fase di valutazione. Al momento della scrittura di questa dissertazione, l'opinione più diffusa nei vari forum e social network è quella di usare Godot come Engine 2D, e preferire motori più ricchi, come Unity e Unreal Engine, per la programmazione 3D. Inoltre, non è attualmente possibile esportare un gioco programmato in Godot che sia compatibile con piattaforme quali Nintendo e PlayStation, con la conseguenza di dover convertire manualmente l'intero codice o dover affidare la conversione ad agenti di terze parti dedicati qualora si volesse pubblicare un gioco su Console. Godot rimane comunque un Engine in continuo aggiornamento; nel forum di github, così come nei forum della documentazione esterna e nei vari social network, i developer del motore di gioco sono piuttosto attivi e responsivi alle varie segnalazioni di bug o richieste di funzioni.
Sono inoltre già in fase di sviluppo aggiornamenti di Engine che prevedono una rifinitura del sistema di rendering, in particolare per la compatibilità con le API grafiche Metal di Apple per un migliore output grafico su device iOS e macOS, oppure permettere agli utenti di creare i propri sistemi di rendering personalizzati da integrare come risorse all'interno del codice ([Joh24]). Tutti gli aggiornamenti e servizi offerti da Godot rimangono infine completamente gratuiti, permettendo a chiunque di cominciare a programmare e dar vita alle proprie opere in qualunque momento, con annessa la possibilità di estendere direttamente il motore di gioco, data la sua natura open source, e personalizzare ulteriormente l'esperienza di sviluppo.

Bibliografia

[And15] A. Andrade. Game engines: a survey. *EAI Endorsed Transactions on Serious Games*, 2(6), 11 2015.

[Bit04] Matteo Bittanti. *Per una cultura del videogames. Teorie e prassi del videogiocare*. Matteo Bittanti, 2004.

[Cap15] Roberto Cappai. Artgame. il videogame come medium artistico., 2015.

[Che18] Mehdi Cheracher. Design Patterns: Event Bus, 2018.

[CM07] A. Clarke and G. Mitchell. *Videogames and Art*. Intellect Books - European Communication Research and Education Association Series. Intellect, 2007.

[Con23] Game Developer Conference. 2023 State of the Game Industry report resource, 2023.

[EGV94] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Etc17] Ignacio Roldán Etcheverry. Introducing c# in godot, 2017.

[Gam25] Epic Games. Blueprints Visual Scripting overview, 2025.

[Hol24] Julian Holfeld. On the relevance of the godot engine in the indie game development industry, 2024.

[Jen00] Henry Jenkins. Art Form for the Digital Age, 2000.

[Jes14] Schell Jesse. *The Art of Game Design: A Book of Lenses*. A. K. Peters, Ltd., USA, 2nd edition, 2014.

[Joh24] Clay John. Godot Rendering Priorities: January 2024, 2024.

[Lin20] Juan Linietsky. Godot Engine was awarded an Epic MegaGrant, 2020.

[LM25a] Juan Linietsky and Ariel Manzur. Godot Engine about the assets library, 2025.

[LM25b] Juan Linietsky and Ariel Manzur. Godot Engine list of features, 2025.

[LM25c] Juan Linietsky and Ariel Manzur. Idle and Physics Processing, 2025.

[LM25d] Juan Linietsky and Ariel Manzur. Singletons (Autoload), 2025.

[Lnx23] LnxPaw. Runtime error when passing a previously freed Object as a statically typed function argument [GDScript], 2023.

[Lov21] Nathan Lovato. The Event Bus Singleton, 2021.

[Sar20] Yuri Sarudiansky. Weighted Random Selection With Godot, 2020.