



UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Imparare la Programmazione Java Divertendosi: Sviluppo Basato sul Game Design di un Applied Game

RELATORE

Prof. **Fabio Palomba**

CORRELATORI

Dott. **Stefano Lambiase**

Dott.ssa **Giusy Annunziata**

Università degli Studi di Salerno

CANDIDATO

Alessio Piro

Matricola: 0512105782

“Don’t give up on your dreams, or your dreams will give up on you.”

-John Wooden

Sommario

Negli ultimi anni, coinvolgere e motivare gli studenti rappresenta una sfida cruciale, riconosciuta anche dalle Nazioni Unite che hanno inserito la qualità dell'educazione tra i "Sustainable Development Goals". In tale scenario, la gamification appare come una delle migliori soluzioni per affrontare questa situazione.

Nonostante numerose ricerche e studi abbiano già reso noto quanto la gamification possa essere un mezzo rivoluzionario nel campo dell'educazione grazie al suo potere motivazionale, il numero di prodotti educativi validi è limitato.

Questa tesi mira a identificare perché la maggior parte dei videogiochi educativi non riesce a motivare gli studenti come previsto. Successivamente, utilizzando tali nozioni, sviluppa un prototipo di Applied Game con l'obiettivo di coinvolgere gli adolescenti nel mondo della programmazione Java. Tale software educativo ha l'obiettivo di mettere in pratica le nozioni apprese nella fase di studio, fornendo agli interessati del settore un esempio di approccio efficace al mondo del Game-Based Learning.

Per comprendere come procedere nello sviluppo di tale progetto sono stati studiati numerosi documenti riguardanti il "Game-Based Learning" e sono stati analizzati vari giochi educativi al fine di identificarne pregi e difetti. Da tali approfondimenti è stato possibile apprendere che l'errore commesso dalla maggior parte degli Applied Games è il medesimo: concentrarsi sulle nozioni da trasmettere trascurando la cura delle funzioni che renderebbero il gioco divertente; In altre parole, la poca cura del Game Design.

In conclusione, questa tesi si concentra sull'uso efficace del videogioco come strumento educativo, ponendo l'accento sulla creazione di un Applied Game che renda avvincente l'apprendimento dei concetti fondamentali della programmazione Java, anche per coloro che si avvicinano a questo mondo per la prima volta. Tale obiettivo è raggiungibile integrando all'interno delle meccaniche di gioco i concetti base della programmazione Java, evitando spiegazioni prolisse e concentrandosi sullo sviluppo di funzioni che possano guidare il giocatore senza restituirci la sensazione di partecipare ad una lezione "alternativa".

Indice

1	Introduzione	3
1.1	Motivazioni e Obiettivi	3
1.2	Risultati	4
1.3	Struttura della Tesi	4
2	Stato dell'arte	6
2.1	Background	6
2.1.1	Gamification	7
2.1.2	Game-Based Learning	8
2.1.3	Perché i videogiochi sono adatti all'apprendimento	9
2.1.4	La motivazione dei giochi educativi	9
2.1.5	Il problema delle ricerche sul Game-Based Learning	11
2.1.6	Elementi di Game Design per il Game-Based Learning	12
2.2	Related Works	13
2.2.1	Robocode	13
2.2.2	Rabbids: Coding!	16
2.2.3	Outcore: Desktop Adventure	19
3	Metodologia	23
3.1	Obiettivi	23
3.2	La Struttura del Gioco	23
3.2.1	La Scelta della Piattaforma	24
3.2.2	La Scelta del Genere di Videogioco	24
3.2.3	Il Gameplay Loop	25
3.2.4	La Selezione del Livello	26
3.2.5	La Mappa	26
3.2.6	La Battaglia	28
3.2.7	Il Luna's	30
3.2.8	Imprevisti e Tesori	35
3.3	Tecnologie Utilizzate	35
3.3.1	La Scelta del Game Engine	35
3.3.2	Altri Software	37
3.4	Concetti Base di Godot Engine	38
3.4.1	Le Scene	38
3.4.2	I Nodi	38
3.4.3	GDScript	39
3.4.4	L'Albero della Scena	39

3.4.5	Le Risorse	39
3.4.6	I Segnali	40
3.5	Architettura dell'Applied-Game	41
3.5.1	Scene Principali	41
3.5.2	Passaggio di Scena	41
3.5.3	Gestione della Mappa	43
3.5.4	Generazione della Descrizione del Metodo	45
3.6	Segnali e Componenti	47
3.7	Dati Statici e Salvataggi	50
3.7.1	Gestione dei Dati Statici	50
3.7.2	Gestione dei Salvataggi	50
4	Risultati	52
4.1	Robocode	52
4.2	Rabbids: Coding!	54
4.3	Outcore: Desktop Adventure	55
5	Conclusioni	57
5.1	Risultati Raggiunti	57
5.2	Sviluppi futuri	58
Ringraziamenti		61

Capitolo 1 - Introduzione

1.1 Motivazioni e Obiettivi

Durante il mio percorso educativo, fin dai primi anni della scuola primaria, ho manifestato un notevole interesse per le discipline legate all'informatica. Questo interesse ha acquisito concretezza fino alla scuola superiore, quando ho avuto il mio primo incontro con la programmazione. Qui, un'insegnante che proponeva obiettivi troppo ambiziosi e un ambiente poco stimolante hanno contribuito a far scemare il mio entusiasmo verso una materia che, agli occhi miei e dei miei compagni, appariva eccessivamente complessa.

Dopo due anni, ho deciso di cambiare indirizzo di studi e ho avuto la fortuna di incontrare una docente che aveva un approccio diverso nell'insegnare la programmazione. Grazie a lei, il mio interesse è stato risvegliato e da allora non si è mai più affievolito. Ho avuto la fortuna di trovare qualcuno durante il mio percorso educativo che mi ha fatto apprezzare la programmazione, ma ho notato che molti altri studenti non hanno avuto la stessa fortuna.

L'informatica, specialmente agli occhi dei giovani o di chi è estraneo a questo campo, può apparire estremamente complessa. Il primo approccio può creare un muro che risulta difficile da abbattere in futuro. Pertanto, mi propongo di avvicinare i giovani al mondo della programmazione fin da subito, consentendo loro di familiarizzare con le basi di Java. Questo approccio mira a evitare che si sentano spaventati o scoraggiati durante il loro percorso scolastico e, se possibile, a suscitare passione anche in coloro che non hanno alcun interesse iniziale per questa materia che amo.

Si è quindi avviata la ricerca di un approccio che consentisse di introdurre nuove persone al mondo della programmazione in maniera gradevole, rendendo il percorso di apprendimento un'esperienza divertente anche per coloro che potessero provare una certa avversione verso questa materia. Tali ricerche hanno portato alla scoperta del concetto di Gamification.

Successivamente, esplorando questo campo, si è scoperto come molti esperimenti siano risultati fallimentari. Perfino un'azienda di rilievo come Nintendo, con titoli come "Mario is Missing", "Mario's Time Machine" e "Mario's Early Years!", non riuscivano a creare giochi educativi che fossero allo stesso tempo divertenti. Le recensioni spesso etichettavano questi titoli come "noiosi" e "pesanti".

Da qui è quindi partito lo sviluppo del progetto "Project Coding", un applied game nato per avvicinare le persone, anche quelle con un interesse limitato verso l'argomento, al mondo della programmazione Java.

Inizialmente, è stato necessario studiare ed analizzare articoli e videogiochi nell'am-

bito del Game Based Learning per comprendere le motivazioni dietro ai fallimenti di molti prodotti del settore. Successivamente, sono state utilizzate le competenze apprese per sviluppare un prototipo di applied game che potesse servire da esempio per i futuri sviluppatori su come creare un gioco educativo di successo, facilitando l'apprendimento e rendendolo piacevole.

1.2 Risultati

Alla fine dello sviluppo, è stata ottenuta una prima versione di Project Coding, il cui codice è presente nell'apposita repository di GitHub [5]. Si tratta di un prototipo, una prima versione dell'applied game che però incorpora in sé i concetti fondamentali dietro la sua creazione. Le meccaniche di gioco, infatti, derivano proprio dai concetti base di Java e sono state studiate per essere facilmente comprese senza necessariamente dover leggere lunghe spiegazioni.

Più precisamente, il gioco è composto da varie sezioni, le cui più importanti sono: la sezione legata alla programmazione e la sezione legata alla battaglia.

Nella sezione legata alla programmazione, il giocatore ha la possibilità di programmare le proprie mosse usando costrutti e classi che gli permettano di definirne il comportamento. Tali mosse possono poi essere utilizzate nella sezione della battaglia per sconfiggere i propri nemici e continuare la partita.

La fase di programmazione è resa estremamente più intuitiva da due fattori:

- Il codice vero e proprio è sostituito da blocchi logici rappresentati come tasselli di puzzle. In questo modo, classi e costrutti sono più facilmente gestibili e riconoscibili dal giocatore, che potrà comunque leggere sempre il codice corrispondente a ciascun elemento in quanto impresso sul tassello stesso;
- Nella schermata di programmazione del metodo è presente una descrizione che viene aggiornata ad ogni modifica effettuata dal giocatore e che gli permette di comprendere rapidamente gli effetti del codice che ha costruito.

Grazie a queste funzioni, anche chi non ha la minima conoscenza del mondo Java può provare a programmare senza dover consultare lunghe guide o noiosi tutorial, ed apprendere il ruolo di ogni tassello partita dopo partita. In più, il pensiero che una buona programmazione possa facilitare gli scontri contro i nemici in un secondo momento, fornisce al giocatore una forte motivazione ad impegnarsi nella costruzione del metodo.

1.3 Struttura della Tesi

La tesi inizia con una breve introduzione, contenuta nel paragrafo appena concluso, esponendo quali sono state le motivazioni che hanno portato alla realizzazione

di questo progetto, quali obiettivi erano stati prefissati e quali risultati sono stati raggiunti. Successivamente, nel capitolo dedicato allo stato dell'arte, vengono illustrati dapprima gli studi che hanno permesso di identificare i principali aspetti del Game-based Learning, per poi analizzare tre videogiochi educativi già esistenti che hanno lo scopo di insegnare competenze informatiche ai giocatori. Si procede con il capitolo dedicato alla metodologia, in cui vengono specificati gli obiettivi del progetto, viene approfondita la struttura del gioco e vengono affrontati argomenti di natura più tecnica che si sono rivelati centrali durante lo sviluppo dell'applied game. Il quarto capitolo si concentra sulle caratteristiche che Project Coding ha ereditato dallo studio dei “related works” analizzati nel capitolo 2. Infine, nell’ultimo capitolo vengono analizzati ancora più a fondo quali risultati sono stati raggiunti e, partendo da questi, quali potrebbero essere le funzionalità da aggiungere per lo sviluppo di un applied game completo.

Capitolo 2 - Stato dell'arte

Questo capitolo illustra i concetti base riguardanti il game-based learning e gli applied games, illustrandone le origini, spiegando perché possono essere rivoluzionari per il mondo dell'apprendimento e mostrando quali sono gli aspetti fondamentali che rendono efficace un videogioco educativo.

In seguito, analizza pregi e difetti di alcuni esempi di applied games votati all'insegnamento delle basi della programmazione presenti sul mercato, selezionati sia per la loro importanza storica che per il loro modo di innovare un settore che, come vedremo, non è mai riuscito ad essere incisivo come molti studiosi si aspettavano anni addietro.

2.1 Background

Ormai da anni, il videogioco è diventato un fenomeno di massa estremamente popolare. Bambini, ragazzi ed adulti di tutto il mondo spendono ore immergendosi in mondi virtuali a cui accedono usando computer, console o anche semplicemente i loro smartphone.

Il media *videoludico* è sicuramente molto diverso da qualunque altro: è l'unico a possedere un grado di interattività così elevato che chi ne fruisce, se dinanzi ad un prodotto di qualità, viene "assorbito" completamente dal mondo di gioco ignorando ciò che ha intorno. È importante sottolineare che un gioco, per avere gli effetti sopracitati, deve essere "di qualità" in quanto sviluppare software di intrattenimento che risultino gradevoli a chi ne fa uso non è sfida da poco. Il numero di fallimenti nel campo videoludico è decisamente elevato e supera di gran lunga quello dei successi. I motivi sono molteplici ma, per la maggioranza dei casi, sono categorizzabili come "errori di game design".

Il *Game Design* è definibile come "l'atto di decidere come un gioco debba essere" [8]. Una definizione semplice per una materia di studio estremamente complessa che, per essere affrontata, prevede conoscenze di sound design, psicologia, scrittura creativa, programmazione e molto altro. Per il momento non verrà approfondito ulteriormente l'argomento ma ciò che è fondamentale comprendere è che se durante lo sviluppo di un videogioco sono state effettuate delle corrette scelte di game design, il software sarà in grado di impegnare il giocatore per ore ed ore facendolo concentrare solo sui compiti relativi al mondo virtuale e isolandolo da qualunque stimolo esterno.

Questa abilità di motivare una persona a tal punto da riuscire a farle mantenere la concentrazione sulla risoluzione di molteplici task per lunghi periodi di tempo è diventata presto materia di studio di molti ricercatori, interessati a scoprire come

ottenere risultati simili anche in campi diversi dall'intrattenimento. Uno di questi campi è, ovviamente, l'apprendimento.

2.1.1 Gamification

Uno dei problemi principali nel campo dell'apprendimento degli ultimi anni è quello di cercare di identificare un modo per coinvolgere e motivare gli studenti nelle attività di studio [3].

La questione, col tempo, è diventata sempre più rilevante, fino al punto che le Nazioni Unite hanno incluso l'innalzamento del livello di qualità dell'educazione tra i "Sustainable Development Goals" [4]. In questo scenario, la Gamification appare come una delle migliori soluzioni a questa sfida.

Si definisce *Gamification* l'uso di approcci, meccaniche e, in generale, elementi classici dei giochi in contesti diversi da quelli ludici [3].

Un classico uso della gamification è l'inclusione, all'interno di software votati all'apprendimento, di classifiche, punteggi, ricompense ed altre classiche meccaniche dei videogiochi al fine di aumentare l'interesse e la motivazione degli studenti. Tuttavia, è necessario specificare che l'utilizzo di tecniche di gamification non implica obbligatoriamente il coinvolgimento di progetti software; esistono, ad esempio, numerosi giochi da tavolo che si pongono l'obiettivo di facilitare l'insegnamento di svariati argomenti.

Tuttavia, è importante non commettere l'errore di pensare che basti aggiungere elementi casuali provenienti dal panorama videoludico per rendere stimolante un software educativo; lo sviluppo di programmi di questo tipo richiedono non solo delle buone idee di game design ma anche una profonda conoscenza della materia che si vuole insegnare e, soprattutto, le giuste idee per amalgamare conoscenza e meccaniche di gioco [4].

Per raggiungere una corretta implementazione degli elementi di gamification in un software, è fondamentale:

- Conoscere il livello di conoscenza di chi usufruisce del software in quanto sfide troppo semplici o troppo complesse possono demotivare gli studenti minando il percorso di apprendimento;
- Chiarire fin da subito gli obiettivi di apprendimento da raggiungere per evitare che tutti gli sforzi fatti dagli studenti possano essere percepiti come vani, senza scopo;
- Creare le attività per la gamification, tenendo a mente che: gli studenti dovrebbero poter ripetere le sfide in caso di fallimento, le attività dovrebbero avere un livello di difficoltà che si adatta al livello di conoscenza dello studente, il livello di difficoltà dovrebbe crescere man mano che si completano

le sfide e che bisogna programmare le attività in modo che vi siano più modi per completarle;

- Inserire elementi di gioco come punti, classifiche, ecc. [3]

Il concetto di gamification negli anni è diventato estremamente popolare e, lentamente, questa sua espansione ha dato vita a nuovi concetti riguardanti l'impiego di videogiochi con obiettivi diversi da quelli del puro intrattenimento. Alcuni esempi rilevanti sono:

- Transformational game: Software sviluppati con l'intento di influenzare e cambiare le idee e i comportamenti del giocatore, anche e soprattutto al di fuori del gioco;
- Funware: Uso di meccaniche di gioco in contesti non ludici con l'obiettivo di incentivare alcune azioni delle persone. Viene spesso adoperato nell'ambito del marketing per instaurare un rapporto di fedeltà del cliente verso l'azienda;
- Edutainment: Forma di intrattenimento con duplice scopo: divertire ed educare;
- Applied Game: Veri e propri videogiochi sviluppati da zero con l'obiettivo di educare il giocatore;
- Game-based Learning: Processo educativo che sfrutta un videogioco per favorire l'apprendimento [6].

2.1.2 Game-Based Learning

Nel tempo sono state date varie definizioni di “*Game-Based Learning*” ma la sostanza non cambia: è il processo di apprendimento tramite l’uso di un gioco. Anche in questo caso non parliamo per forza di un videogioco dato che anche apprendere nuove conoscenze tramite giochi analogici ricade nel campo del Game-Based Learning [4].

Alcuni studi che trattano solo ed unicamente l'apprendimento tramite l'uso di videogiochi preferiscono usare il termine “*Digital Game-Based Learning*” per eliminare qualunque tipo di ambiguità [7].

Soltanente, insieme al concetto di Game-Based Learning, viene chiarita anche la definizione di “*Applied Game*”: videogiochi sviluppati interamente da zero col fine di insegnare nuove nozioni al giocatore.

È necessario puntualizzare fin da subito la differenza tra Gamification e Applied Games: mentre applicando la tecniche di gamification si inseriscono elementi classici del mondo dei videogiochi in contesti non ludici, creare Applied Games significa sviluppare videogiochi completamente nuovi con fini didattici, ponendo estrema cura nel bilanciare la componente di apprendimento e la componente di gioco [6].

2.1.3 Perché i videogiochi sono adatti all'apprendimento

Ci sono numerose caratteristiche che rendono il videogioco un mezzo potenzialmente perfetto per l'apprendimento. Alcune delle più importanti sono:

- Motivazione: Senza ombra di dubbio, la motivazione è la caratteristica principale che ha spinto numerosi studiosi a cercare un modo di utilizzare i videogiochi per scopi educativi. Il potere che il media videoludico ha dimostrato di possedere nel tenere impegnato un giocatore per ore ed ore consecutive ha ispirato numerosissimi studi che analizzano il fenomeno tramite svariati punti di vista differenti;
- Adattabilità: Un videogioco può essere programmato in modo da “adattarsi” al giocatore, migliorando così la sua esperienza. Questo può accadere quando il gioco cambia difficoltà a seconda del livello di chi lo utilizza, permette di personalizzare l’aspetto del personaggio principale, fa apparire o meno degli aiuti a schermo a seconda di quanto quella situazione stia creando problemi al giocatore, ecc. Appare ovvio che, soprattutto per videogiochi con scopi didattici, esistono meccaniche di adattabilità che risultano più importanti di altre, come la modifica del livello di difficoltà, ma anche quelle più meramente estetiche hanno la loro importanza in quanto aiutano il giocatore ad immergersi nel mondo di gioco. Da game designer quali bisogna essere per progettare videogiochi educativi, è importante valutare ogni aspetto del proprio lavoro e non concentrarsi unicamente sugli aspetti educativi, come in passato hanno fatto numerosissime aziende per poi fallire nel loro intento di creare un applied game coinvolgente;
- Graceful Failure: Il Graceful Failure è un concetto che non viene sempre implementato nei videogiochi tradizionali ma che invece risulta fondamentale in quelli educativi. Per *Graceful Failure* si intende la caratteristica di un gioco di non far percepire al giocatore il fallimento come un evento del tutto negativo, bensì come uno step utile per avvicinarsi ulteriormente all’obiettivo finale. Secondo Hoffman e Nadelson, sapere che le conseguenze di un fallimento sono minime incoraggerebbe i giocatori a prendere più rischi e ad esplorare nuove soluzioni, atteggiamenti sicuramente molto positivi che un buon software educativo dovrebbe sempre provare ad incentivare [6].

2.1.4 La motivazione dei giochi educativi

Tra i motivi per cui l’approccio del Game-Based Learning risulta in teoria più che valido, quello di motivare il giocatore ad apprendere è sicuramente il più centrale.

Praticamente ogni studio in questo campo si sofferma nello spiegare l'incredibile potere motivazionale che i giochi educativi possono avere e quindi appare necessario dedicare all'argomento un po' più di spazio.

Prima di procedere oltre, però, è necessario definire il concetto di motivazione nel contesto dell'apprendimento; Citando lo studio condotto da Huang, Huang e Tschopp, “la motivazione è l'elemento essenziale per avviare e sostenere l'apprendimento e le prestazioni”. Se non fosse già abbastanza chiaro, questa è la ragione per cui includere stimoli motivazionali all'interno di un videogioco votato all'apprendimento risulta fondamentale [7].

Che si parli di applied games o di semplici videogiochi, un concetto fondamentale che è alla base di un buon lavoro di game design è quello del flow.

Introdotto da Csikszentmihalyi nel 1975, il concetto di *flow* può essere descritto come lo stato cognitivo di una persona che, completamente concentrata nello svolgimento di un compito, non avverte stimoli e non prova emozioni che non siano correlate al compito stesso [7].

Il concetto di flow è fondamentale nel game design perché è esattamente ciò che un buon videogioco dovrebbe far provare a chi lo gioca. Questo obiettivo non è di certo semplice da raggiungere ma esistono alcuni fattori chiave da tenere in considerazione per cercare di avvicinarci il più possibile:

- Obiettivi chiari, per non far avvertire un senso di inutilità delle proprie azioni al giocatore;
- Assenza di distrazioni;
- Reazioni del gioco immediate agli input del giocatore, in modo da farlo sentire in controllo della situazione e non permettergli di distrarsi;
- Sfide continue, in modo da stimolare costantemente il giocatore.

Quest'ultimo punto è forse il più importante non che il più difficile da bilanciare all'interno di un gioco. Se una sfida è troppo semplice, il giocatore rischia di annoiarsi mentre se una sfida è troppo difficile, il giocatore potrebbe avvertire un senso di frustrazione, demoralizzarsi e abbandonare così il gioco. È per questo che bisogna sempre tenere in considerazione il livello iniziale del giocatore nelle abilità richieste e adattare costantemente la difficoltà delle sfide alla sua crescita mano a mano che avanza nel gioco facendo attenzione a non farlo scontrare con attività troppo complesse per le sue capacità [8].

Un'altra teoria relativa alla motivazione estremamente utile da conoscere per progettare un buon videogioco educativo è la “Cognitive Evaluation Theory” di Ryan e Deci che essenzialmente divide la motivazioni in due categorie:

- La *motivazione intrinseca*, che spinge una persona a svolgere azioni per interesse e divertimento;

- La *motivazione estrinseca*, che porta una persona a mettere in atto determinati comportamenti ai fini di ottenere dei risultati ben precisi (soldi, voti, premi, ecc.) [7].

La motivazione estrinseca è facile da stabilire ma piuttosto difficile da mantenere alta nel tempo mentre la motivazione intrinseca riesce a durare decisamente più a lungo ed aiuta chi la prova a raggiungere lo stato di flow.

Esistono svariati motivatori intrinseci utilizzati per rendere un videogioco accattivante come quelli scoperti da Malone e Lepper: la sfida, la fantasia, la curiosità, il controllo, la cooperazione, la collaborazione e la competizione. Questi sono solo alcuni degli elementi che possono essere utilizzati e plasmati per rendere un gioco divertente. Da questi si può partire per pensare a delle meccaniche da implementare nei propri giochi per cercare di far raggiungere lo stato di flow al giocatore ed esistono già degli studi in merito come “Designing Engaging Games for Education: A Systematic Literature Review on Game Motivators and Design Principles” di Teemu H. Laine e Renny S. N. Lindberg, che prova a fornire delle linee guida specifiche per la progettazione di videogiochi educativi elencando e spiegando dei Principi di Design appositi [4].

2.1.5 Il problema delle ricerche sul Game-Based Learning

Un discorso che è sicuramente fondamentale affrontare per comprendere a pieno il campo del Game-Based Learning è quello delle difficoltà nello studio e sviluppo di tecniche ed euristiche per progettare applied games efficaci.

I videogiochi, infatti, diversamente da come potrebbe pensare qualcuno completamente esterno al settore, non sono tutti uguali ma, al contrario, variano moltissimo uno dall'altro. Possono essere raggruppati per categorie in base alle meccaniche di gioco e le scelte di game design prese durante lo sviluppo ma anche due giochi dello stesso genere possono risultare completamente diversi e impattare in modo differente a livello emotivo sul giocatore.

Entrando più nello specifico, nel campo degli applied games non solo i giochi differiscono per genere ma anche per campo della materia da apprendere e tipologia di contenuti da imparare.

Insomma, in un mondo di possibilità così vasto come quello del videogioco con finalità educative, i risultati di una ricerca effettuata sull'uso di un applied game difficilmente possono essere validi anche per un altro, il che rende lo studio della materia decisamente complesso [6].

Inoltre, bisogna tenere conto anche che spesso questi videogiochi vengono utilizzati all'interno di classi diverse di istituti diversi. Anche solo il comportamento e le competenze dell'insegnante possono impattare positivamente o negativamente sull'esperienza complessiva.

Considerando tutti questi fattori non è così sorprendente scoprire che le ricerche nel campo del Digital Game Based Learning che si sono concentrate sulla crescita della motivazione degli studenti con l'utilizzo di giochi digitali per l'apprendimento hanno spesso portato a dei risultati inconsistenti [7].

Quindi, per quanto alcuni studi provino a fornire consigli e linee guida generali da seguire, non esiste una procedura unica e perfetta per creare applied games ed è anche per questo che, escludendo pochi successi e nonostante l'enorme potenziale, i giochi educativi hanno fallito nel raggiungere il pubblico mainstream (ma, come verrà illustrato nel paragrafo successivo, non è l'unico motivo) [1].

2.1.6 Elementi di Game Design per il Game-Based Learning

È innegabile che i videogiochi educativi, tralasciando alcune rare eccezioni, abbiano fallito nel raggiungere un pubblico ampio e vari studi sono stati svolti per capirne la ragione arrivando ad una conclusione: la maggior parte degli applied games è carente sotto il punto di vista del game design.

È stato già reso noto quanto il campo del game design sia complesso e sfaccettato; quindi, provare a riassumere in poche righe come si progetta un buon videogioco sarebbe praticamente impossibile. Tuttavia, ci sono degli elementi di questa materia a cui un designer di giochi educativi dovrebbe prestare particolare attenzione, come la rappresentazione delle informazioni.

Riuscire a rappresentare informazioni in modo chiaro e preciso è fondamentale per far sì che il giocatore le interpreti nella maniera corretta, tanto più in ambienti di apprendimento virtuali. Il media videoludico è speciale proprio per il suo grado di interattività; un giocatore può imparare nuove nozioni non solo leggendole a schermo, metodo che alle volte risulta noioso e controproducente, ma anche e soprattutto “interagendo” con rappresentazioni di queste [1]. Per esempio: in un gioco di fisica destinato ad un pubblico di bambini, il giocatore può interagire con la rappresentazione di una ruota per comprenderne il movimento e il funzionamento in maniera rapida ed intuitiva.

Una rappresentazione inaccurata delle informazioni può innescare effetti negativi significativi sull'umore del giocatore, portando a sensazioni quali confusione, smarrimento o frustrazione.

Direttamente connesso al concetto di rappresentazione delle informazioni vi sono quelli di difficoltà percepita e difficoltà attuale di un task. Prendendo in considerazione un'attività di gioco, la sua difficoltà percepita è quella avvertita dal giocatore mentre la sua difficoltà attuale è quella reale, che si riferisce a quanto sia difficile per i giocatori completare il task. Spesso, a causa di una rappresentazione errata delle informazioni, la difficoltà percepita di un task può essere più alta o più bassa rispetto a quella attuale. Quando ciò succede, gli effetti sul giocatore possono essere molto negativi e avere ripercussioni sul suo umore, minando il raggiungimento

o il mantenimento dello stato di flow oppure facendogli apprendere delle nozioni sbagliate sulla materia su cui il gioco si concentra.

L'insieme di aspetti a cui un designer di applied game dovrebbe prestare attenzione è ben più ampio di quanto riportato. Qui di seguito, verranno analizzati dei videogiochi educativi relativi all'apprendimento delle basi della programmazione per identificarne pregi e difetti, al fine di comprendere ancora meglio il campo del Game-Based Learning.

2.2 Related Works

Questo capitolo illustra e analizza alcuni applied games dedicati all'apprendimento dei principi base della programmazione già presenti sul mercato, ponendo particolare attenzione alle loro limitazioni o alle differenze con il progetto soggetto di questa tesi.

2.2.1 Robocode

Robocode è uno dei videogiochi legati all'apprendimento della programmazione più datati nonché più importanti della storia dei giochi educativi. Creato nel 2000 dal ricercatore Mat Nelson con l'obiettivo di facilitare l'apprendimento del linguaggio Java, il progetto è diventato completamente open source nel 2006 ed è ancora disponibile gratuitamente sul portale SourceForge oggi.

Come riportato anche sul file "ReadMe" del gioco, Robocode è stato molto popolare in passato, a tal punto da essere integrato nelle lezioni in scuole ed università per provare a coinvolgere maggiormente gli studenti.

Nonostante siano passati diversi anni dal suo lancio, Robocode presenta ancora una folta community che non solo continua a divertirsi creando nuovi robot da far combattere ma che aggiorna costantemente il software, i cui update vengono pubblicati ancora oggi.

Come suggerito dallo slogan del gioco, "Build the best, destroy the rest!", il concetto fondamentale di Robocode è semplice: si tratta di programmare il proprio carrarmato in Java, metterlo in competizione in un'arena contro altri carrarmati forniti dal gioco o creati da altri giocatori e cercare di distruggerli subendo il minor numero possibile di danni.

Il gameplay di questo titolo è divisibile in due macro-sezioni:

- La parte di programmazione dove bisogna modificare, tramite un editor, il codice del proprio carrarmato, descrivendone il comportamento e le reazioni alle mosse degli avversari;

- La parte dello scontro dove i carrarmati, seguendo le direttive dategli tramite istruzioni Java, si scontrano tra di loro con l'obiettivo di abbattere più avversari possibili cercando di limitare i danni ricevuti. Alla fine di questa fase, un tabellone con le statistiche e i punteggi relativi all'incontro viene visualizzato e il giocatore in cima a questa classifica viene decretato il vincitore dello scontro.

Analizzando il gameplay sotto il punto di vista del game design appare evidente che uno dei motivatori principali di Robocode è la competizione dato che l'obiettivo finale, chiaro al giocatore fin da subito, è programmare un robot che possa battere quelli degli avversari, potenzialmente programmati da altri giocatori. Tuttavia, Robocode è un titolo che vive grazie ad una community molto attiva che, tramite una wiki, un gruppo Google ed un gruppo Facebook, crea legami condividendo strategie, pezzi di codice o creando delle sfide pensate per migliorare le proprie abilità da programmatori di robot. Esistono persino delle competizioni organizzate in cui i giocatori si sfidano tra di loro per decretare chi sia il più abile.

Una community così attiva e unita promuove un forte senso di appartenenza tra i giocatori, fungendo così da importante motivatore e riuscendo a mantenere vivo l'interesse dei suoi membri per un periodo prolungato.

Non dovrebbe risultare difficile immaginare cosa sarebbe successo a Robocode, un gioco basato principalmente sulla competizione tra programmatori, se alle sue spalle non ci fosse stato un gruppo di giocatori così fedele al titolo, il che fa riflettere su quanto sia importante il concetto di “community”.

Nonostante tutti i lati positivi appena descritti però, Robocode presenta anche delle criticità piuttosto evidenti, frutto di un software che, nonostante sia ancora usato ed apprezzato, risente del peso degli anni.

Il comparto estetico di Robocode è spoglio, primitivo e decisamente poco accattivante. La rappresentazione dei carrarmati durante gli scontri può anche ritenersi funzionale ma l'interfaccia del gioco è decisamente uno dei suoi più grandi punti deboli. Quando un nuovo giocatore avvia il file eseguibile si ritrova davanti ad una finestra (Figura 2.1) occupata per la maggior parte da un rettangolo nero con al centro il logo del gioco, da alcune opzioni in alto a sinistra, e da dei pulsanti situati nella parte bassa della schermata, tra cui uno slider molto ampio la cui funzione, assolutamente non segnalata neanche da una scritta e decisamente poco intuitiva, è quella di aumentare la velocità della simulazione. Insomma, l'esperienza relativa al primo avvio di Robocode per un nuovo giocatore è così confusa e poco gradevole che la sua reazione può essere una delle seguenti:

- Il giocatore abbandona il gioco;
- Il giocatore tenta di capire come procedere;

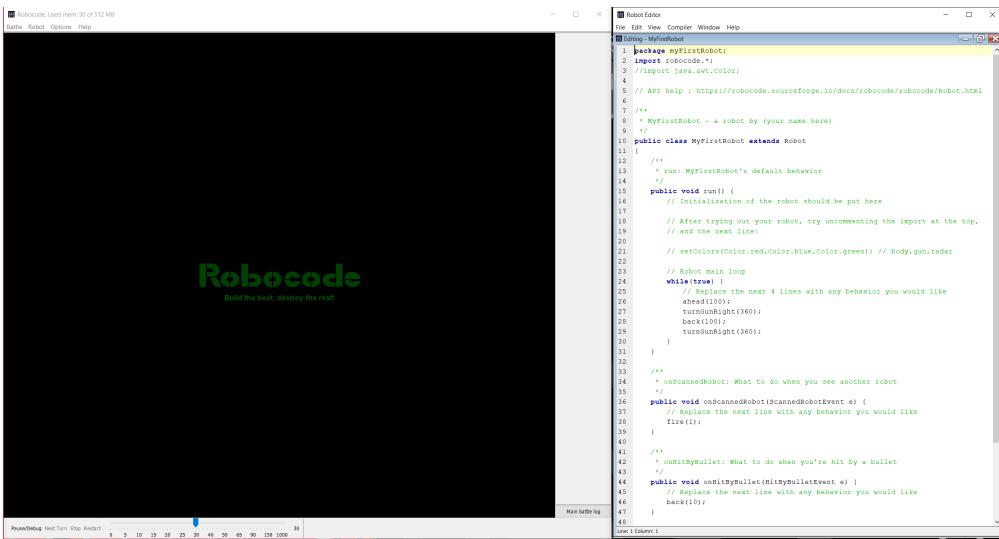


Figura 2.1: Schermata iniziale e editor di codice per la programmazione del robot di *Robocode*.

- Il giocatore cerca tutorial su internet.

Nessuna delle tre situazioni risulta particolarmente piacevole, ma la possibilità di perdere un nuovo giocatore dopo pochi secondi dall'inizio dell'esperienza rappresenta un problema enorme per un applied game.

Un altro grosso problema relativo al primo approccio con Robocode è la totale assenza di tutorial o aiuti di ogni tipo all'interno del gioco. Un nuovo giocatore che non si è documentato a sufficienza sul titolo prima di scaricarlo difficilmente può capire da solo in cosa consiste Robocode, qual è il suo obiettivo, quali sono le possibilità che il software gli mette a disposizione. L'unica sezione del videogioco un po' più guidata è quella relativa alla programmazione del bot dove dei commenti aiutano la comprensione del codice preinserito dal gioco stesso. Tuttavia, una persona che non conosce le basi della programmazione Java, senza l'aiuto di un insegnante o di tutorial esterni, non potrà fare molto data la totale assenza di guide per persone con poche conoscenze sull'argomento.

Ricapitolando, Robocode è un titolo educativo molto valido e divertente nonostante sia sul mercato da molto tempo. La quasi totale assenza di guide e i pochi aiuti disponibili per chi ha poche conoscenze di Java, suggerisce che questo videogioco punti maggiormente ad un pubblico composto da persone con delle competenze di Java già abbastanza solide alla ricerca di un modo divertente e coinvolgente per svilupparle ulteriormente oppure di neofiti supervisionati da un insegnante.

La divisione del gameplay in due fasi di gioco, in cui la prima, quella relativa alla programmazione, influisce profondamente sulla seconda, quella relativa allo

scontro, è risultata una carta vincente di questo titolo. Nel videogioco soggetto di questa tesi, questo concetto verrà ripreso e migliorato in modo da spronare il giocatore a dare il massimo in entrambe le fasi, che in questo caso si condizionano a vicenda, per ottenere vantaggi nell'altra.

2.2.2 Rabbids: Coding!

Rabbids: Coding! è un videogioco gratuito sviluppato da Ubisoft, una delle aziende più importanti del settore e proprietaria di brand videoludici molto famosi come Assassins' Creed e Far Cry.

Il titolo, rilasciato il 30 settembre 2020, è un classico puzzle game a livelli che utilizza una delle mascotte della software house francese, i Rabbids, per introdurre i principi della programmazione a un pubblico giovane.

Rabbids: Coding! è disponibile gratuitamente sia su PC che su dispositivi mobile; Tuttavia, la versione per IOS e Android è stata modificata per adattarsi ad uno schermo più piccolo e ai comandi touch ma ciò ha portato più problemi che benefici. La metafora alla base del gioco, ovvero quella di rappresentare blocchi di codice come pezzi di un puzzle, è stata sostituita con degli strani e controintuitivi quadrati che fin troppo spesso rischiano di essere toccati accidentalmente cambiando gli ordini dei comandi in maniera poco chiara senza la possibilità di effettuare annullare automaticamente l'operazione e frustrando il giocatore che può vedere minuti di ragionamenti e tentativi resi vani da un movimento maldestro del proprio dito. Per questo motivo, si è presa in considerazione solamente la versione del gioco sviluppata per computer.

Rabbids: Coding! è un puzzle game vecchio stile, ovvero un titolo il cui contenuto è contraddistinto dalla presenza di numerosi enigmi che il giocatore deve risolvere. In questo caso, il videogame è strutturato in quaranta livelli dalla difficoltà crescente; Una volta terminato il primo si ha accesso al secondo, una volta terminato il secondo si ha accesso al terzo e così via.

L'input del gioco è sicuramente gradevole per i più giovani: una navicella spaziale è stata presa d'assalto dai Rabbids, delle creature dal comportamento stravagante che hanno messo a soqquadro ogni stanza. Il compito del giocatore è quello di sbarazzarsi delle piccole creature bianche riponendole in lavatrici dagli effetti simili a quelli dei buchi neri e ripulire le varie stanze raccogliendo degli oggetti fuori posto. Per fare questo, si prende il controllo sia di robot volanti che degli stessi Rabbids sottoposti al lavaggio del cervello per sottostare agli ordini del giocatore.

La parte educativa sta proprio nel come vengono impartiti questi ordini, ovvero tramite la programmazione. Infatti, in una struttura che ricorda molto il software di programmazione Scratch, il giocatore deve mettere in ordine o concatenare tra di loro dei blocchi di codice, rappresentati tramite pezzi di un puzzle, in modo da definire il comportamento del personaggio protagonista del livello (Figura 2.2).



Figura 2.2: Primo livello di *Rabbids: Coding!*

Una volta terminata la fase di programmazione si può avviare il codice e, osservando i movimenti del coniglio o del robot, capire se il codice rispecchia le proprie intenzioni o se siano stati commessi degli errori. Dato che ad ogni movimento del protagonista del livello viene indicato al giocatore quale pezzo di codice ha causato tale azione, diventa estremamente semplice individuare la posizione di eventuali errori e altrettanto agevole risulta correggerli.

Inizialmente i comandi messi a disposizione del giocatore sono molto basilari e prevedono solo l'esecuzione di movimenti semplici del protagonista come avanzare di una casella o girarsi di novanta gradi. Tuttavia, con l'avanzare dei livelli, la tipologia di pezzi di puzzle aumenta, introducendo anche strutture più complesse come l'if o i cicli. Il titolo però non affronta argomenti avanzati della programmazione in quanto è pensato come possibile primo approccio al mondo dell'informatica per i più giovani.

Il gioco, inoltre, prova a premiare in particolar modo i giocatori che pongono attenzione all'ottimizzazione del proprio “codice.” Per ogni livello, infatti, è indicato un numero di mosse “ottimali” da utilizzare per risolvere l'enigma. Tale numero, però, non è da considerarsi come un limite; il giocatore può usare il numero di mosse che ritiene necessario e il livello risulterà ugualmente completato. Tuttavia, alla fine di ogni stage, al lavoro svolto dagli aspiranti programmatore viene dato un voto in stelline, dove una stella indica un numero molto alto di mosse non necessarie utilizzate, mentre tre stelle è indice di un ottimo lavoro di ottimizzazione.

Analizzando i pregi di “Rabbids: Coding!”, la metafora programmazione-puzzle è sicuramente una di quelli su cui porre maggiore enfasi. Cercare di insegnare nuovi concetti ad un giocatore utilizzandone di già noti è sicuramente un’idea vincente che, soprattutto nell’ambito educativo, può fare la differenza nella comprensione di argomenti altrimenti troppo complessi.

Un plauso al titolo è d’obbligo anche per l’accompagnamento del giocatore/studente nel percorso di apprendimento: ogni concetto che viene presentato viene messo al centro di numerosi livelli a difficoltà crescente, in modo che il giocatore, sempre aiutato da scritte introduttive ai vari stage ed incoraggiato da frasi motivazionali nei casi di fallimento, possa assimilarlo e comprenderlo al meglio prima di passare al successivo.

Tuttavia, il titolo, anche nella versione PC, presenta più di un difetto che è necessario evidenziare ed analizzare.

Un discorso piuttosto particolare è quello relativo proprio al genere di cui fa parte “Rabbids: Coding！”, quello dei puzzle game. In generale, trasformare un compito di una materia scolastica in un enigma sembra il modo più efficace per rendere lo studio un’esperienza ludica. Tuttavia, bisogna considerare che i rompicapi puri, soprattutto tra i più giovani, non sono molto popolari. Non a caso, per anni si è addirittura discusso se i puzzle game fossero considerabili videogiochi, arrivando solo dopo numerose discussioni ad una risposta affermativa [8].

I puzzle game, se non per alcune persone, tendono a non avere un grosso potere motivazionale di per sé e possono stancare dopo poco. Inoltre, un videogioco basato solo ed unicamente sulla risoluzione di enigmi presenta una scarsissima rigiocabilità dato che, una volta trovata la soluzione, il livello di sfida di una seconda partita diventa pari a zero. Per questi motivi, di solito, i giochi di questo genere tentano di aggiungere dei motivatori esterni; ad esempio Portal, il capolavoro targato Valve, è un videogioco in cui si susseguono rompicapi già di loro più stimolanti grazie al continuo rinnovamento delle meccaniche di gioco, ma, soprattutto, aggiunge una componente narrativa estremamente coinvolgente che spinge il giocatore a voler continuare la sua avventura per scoprire qualcosa in più sul mondo di gioco.

Uno dei metodi per rendere gli enigmi più interessanti ed eleganti è quello di incorporarli in un gameplay più ampio, che non si limiti a bloccare il giocatore dinanzi ad un palese rompicapo, bensì che lo integri in un ambiente di gioco più coinvolgente ed affascinante, strategia che possiamo ritrovare, ad esempio, nei capolavori della saga The Legend of Zelda.

Tornando a “Rabbids: Coding！”, purtroppo, il titolo manca di veri motivatori oltre la mera soddisfazione di risolvere l’enigma e sbloccare il successivo. Le stelle possono convincere il giocatore ad impegnarsi di più ma, in fin dei conti, anche questa risulta una meccanica fine a sé stessa dato che, oltre ad incrementare un valore numerico nel menù di gioco, non aggiungono molto altro all’esperienza. Il rischio,

quindi, è quello che il giocatore, dopo aver superato qualche livello, in mancanza di una storia, di personalizzazione dei personaggi, di vere ricompense che diano valore ai suoi sforzi, abbandoni il gioco per mancanza di interesse.

2.2.3 Outcore: Desktop Adventure

Outcore: Desktop Adventure è un videogioco “indie”, termine che viene utilizzato per indicare titoli pubblicati da team di sviluppo indipendenti composti da poche o, addirittura, singole persone, come nel caso preso in esame. Pubblicato il 26 Settembre 2022, il titolo è scaricabile in modo completamente gratuito dalla famosa piattaforma per videogiochi Steam.

Descrivere Outcore risulta davvero complesso perché, come spesso accade nel panorama indie, questo progetto presenta caratteristiche uniche e sopra le righe che donano all’opera una forte personalità capace di far vivere ai giocatori un’esperienza memorabile.

La maggior parte dell’esperienza del titolo prende forma sul desktop del computer del giocatore dove appare una piccola ragazza misteriosa, di nome Lumi, che ha perso la memoria. Starà al giocatore accompagnare quella che diventerà un’ospite fissa del suo PC in un’avventura alla ricerca dei suoi ricordi. Per riuscire nell’impresa sarà necessario svolgere le attività più disparate: cercare file all’interno delle cartelle di gioco, creare stravaganti forme con Microsoft Paint, affrontare fasi di platforming in ambienti che più ricordano i videogiochi tradizionali e combattere contro nemici sempre più forti, anche con il proprio desktop a fare da arena.

Insomma, Outcore: Desktop Adventure è sicuramente un titolo variegato e originale, pronto a sorprendere continuamente il giocatore con idee innovative, una storia intrigante ed una comicità che sa come far sorridere il giocatore, ma se il gioco si fermasse a questo sembrerebbe fuori luogo parlare di un titolo del genere in una tesi riguardante lo sviluppo di un applied game.

In realtà, il titolo sviluppato da Doctor Shinobi, tra le varie fasi di gioco che caratterizzano l’avventura, ne presenta una che dona ad Outcore anche una componente da gioco educativo.

A poco più di quarantina minuti dall’inizio dell’avventura, infatti, una serie di azioni del giocatore portano all’avvio di un software chiamato “Idle Game”, un gioco all’interno del gioco.

Idle Game ricorda per molti versi “Rabbids: Coding!”; è un gioco strutturato a livelli in cui bisogna muovere un piccolo personaggio (che NON è la sopraccitata Lumi) su una griglia, programmando in anticipo i suoi movimenti con lo scopo di raggiungere un determinato punto dopo aver raccolto delle monete sparse per la mappa. A differenza del titolo Ubisoft però, qui non vi saranno pezzi di puzzle per permettere al giocatore di programmare i movimenti dell’omino, bensì un semplicissimo file di testo. Infatti, il giocatore dovrà scrivere manualmente o, sem-



Figura 2.3: Minigioco sulla programmazione di *Outcore: Desktop Adventure*.

plicemente, sfruttando la funzione “copia e incolla”, i comandi nel file “Worker1.txt” situato in una delle directory di gioco per controllare i movimenti del personaggio secondo le proprie preferenze (Figura 2.3).

Dalla descrizione fornita, il compito sembrerebbe piuttosto arduo e macchinoso, soprattutto per persone che non hanno esperienza nel campo della programmazione, ma tutto cambia con la guida di Lumi. Il personaggio, infatti, completamente esterno all’Idle Game e con cui il giocatore avrà già avuto modo di creare un minimo di legame affettivo, guida lo stesso quel tanto che basta per fargli comprendere ciò che deve fare senza fornire la soluzione.

Progressivamente, il giocatore accumula monete spendibili in un negozio virtuale. Con queste monete, può procedere all’acquisto di ulteriori livelli, progredendo nell’avventura, oppure acquisire nuovi comandi, come ad esempio il ciclo “while”. Questi comandi possono risultare cruciali per risolvere determinati enigmi o per ottimizzare il codice, semplificandone la struttura.

Anche in momenti più noiosi, come l’esecuzione di un ciclo while che di fatto automatizza la raccolta infinita di monete in un livello del mini-gioco, il silenzio viene presto spezzato dalla solare personalità di Lumi che fornisce al giocatore informazioni utili per comprendere la trama generale del gioco.

Al completamento del quarto livello, il giocatore ha la possibilità di chiudere “Idle Game” e procedere con il gioco principale ma, nel caso lo desiderasse, potrebbe anche continuare a risolvere enigmi relativi alla programmazione con l’obiettivo di accumulare una grande quantità di monete e acquisire nuovi potenziamenti che influenzino non solo il mini-gioco, ma anche l’intera avventura proposta da Outcore.

Per quanto la struttura del mini-gioco educativo inserito in Outcore: Desktop Adventure risulti simile a “Rabbids: Coding!”, ha delle funzionalità che elevano l’esperienza di gioco ad un livello superiore. La raccolta delle monete, ad esempio, non è fine a sé stessa, come nel caso delle stelle dell’avventura targata Ubisoft; raccogliere tanti soldi, anche oltre il necessario, permette al giocatore di sbloccare nuovi livelli, nuove funzioni per semplificarsi il lavoro o uno strano premio contraddistinto da tre punti interrogativi che sicuramente riesce ad alzare il grado di curiosità del giocatore. Sforzarsi, in Idle Game, ripaga il giocatore con delle ricompense vere, che possano soddisfarlo e per cui vale la pena impegnarsi.

Inoltre, un forte motivatore che spinge il giocatore ad andare avanti è sicuramente la trama principale dell’intero gioco. Non si vuole sbloccare il prossimo livello per il semplice gusto di farlo, bensì per avvicinarsi un passo in più alla verità nascosta del titolo, per conoscere meglio quella strana entità che all’improvviso è comparsa sul proprio desktop o semplicemente per vedere come reagisce Lumi alla risoluzione di un enigma che, a prima vista, può sembrare estremamente complesso.

Tuttavia questo motivatore può essere un’arma a doppio taglio: il fatto che il giocatore possa abbandonare la fase di programmazione dopo solo il quarto livello per proseguire con la trama porta la maggior parte delle persone a chiudere Idle Game per continuare a seguire la verità sul passato di Lumi e sono pochi coloro che continuano a programmare i movimenti del piccolo personaggio per raccogliere monetine, nonostante i papabili premi che potrebbero tornare utili anche più in avanti nel gioco principale.

Probabilmente questo non costituisce un problema rilevante in Outcore, poiché il suo obiettivo principale non è insegnare i fondamenti della programmazione. Tuttavia, dato che il tema trattato in questa tesi è lo sviluppo di un applied game, è comunque importante considerare questa informazione.

Outcore: Desktop Adventure ha ricevuto su Steam un giudizio del pubblico “estremamente positivo”, con ben il 97% di voti positivi. Tuttavia, ciò che colpisce è l’abbondanza di citazioni legate alla fase di programmazione nelle recensioni presenti sulla pagina Steam del gioco. Scorrere i vari commenti spesso porta a incontrare frammenti di codice tratti dal documento "Worker1.txt" o interpretazioni ironiche dei comandi di gioco, un segno che, nonostante "Idle Game" sia solo una breve parte di un’esperienza più ampia, è riuscita a far breccia nelle menti dei giocatori e a diventare iconica.

Se invece si controllano le recensioni di “Rabbids: Coding!” pubblicate sulla piattaforma Google Play Store, si nota un certo malcontento generale per un titolo che, invece, non sembra essere riuscito a lasciare il segno. Il paragone tra la ricezione del pubblico dei due titoli fa riflettere sul fatto che, in un gioco, anche partendo da una struttura di gameplay simile, aggiungere una buona trama, dei motivatori validi e dei personaggi iconici può cambiare completamente l’esperienza di gioco.

Related Works	Gameplay	Pro	Contro
RoboCode	Gioco in cui bisogna programmare il comportamento del proprio carrarmato per farlo scontrare con quello di altri giocatori.	<ul style="list-style-type: none"> + Il pensiero di dover competere contro altri giocatori è un forte motivatore che spinge il giocatore a voler programmare il proprio carrarmato al meglio; + Presenta una community attiva che dona al giocatore che ne fa parte un senso di appartenenza. 	<ul style="list-style-type: none"> - Il comparto estetico è spoglio, vetusto e poco intuitivo; - Totale assenza di tutorial o di introduzione alla programmazione per neofiti.
Rabbids: Coding!	Puzzle game in cui si impariscono ordini al personaggio tramite la combinazione di blocchi di codice rappresentanti i principali costrutti di controllo del flusso come if, for, ecc.	<ul style="list-style-type: none"> + Intuitivo e facile da comprendere anche per chi non ha mai programmato grazie alla metafora programmazione – puzzle; + Componente grafica accattivante, soprattutto per i più giovani; + Ricompensa i giocatori che riescono a migliorare il proprio “codice”. 	<ul style="list-style-type: none"> - Alla lunga può risultare ripetitivo a causa della mancanza di motivatori oltre la mera soddisfazione di completare un livello; - Presenta una scarsa rigiocabilità.
Outcore: Desktop Adventure (Idle Game)	Sezione puzzle del gioco in cui bisogna programmare i movimenti di un personaggio tramite un documento di testo con l'obiettivo di risolvere enigmi.	<ul style="list-style-type: none"> + Il gioco ricompensa il giocatore con premi di valore quali nuove meccaniche, livelli bonus o sorprese inaspettate; + Presenza di una trama che porta il giocatore ad impegnarsi per scoprire di più sul mondo di gioco; + Presenza di una protagonista carismatica a cui è facile affezionarsi e che interagisce direttamente con il giocatore, spronandolo a completare i livelli. 	<ul style="list-style-type: none"> - Esperienza breve che fa parte di un gioco ben più ampio che non ha, in generale, fini educativi.

Tabella 1: Riepilogo delle analisi sui Related Work presentati in questo capitolo.

Capitolo 3 - Metodologia

Questo capitolo ha l'obiettivo di spiegare ed analizzare nel dettaglio i più importanti aspetti del progetto. Più precisamente, illustra quali sono gli obiettivi che hanno ispirato la creazione dell'applied game (il cui codice è visionabile nell'apposita repository di GitHub [5]), quali sono le sue principali funzioni, quali sono stati gli strumenti che hanno permesso il suo sviluppo e, infine, analizza come sono state affrontate alcune sfide relative alla sua progettazione e programmazione.

3.1 Obiettivi

Questa tesi mira a creare un prototipo di videogioco educativo per insegnare i principi fondamentali del linguaggio Java. L'obiettivo è quello di individuare un approccio allo sviluppo di un applied games che risulti più efficace rispetto a quelli già utilizzati dai prodotti che attualmente popolano il mercato. Per raggiungere tale risultato, è necessario porre maggiore enfasi sullo studio e l'analisi dei principi e delle regole del game design, in modo da riuscire successivamente ad integrare i concetti del linguaggio orientato agli oggetti direttamente nel gameplay dell'applied game.

In particolare, questo progetto mira a coinvolgere non solo coloro che sono già interessati all'argomento e sono alla ricerca di un primo approccio ad esso leggero e divertente, ma anche coloro che considerano l'informatica un campo estremamente complesso e distante. Si vuole dimostrare che, con l'approccio giusto, tutti possono avvicinarsi a questo mondo ed iniziare ad apprezzarlo con il minimo sforzo.

Per raggiungere tale obiettivo risulta fondamentale riuscire a creare un videogioco che non solo sia fruibile da chiunque ma che riesca ad intrattenere anche coloro che non hanno mai avuto interesse nell'argomento; in altre parole, l'applied game deve essere percepito come un normalissimo "videogioco". Al fine di raggiungere questo obiettivo, è necessario riuscire a trasmettere nuove conoscenze attraverso le meccaniche di gioco, senza restituire mai al giocatore la sensazione di star partecipando ad una peculiare "lezione di informatica".

3.2 La Struttura del Gioco

Prima di iniziare lo sviluppo del progetto, che è stato momentaneamente denominato "*Project Coding*", si è reso necessario effettuare alcune considerazioni. Prima di tutto, risulta fondamentale la scelta della piattaforma per cui sviluppare l'applied game.

3.2.1 La Scelta della Piattaforma

Per quanto sul mercato esistano svariati dispositivi dedicati principalmente alla fruizione di videogiochi come i più recenti PlayStation 5, Xbox Series X o Nintendo Switch, ignorare il potenziale del “*mobile gaming*”, ovvero dei videogiochi per smartphone, potrebbe essere un grave errore, principalmente per due ragioni:

- Rispetto alle console per videogiochi, che vengono acquistate maggiormente da appassionati del mondo videoludico, gli smartphone sono largamente più diffusi; ciò significa che pubblicare un gioco su piattaforme quali iOS e Android permette di estendere ampiamente il bacino di utenza potenziale;
- Il mercato dei giochi per dispositivi mobili sta vivendo un periodo di crescita straordinaria negli ultimi anni. Se prima solo una minoranza di videogiocatori utilizzava il proprio smartphone per regalarsi qualche minuto di svago, ad oggi i dati confermano che la situazione si è completamente ribaltata ed il mercato mobile è diventato il settore più importante dell’intera industria dei videogiochi.

Considerando quanto detto, si è optato per lo sviluppo dell’applied game su piattaforma mobile.

3.2.2 La Scelta del Genere di Videogioco

Per delineare la struttura principale del videogioco, è stato essenziale condurre un’analisi dell’architettura dei videogiochi preesistenti.

Come già precedentemente espresso, il termine “videogioco” include una vasta sfera di opere interattive che possono presentare caratteristiche strutturali completamente diverse tra loro. Per questo motivo, nel mondo videoludico si tende a dividere i videogiochi per categoria a seconda delle caratteristiche che li contraddistinguono. Dopo un’accurata ricerca si è deciso di optare per la creazione di un videogioco che possedesse le peculiarità del genere “roguelite”.

Quando si parla di "roguelite", si fa riferimento a un gioco che presenta, solitamente, la seguente struttura:

- Il giocatore inizia una nuova partita;
- Durante la partita, il giocatore progredisce in una serie di livelli generati proceduralmente sconfiggendo nemici, acquisendo nuove abilità e altri oggetti di gioco;
- Alla fine della partita, che sia terminata con una vittoria o una sconfitta, il giocatore guadagna esperienza, monete o altre ricompense permanenti che

gli permettono di iniziare la prossima partita con un personaggio più forte o di sbloccare nuovi livelli;

- Le abilità e gli oggetti ottenuti durante una partita sono utilizzabili solo in quella specifica partita e non vengono conservati per la successiva;
- Le partite sono relativamente brevi e solitamente durano al massimo un'ora di gioco.

Ovviamente ogni gioco può adottare varianti delle caratteristiche sopracitate o aggiungerne delle proprie.

La struttura roguelite deriva da un genere ben più famoso che si è affermato negli ultimi anni, ovvero il “roguelike”. Questo genere però è decisamente più punitivo e frustrante: in generale, le ricompense per la vittoria di una partita sono meno importanti di quelli di un videogioco roguelite ma soprattutto, in caso di sconfitta, il giocatore non ottiene altro se non una schermata di “Game Over”, anche dopo partite durate ben più di un'ora, il che può provocare una forte frustrazione, emozione che in un progetto educativo come Project Coding andrebbe assolutamente evitata.

La struttura roguelite quindi permette al videogioco di raggiungere un obiettivo importante per un applied game, ovvero l'implementazione del concetto di Graceful Failure: un giocatore che viene sconfitto ovviamente dovrà ricominciare da capo ma non vedrà i suoi sforzi come vani perché il gioco lo ricompenserà con monete, punti esperienza ed altri premi che gli permetteranno di potenziare il proprio personaggio permanentemente e riprovare, magari utilizzando un nuovo approccio, consapevole delle sfide che lo attendono e che in precedenza lo hanno condotto alla sconfitta. Il fallimento, quindi, non è più un evento negativo, ma un passo in avanti per arrivare alla vittoria più facilmente.

Inoltre, questa tipologia di gioco risulta molto efficace per un videogioco per smartphone indirizzato ad un pubblico giovane: il giocatore può portare a termine una partita tramite brevi sessioni di gioco in cui, ad esempio, può sconfiggere un singolo nemico e poi fermarsi, per poi riprendere la partita da dove l'aveva lasciata; in questo modo il gioco riesce a adattarsi alla veloce e ritmata vita degli adolescenti.

3.2.3 Il Gameplay Loop

Per comprendere appieno la struttura di base di Project Coding, è essenziale iniziare spiegando il concetto di "gameplay loop". Il termine "gameplay loop" è ampiamente utilizzato nel campo del Game Design per descrivere il ciclo di azioni compiute dal giocatore durante una partita. In sintesi, definisce le attività che il giocatore svolge durante il gioco.

Introduciamo quindi il gameplay loop base di Project Coding:

- Il giocatore inizia una nuova partita;
- Viene messo di fronte a varie sfide e varie scelte;
- Vince/perde ottenendo delle ricompense adeguate ai risultati raggiunti;
- Usa le risorse ottenute per potenziare il personaggio;
- Inizia una nuova partita con evidenti vantaggi rispetto alla precedente, dati sia dalle statistiche migliorate del suo personaggio, sia dall'abilità del giocatore stesso che ha acquisito più esperienza.

Di seguito, verranno analizzate le varie fasi di gioco che compongono una singola partita di Project Coding.

3.2.4 La Selezione del Livello

All'inizio di una nuova partita, il giocatore si trova di fronte alla decisione cruciale di selezionare un livello. Inizialmente, sarà possibile accedere solo a un livello, mentre gli altri saranno sbloccati gradualmente man mano che il giocatore avanza nel gioco.

Nonostante la struttura base dell'applied game non muti di partita in partita, la scelta del livello influisce significativamente su molti aspetti: se si considera solo il lato estetico, ogni livello è caratterizzato da location uniche nella maggior parte delle fasi di gioco, in modo che il giocatore percepisca anche un senso di progressione e di novità man mano che sblocca ed affronta nuovi stage; dal lato del gameplay invece, molte proprietà mutano di livello in livello, come, ad esempio, la potenza dei nemici, la tipologia di oggetti che si possono ottenere per potenziarsi, la quantità e la qualità delle ricompense finali e la categoria di imprevisti che il giocatore potrebbe trovarsi ad affrontare lungo il suo cammino.

Ma anche se il giocatore dovesse o volesse ripetere lo stesso livello più di una volta non si ritroverebbe comunque davanti alla stessa identica esperienza precedentemente vissuta, grazie ad una caratteristica fondamentale della sezione subito successiva alla selezione del livello che verrà approfondita in seguito.

3.2.5 La Mappa

Una volta scelto il livello desiderato, il software presenterà al giocatore una mappa costituita da vari nodi, ciascuno contrassegnato da un determinato simbolo. Questa mappa rappresenta il percorso che il giocatore dovrà seguire per completare il livello ed è illustrata nella Figura 3.1.

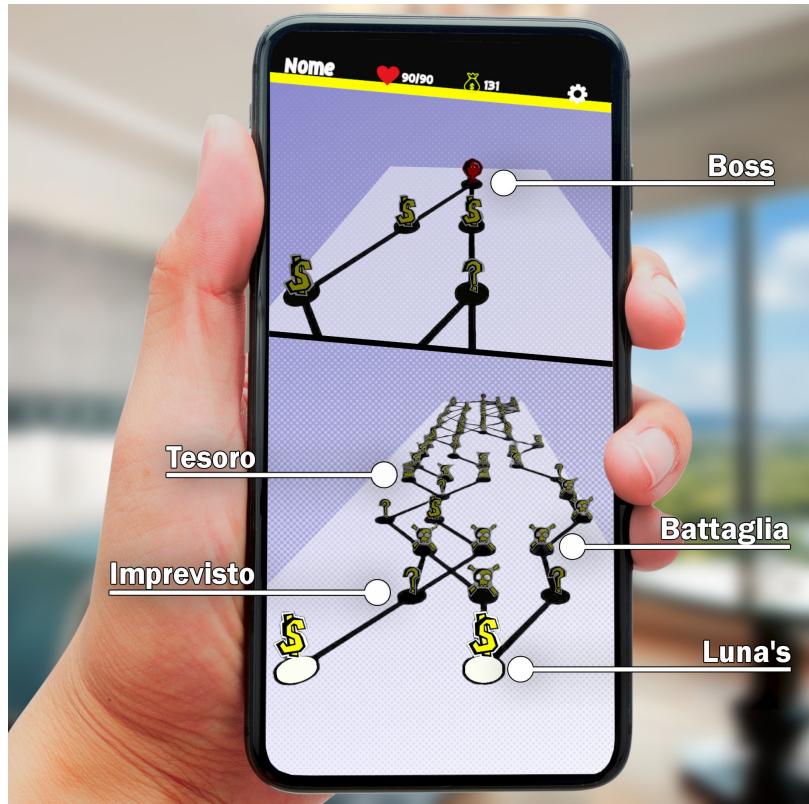


Figura 3.1: Implementazione della mappa di una partita di Project Coding.

Ogni nodo rappresenta un particolare evento che il giocatore dovrà affrontare per procedere e passare al nodo successivo. Ci sono in totale 5 tipologie di eventi che verranno approfonditi in seguito: Il Luna's, la Battaglia, l'Imprevisto, il Tesoro e, infine, presente sempre e solo all'ultimo nodo, la Battaglia Contro il Boss.

La mappa offre al giocatore una serie di percorsi, talvolta intrecciati tra loro, offrendogli completa libertà di scelta riguardo al percorso da seguire. L'unica restrizione è che può muoversi solo in avanti, spostandosi su un nodo direttamente connesso a quello attuale.

La caratteristica di questa fase che rende unica ogni partita, precedentemente citata durante la presentazione della selezione del livello, riguarda la generazione della mappa. Infatti, all'inizio di ogni partita, il software genera la mappa proceduralmente, scegliendo automaticamente la disposizione, la tipologia e perfino le caratteristiche degli eventi dei nodi. Le uniche limitazioni imposte al generatore di mappe sono:

- i nodi della prima riga devono sempre portare all'evento “Luna's” in quanto ciò permette al giocatore di prepararsi alla partita (approfondiremo questo aspetto più avanti);

- l'ultima riga deve essere composta da un solo nodo rappresentante l'evento della Battaglia Contro il Boss;
- la dimensione della mappa è fissa, basata su una griglia di 20 righe per 5 colonne.

Dopo aver selezionato un nodo raggiungibile, contraddistinto da un brillante colore giallo che lampeggia di bianco, il giocatore può accedere a un nuovo evento. Questo evento varia in base al simbolo posizionato sopra la piattaforma circolare che compone il nodo. Come precedentemente affermato, i primi nodi accessibili all'inizio di una partita sono di tipo "Luna's" ma, per facilitare la comprensione di ogni sezione, verrà illustrato prima un'altra tipologia di evento: la Battaglia.

3.2.6 La Battaglia

Le battaglie consistono in scontri in modalità "uno contro uno" dove l'avversario è controllato dal gioco stesso. Una volta avviata, il giocatore si ritrova davanti alla schermata mostrata nella Figura 3.2. In questa fase, l'avversario attacca il giocatore tramite tecniche di vario tipo: dal lancio di oggetti che feriscono il personaggio al contatto fino ad arrivare a potenti colonne di fuoco provenienti dai lati dello schermo.

Il giocatore è quindi chiamato a studiare e comprendere tali attacchi per impartire i comandi corretti al proprio personaggio al fine di evitare le minacce e subire meno danni possibili. Il personaggio può essere controllato dal giocatore attraverso il semplice movimento del dito sullo schermo, comunemente noto come "*swipe*". Ad esempio, uno swipe verso destra farà muovere il personaggio verso destra, uno swipe verso l'alto lo farà saltare, mentre uno swipe verso il basso lo farà rapidamente ritornare al suolo.

Se, sfortunatamente, la barra della vita del giocatore, situata nella parte inferiore dello schermo, si esaurisse e raggiungesse lo 0, la partita verrebbe conclusa. Il giocatore sarebbe quindi riportato al menu iniziale, ma prima di ciò, il gioco fornirebbe una valutazione delle sue prestazioni, indicando il numero di monete guadagnate durante la partita. Queste monete possono successivamente essere impiegate per migliorare le statistiche del giocatore e conferirgli un vantaggio nella partite future.

Ora che è chiaro come il giocatore può difendersi dagli attacchi nemici, è possibile procedere ad esaminare le modalità con cui può contrattaccare.



Figura 3.2: Implementazione della battaglia in Project Coding.

Nella parte inferiore dello schermo, ai lati della barra della vita, si trovano due pulsanti denominati "Mossa 1" e "Mossa 2". Questi pulsanti sono circondati da una barra di caricamento chiamata "Reload Bar"; ciascun pulsante è cliccabile solo quando la relativa Reload Bar è completamente carica. Quando uno di questi pulsanti virtuali viene premuto, il personaggio del giocatore compie un'azione che può consistere in un attacco contro il nemico, un tentativo di potenziare o indebolire l'avversario, oppure nel recupero dei punti vita (anche detti "HP", sigla di "Health Points").

Il comportamento legato ai due pulsanti cambia di partita in partita, di giocatore in giocatore, e questo perché tali "mosse" devono essere programmate in una fase che precede la battaglia; durante l'evento del "Luna's."

Il gioco presenta, inoltre, un sistema di debolezze e resistenze in base all'elemento dell'attacco e dell'avversario in stile morra cinese. Più precisamente esistono tre tipi di elementi associabili ai nemici, ai personaggi e alle mosse: fuoco, acqua ed erba; il fuoco è resistente all'erba ma è debole all'acqua, l'acqua è resistente al fuoco ma debole all'erba, l'erba è resistente all'acqua e debole al fuoco. Un attacco di fuoco causerà quindi danni maggiori su un avversario di tipo erba mentre risulterà

meno efficace su un avversario di tipo acqua. Esistono anche attacchi e nemici che presentano l'elemento “normale” e che quindi subiscono/infliggono sempre lo stesso numero di danni da/verso qualunque elemento.

Una volta sconfitto il nemico facendo calare la sua barra della vita a 0, il giocatore viene premiato con una piccola ricompensa per poi essere riportato alla mappa dove avrà la possibilità di procedere lungo il suo percorso.

3.2.7 Il Luna's

Dopo aver compreso il funzionamento delle battaglie, si può procedere con la spiegazione del primo evento al quale il giocatore viene esposto all'inizio di una nuova partita, ma che può ripresentarsi anche nelle fasi successive del gioco: il "Luna's".



Figura 3.3: Implementazione di tre schermate del Luna's. Da sinistra verso destra: Menu Principale, Distributore Automatico e Panchina.

Dopo aver selezionato un nodo con il simbolo del dollaro sulla mappa, il giocatore viene virtualmente trasportato in un piccolo locale ricco di distributori automatici. In questo luogo, ha accesso a tre sezioni differenti:

- Distributore Automatico: qui il giocatore può acquistare nuovi oggetti che servono per comporre una mossa. Più precisamente può acquistare oggetti chiamati “pezzi di codice” e “pacchetti”. Il loro utilizzo verrà approfondito in

un secondo momento. Questa schermata è illustrata nell'immagine centrale della Figura 3.3;

- Panchina: qui il giocatore può spendere una somma di denaro per riposare sulla panchina e recuperare punti vita (immagine destra della Figura 3.3);
- PC: questa è la fase in cui il giocatore può “programmare” i propri attacchi utilizzando i pacchetti e pezzi di codice di cui è in possesso.

Mentre le sezioni del Distributore Automatico e della Panchina vertono su concetti piuttosto semplici, il PC risulta leggermente più complesso ma rappresenta il cuore della componente educativa di Project Coding.

Una volta che si accede alla sezione PC tramite i pulsanti appositi presenti nel menu principale del Luna's o nelle altre sezioni, il giocatore viene messo di fronte ad una schermata (Figura 3.4) occupata per la maggior parte dalla definizione di una classe Java. Tale classe non è altro che la rappresentazione del personaggio del giocatore:

- In alto è possibile notare la scritta import affiancata da tre spazi vuoti, riempibili dai pacchetti di cui dispone il giocatore.
- Nella parte sottostante vi è la dichiarazione della classe del giocatore, con la definizione delle variabili che indicano le statistiche del giocatore e, successivamente, tre metodi posti ognuno su un pulsante diverso: uno è il costruttore, uno è il metodo collegato al pulsante “Move1” e, come intuibile, l’ultimo è il metodo collegato al pulsante “Move2”. Cliccando su uno di questi viene mostrata una seconda schermata per la completa personalizzazione del metodo.
- Infine, nella parte bassa dello schermo, è possibile visualizzare i pacchetti del giocatore. Questi sono trascinabili all’interno degli slot dell’import che si trova nella parte alta dello schermo. Così facendo, il giocatore può importare i pacchetti e fare uso delle classi al loro interno nella sezione dedicata alla definizione dei metodi.

Per una migliore comprensione delle dinamiche di gioco, prima di passare alla sezione successiva, è essenziale comprendere il ruolo di un "pacchetto" all'interno dell'economia del gioco.

I *pacchetti* sono degli oggetti di gioco che possono essere acquistati nel negozio o trovati come ricompensa nei tesori. Questi pacchetti contengono delle “*classi*”, i cui metodi possono essere utilizzati come parte del codice per definire le finalità delle mosse. Per fare un esempio, il pacchetto "fire" può contenere la classe Flame il cui metodo attack() può essere utilizzato all'interno di un attacco per scagliare

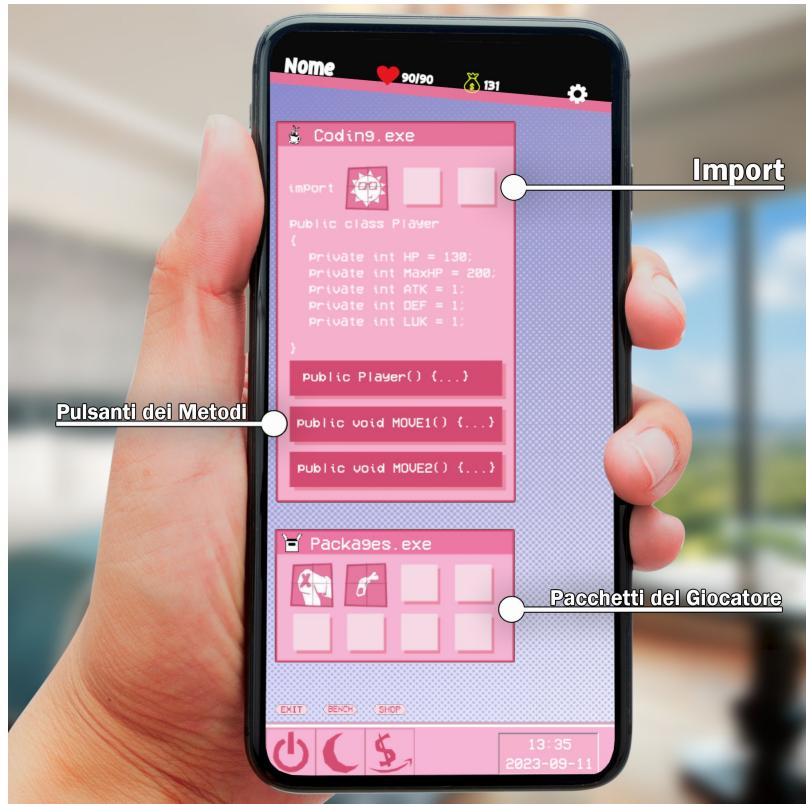


Figura 3.4: Implementazione della schermata PC di Project Coding, dedicata alla funzione di "import" dei pacchetti.

un potente colpo di fuoco contro il nemico. Esistono varie tipologie di pacchetti che contengono classi diverse i cui effetti spaziano dal lancio di un attacco contro il nemico al recupero dei punti vita del proprio personaggio.

Quando un pacchetto viene “importato”, trascinandolo dalla sezione dei pacchetti del giocatore alla sezione dell’import, le classi di tale pacchetto diventano disponibili nella schermata di definizione del metodo. Per capire qual è il ruolo di una classe, risulta necessario analizzare la schermata di definizione del metodo.

Per modificare l’effetto di una mossa è necessario cliccare sul pulsante del metodo corrispondente. Così facendo, si apre la schermata di definizione del metodo (Figura 3.5), i cui elementi, in ordine dal più in alto al più in basso, sono:

- *Reload Time:* In questa sezione, contrassegnata da un orologio, viene indicato il tempo di ricarica dell’attacco, ovvero quanto tempo deve passare durante una battaglia tra un’esecuzione dell’attacco ed il successivo. Questo tempo aumenta o diminuisce a seconda della quantità, della qualità e della tipologia delle classi utilizzate all’interno del metodo. Il tempo di caricamen-

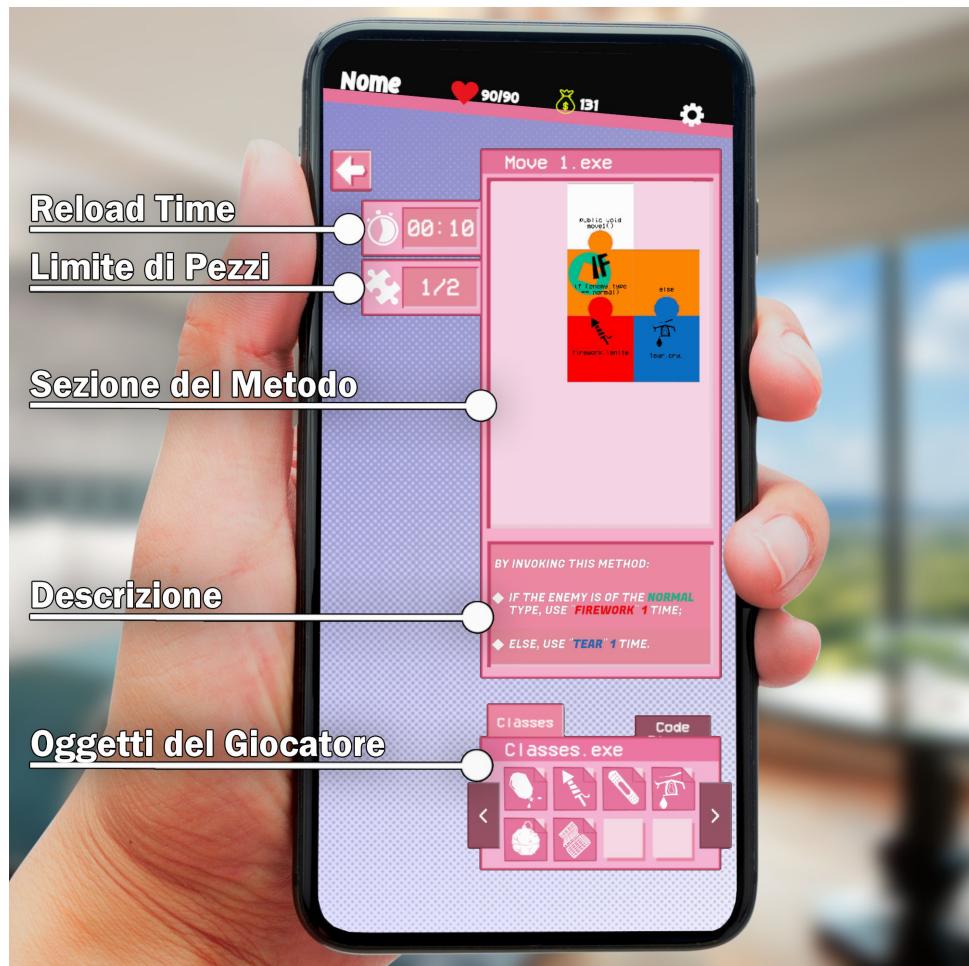


Figura 3.5: Implementazione della sezione di Project Coding dedicata alla personalizzazione dei metodi.

to della Reload Bar del pulsante in battaglia equivale al valore del Reload Time;

- Limite di pezzi: In questa sezione, contrassegnata dal simbolo di un pezzo di puzzle, il giocatore può tenere sotto controllo il numero di pezzi inseriti all'interno del codice e quanti ancora ne può inserire prima di raggiungere il limite;
- Sezione del metodo: Qui è possibile comporre il proprio attacco con sezioni di codice, come fossero tasselli di un puzzle. Per fare ciò, basta incastrare le classi e i pezzi di codice nel puzzle trascinandoli nei posti opportuni. Ad esempio: trascinare un “for 2” nel primo slot e poi collegare a questo la

classe “cerotto”, permetterà al giocatore, alla pressione del tasto durante la battaglia, di curarsi per ben due volte consecutive. La sezione bianca è fissa e inamovibile mentre gli altri pezzi possono essere inseriti e rimossi a piacimento;

- Descrizione: Dato che il giocatore potrebbe non avere familiarità con il linguaggio Java, in questa sezione viene sempre illustrata una breve descrizione su quali saranno gli effetti dell’attacco appena programmato durante le battaglie. Il giocatore può inserire e rimuovere i pezzi sperimentando a suo piacimento, sapendo ogni volta con esattezza quale sarà il risultato del suo attacco. È importante notare che questa caratteristica snellisce decisamente il processo di apprendimento del giocatore che non dovrà leggere attentamente lunghi tutorial prima di comprendere come programmare un metodo, bensì imparerà i concetti semplicemente sperimentando con il posizionamento dei tasselli;
- Raccolta degli oggetti del giocatore: Nella parte bassa dello schermo è possibile selezionare i pezzi di codice (if, for, ed else if) o le classi di proprietà del giocatore. Sia i pezzi di codice che le classi possono essere trascinati nella sezione del puzzle per definire gli effetti della mossa.

Volendo quindi definire meglio gli oggetti di gioco di questa schermata:

- I "*pezzi di codice*" rappresentano i costrutti principali del linguaggio Java (e non solo) come “If”, “Else If” o “For” e permettono di rendere più flessibili le azioni collegate alle mosse del personaggio. Per esempio, i pezzi di codice di tipo if permettono di definire il comportamento della mossa a seconda del tipo dell’avversario. In questo modo, ad esempio, è possibile lanciare un attacco di tipo fuoco se il nemico è di tipo erba o un attacco di tipo acqua se il nemico è di tipo fuoco, massimizzando i danni inflitti e chiudendo lo scontro più rapidamente, in modo da esporsi il meno possibile agli attacchi nemici.
- Le "*classi*" sono i tasselli conclusivi del puzzle che compone la mossa. Infatti, a questi non è possibile collegare ulteriori pezzi, anche se non si è ancora superato il limite indicato nella parte sinistra dello schermo. Le classi servono a definire il comportamento della mossa e ne esistono di vari tipi: ci sono quelle che scatenano forti attacchi, quelle che permettono di recuperare punti vita o ancora quelle che permettono di alzare o abbassare le statistiche proprie o dei nemici.

Se capire che i metodi “Move1” e “Move2” siano collegati ai corrispettivi pulsanti nella sezione della battaglia risulta piuttosto intuitivo, un maggiore approfondi-

mento è necessario per comprendere quali siano gli effetti del costruttore. Il *costruttore*, infatti, è un metodo componibile esattamente come gli altri due ma si attiva una sola volta per battaglia, ovvero all'inizio, proprio come se per ogni battaglia un'istanza del personaggio venisse creata tramite il suo metodo costruttore.

A livello di gameplay, il vantaggio che il metodo costruttore offre è quello di attivare i suoi effetti senza dover aspettare il Reload Time come per gli altri metodi; qualunque sia il reload time del costruttore, viene ignorato e gli effetti vengono attivati subito dopo l'inizio della battaglia. Questa funzione può essere particolarmente utile per l'utilizzo di classi con effetti curativi o che aumentano le statistiche, in quanto presentano dei tempi di ricarica estremamente lunghi che, applicati ai metodi "Move1" e "Move2", porterebbero enormi svantaggi in battaglia.

3.2.8 Imprevisti e Tesori

I nodi della mappa, contraddistinti da un punto interrogativo, una volta cliccati, avviano l'evento "Imprevisto". Simile al concetto di "Imprevisto" del Monopoli, questa sezione di gioco mette il giocatore davanti ad una scelta che può portare a conseguenze positive o negative.

I nodi della mappa contraddistinti dall'icona di uno scrigno, presenti in quantità decisamente minori rispetto agli altri, attivano invece l'evento "Tesoro", che consente al giocatore di raccogliere un oggetto (pezzo di codice, pacchetto o una cospicua quantità di denaro) utile per facilitare la sua partita.

3.3 Tecnologie Utilizzate

3.3.1 La Scelta del Game Engine

Per realizzare un software, una volta aver definito il concept e le idee chiave alla sua base, risulta fondamentale selezionare gli strumenti necessari per avviare il processo di sviluppo.

Nel caso specifico dei videogiochi esistono svariate tipologie di strumenti da tenere in considerazione che variano in base a ciò che si vuole costruire, ma il cuore dello sviluppo è il game engine. Un "*game engine*" non è altro che un software progettato e realizzato allo scopo di supportare i programmati durante il processo di sviluppo di un videogioco.

Esistono tanti game engine sul mercato con caratteristiche completamente diverse tra di loro; tra i più popolari troviamo sicuramente Unreal Engine e Unity, due tool estremamente versatili che si adattano molto bene a qualunque tipologia di progetto e sono usati anche da importanti aziende del settore videoludico, ma esistono anche engine che forniscono strumenti molto specifici per alcune tipologie

di videogiochi come il Ren'Py Visual Novel Engine per la creazione di “romanzi visivi interattivi” oppure RPG Maker per la creazione di giochi di ruolo che usano la tecnica della pixel art. Ci sono anche casi in cui le software house, prima di sviluppare un videogioco, decidono di creare il proprio game engine per implementare funzioni specifiche o, semplicemente, per evitare le limitazioni economiche imposte dagli sviluppatori di game engine di terze parti. In un panorama così vasto di scelte, negli ultimi anni si è fatto strada un nuovo game engine che sta allargando sempre più la sua base di utenza e si sta candidando come valida alternativa ai leader del settore: il *Godot Engine*.

Godot è un game engine open-source, completamente gratuito, la cui prima versione è stata rilasciata nel 2014. Da allora, il software si è evoluto sempre di più diventando oggi un ottimo strumento per tutti coloro che vogliono sviluppare un videogioco in due o tre dimensioni.

Per lo sviluppo di Project Coding, dopo un’attenta analisi delle possibilità che il mercato dei motori di gioco mette a disposizione, è stato deciso di usare proprio il Godot Engine per una serie di motivazioni:

- La leggerezza del programma: se a primo impatto questa può sembrare una caratteristica banale, la capacità di Godot di avviarsi rapidamente e di funzionare in modo efficiente anche su hardware meno performanti lo rende una scelta imprescindibile per gli sviluppatori che non dispongono di computer di fascia alta. Infatti, i suoi concorrenti più famosi come Unity e Unreal sono entrambi software molto pesanti, che richiedono lunghi tempi di caricamento prima di poter mettere il programmatore nella posizione di lavorare. Inoltre, questa caratteristica di Godot lo rende altamente portatile; dato che non è necessaria un’installazione, è possibile caricare il programma (e magari la cartella con i propri progetti) su una chiavetta USB per poterla poi collegare ad un altro PC e continuare a lavorare, anche, ad esempio, usando laptop di fascia bassa. Consci di questo, gli sviluppatori hanno inserito all’interno del programma l’accesso alla documentazione ufficiale, in modo da poter essere consultata dagli utenti ovunque e in qualsiasi momento, anche senza una connessione ad Internet;
- Godot è open source: rispetto ai suoi concorrenti, Godot è completamente open source e ciò porta a numerosi vantaggi: prima di tutto, è possibile sviluppare videogiochi in modo completamente gratuito e senza vincoli rispetto a Unity (che è utilizzabile gratuitamente ma blocca alcune caratteristiche dietro abbonamenti mensili o annuali da pagare obbligatoriamente se il proprio gioco genera più di 100.000 dollari di incassi) o Unreal Engine (che è gratuito ma richiede di versare una percentuale uguale al 5 % dei guadagni che eccedono il milione di dollari derivanti dalla vendita di un software sviluppato con questo engine); Un altro beneficio è quello di poter accedere

al codice sorgente dell'engine e, volendo modificarlo a piacimento, potendo così aggiungere feature o risolvere bug. Si può anche condividere la scoperta di problemi o la volontà di aggiungere nuove funzioni all'engine con la più attiva community che sostiene il progetto per migliorare ulteriormente il software;

- Una community di sviluppatori sempre attiva: Come accennato prima, Godot gode di una community molto attiva che lavora all'engine con estrema dedizione. Esistono forum e pagine dedicate a Godot dove è possibile condividere idee o riportare bug per far in modo che il team principale dietro lo sviluppo dell'engine, o anche semplicemente un altro utente interessato, se ne occupi il prima possibile. La community di Godot, inoltre, è sempre molto disponibile e pronta ad aiutare novizi e veterani in difficoltà a risolvere i problemi che riscontrano durante lo sviluppo del loro gioco;

Ovviamente Godot non è perfetto; ad esempio, i risultati raggiunti sul lato estetico da Unreal Engine, come il controllo delle luci, la cura delle ombre e tanti altri dettagli che hanno reso il motore di gioco di Epic Games il leader del mercato non sono replicabili con un videogioco sviluppato con Godot Engine, ma questa non è una problematica che riguarda questo progetto dato che Project Coding punta ad essere un software relativamente leggero che possa girare su smartphone anche di fascia medio-bassa.

3.3.2 Altri Software

Per quanto il cuore dello sviluppo di un videogioco sia il game engine, risulta altamente complesso creare al suo interno ogni asset di cui si necessita per la buona riuscita del progetto.

Per la creazione di modelli 3D, siano questi personaggi, oggetti o luoghi, è stato utilizzato il software di modellazione 3D Blender.

In questo campo, non è stato difficile individuare l'alternativa migliore per raggiungere gli obiettivi del progetto nel campo della modellazione 3D: *Blender* è un software open source supportato ampiamente dalla community tramite creazioni di plug-in e di tutorial di ogni genere. Le possibilità che questo software mette a disposizione degli utenti sono innumerevoli e non è un caso che stia diventando uno standard dell'industria utilizzata anche da aziende affermate che operano nel campo della modellazione e dell'animazione.

Per quanto riguarda invece la creazione di sprite, icone, immagini e texture sono stati utilizzati due software diversi: Procreate e GIMP.

Procreate è un'applicazione per iPad che permette di utilizzare la propria Apple Pencil per realizzare disegni ed animazioni digitali. Grazie al suo costo estremamente accessibile di circa dieci euro e al suo ampio ventaglio di funzioni, questo

software si è rivelato fin da subito come un ottimo tool per donare un aspetto agli elementi bidimensionali di Project Coding.

GIMP è un software gratuito utilizzato per l'elaborazione digitale delle immagini. In questo contesto, è stato impiegato per ottimizzare i dettagli di immagini generate con Procreate e per definire le texture dei modelli creati in Blender. Il processo implicava l'esportazione delle mappe UV dei vari modelli da Blender in formato PNG, consentendo così di applicare i disegni desiderati direttamente sui modelli. Questa procedura è stata anche utilizzata per creare l'effetto "outline" su alcuni modelli, ossia il bordo nero che circonda i modelli, conferendo loro un aspetto "cartoon".

3.4 Concetti Base di Godot Engine

Non è obiettivo di questo capitolo insegnare al lettore il funzionamento di Godot, ma risulta necessario introdurre alcuni suoi concetti fondamentali per rendere comprensibili le scelte di design prese per lo sviluppo di Project Coding che verranno esplicate successivamente.

3.4.1 Le Scene

L'intero Godot Engine si basa sul concetto di “*scena*”. Una scena può essere vista come un contenitore che raggruppa oggetti, nodi e risorse (altri concetti base di Godot che verranno esplorati più tardi) o perfino altre scene, per creare una porzione specifica del gioco. Nella pratica, una scena può essere qualunque elemento del gioco: un personaggio, un'arma, un menù, un livello; basta raggruppare alcuni nodi in modo che formino un concetto e salvarli sotto forma di scena per renderli riutilizzabili in qualsiasi altra scena, in qualsiasi momento e in qualsiasi contesto. Il concetto di scena, quindi, promuove la riusabilità di porzioni di codice e di insiemi di nodi [2].

3.4.2 I Nodi

Una scena è composta da uno o più *nodi*, che possono essere considerati i mattoncini di base per la creazione di una scena in Godot. Ad esempio, per creare un personaggio, si inizia aggiungendo un nodo che fungerà da "genitore" per gli altri nodi del personaggio. Successivamente, si definisce l'aspetto del personaggio utilizzando un nodo "Sprite2D" per una rappresentazione bidimensionale, consentendo di dar forma al personaggio tramite un'immagine. Per gestire le collisioni del personaggio con altri oggetti nel mondo di gioco, si aggiunge un nodo "CollisionShape2D". È anche possibile includere una telecamera che seguirà i movimenti del personaggio attraverso un nodo "Camera2D" e continuare ad aggiungere altri

nodi e funzionalità secondo le necessità del personaggio. Salvare questo insieme di nodi come una scena, consente allo sviluppatore di riutilizzare tale gruppo in un'altra scena, con la possibilità di modificare tutte le istanze del personaggio agendo direttamente sulla scena originale.

Come intuibile dalle nomenclature usate nell'esempio, Godot mette a disposizioni una vasta varietà di nodi dalle caratteristiche variegate, con l'obiettivo di implementare le più classiche funzionalità che uno sviluppatore cerca durante la programmazione di un videogioco. Tuttavia, è sempre possibile partire da un nodo già esistente (il più basilare è "Node" da cui derivano tutti gli altri) per poter implementarne di propri con caratteristiche completamente modificabili. A tal proposito, è possibile "attaccare" ad ogni nodo del codice per definirne il comportamento tramite uno script [2].

3.4.3 GDScript

All'interno degli script è possibile definire il comportamento dei nodi tramite l'utilizzo di codice. Godot supporta ben quattro linguaggi di programmazione: C, C#, C++ e GDScript.

Fortemente ispirato a Python, *GDScript* è un linguaggio object-oriented a tipizzazione dinamica creato e ottimizzato appositamente per il Godot Engine. È altamente consigliato il suo utilizzo data la sua semplicità anche per i neofiti della programmazione. Inoltre, è il linguaggio più utilizzato dalla community, il che rende estremamente semplice trovare tutorial e guide, ufficiali o meno, che utilizzino questo linguaggio [2].

3.4.4 L'Albero della Scena

Si può pensare alla struttura di un gioco creato con Godot come un albero. In cima a questo albero c'è il nodo radice, mentre tutte le altre scene che compongono i livelli, i personaggi e gli oggetti di gioco sono figli di questo nodo o di suoi figli. Nella Figura 3.6 è illustrato, ad esempio, l'albero della scena dedicata al Luna's. Questo è il concetto alla base della struttura del Godot Engine, dove tutto è organizzato all'interno di uno "*Scene Tree*" [2].

3.4.5 Le Risorse

Precedentemente è stato affrontato l'argomento del "nodo" in Godot ma esiste un concetto altrettanto basilare e fondamentale per questo engine: la risorsa.

Un nodo si occupa di fornire funzionalità, come disegnare sprite, simulare la fisica o organizzare l'interfaccia utente. Le *risorse*, invece, non sono altro che "contenitori di dati" da cui i nodi possono ricavare informazioni utilizzabili per fornire le proprie funzionalità.

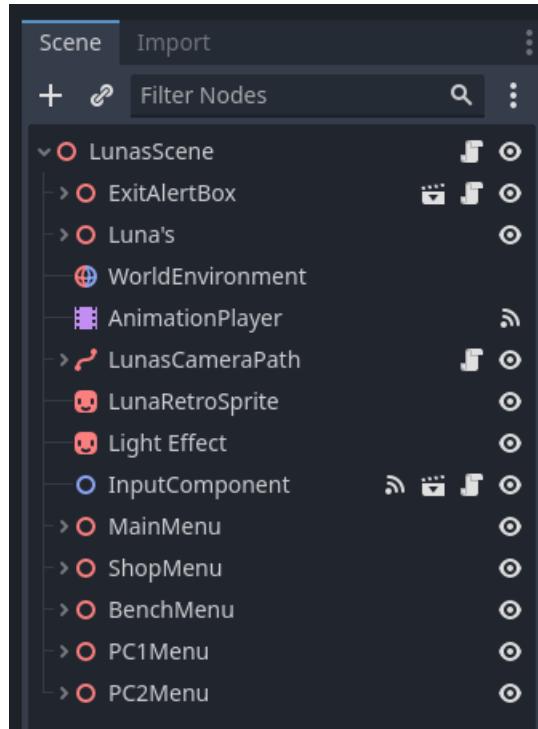


Figura 3.6: Albero della scena "LunasScene"

Alcuni esempi di risorse in Godot sono: texture, script, mesh, animazioni, font e tanto altro. È anche possibile creare delle risorse da zero utilizzando uno script; ad esempio, spesso viene consigliato di utilizzare una risorsa personalizzata per la gestione dei salvataggi (ma presto verrà spiegato perché questa operazione può comportare un rischio per la sicurezza dei futuri utilizzatori del gioco) [2].

3.4.6 I Segnali

Come in altri linguaggi di programmazione e altri game engine, anche in Godot esiste il concetto di “*segnalet*”. I nodi hanno la capacità di emettere segnali quando si verificano eventi specifici. Questi segnali possono essere catturati da altri nodi che si sono precedentemente registrati per riceverli o che sono stati collegati direttamente dallo sviluppatore utilizzando la sezione dedicata nell’editor di Godot.

Godot mette a disposizione una vasta gamma di segnali standard per ciascun tipo di nodo. Ad esempio, nel caso del nodo "Area2D", questo può rilevare quando la "CollisionShape" di un'altra area entra in contatto con la sua. Quando questo evento si verifica, il nodo "Area2D" emette un segnale noto come "area_entered". Questo segnale può essere intercettato da altri nodi o dallo stesso nodo, che ha la possibilità di attivare un comportamento specifico precedentemente definito in

una funzione dedicata.

Ovviamente è anche possibile creare segnali completamente personalizzati per qualsiasi tipologia di nodo.

Grazie ai segnali, è possibile implementare anche in Godot l'observer pattern, che, come verrà mostrato successivamente, può essere una risorsa di grande valore. In combinazione con le scene, infatti, aiuta a creare una struttura a basso accoppiamento che agevola notevolmente il riutilizzo del codice [2].

3.5 Architettura dell'Applied-Game

Dopo aver spiegato i concetti base del Godot Engine, è possibile approfondire più nel dettaglio l'architettura del progetto.

In questo capitolo, verranno esaminate le logiche fondamentali utilizzate nello sviluppo di vari aspetti di Project Coding. È importante notare che la creazione di questo videogioco educativo rappresenta un progetto esteso e complesso. Pertanto, è stato necessario condurre una selezione accurata degli aspetti più cruciali e fondamentali del software da presentare in questo capitolo al fine di evitare una spiegazione eccessivamente prolissa.

3.5.1 Scene Principali

È stato precedentemente evidenziato come alla base di ogni progetto sviluppato con Godot Engine ci siano le scene; Project Coding non fa eccezione.

L'applied game si suddivide principalmente in sei scene, ognuna rappresentante una determinata sezione del gioco: il Menu Iniziale, la Mappa, il Luna's, la Battaglia, l'Imprevisto e il Tesoro.

Inoltre, vi è un'altra scena fondamentale per il funzionamento del software: *Global*.

Global è una scena “*Autoload*”, ovvero un singleton sempre attivo all'interno del gioco, che ha lo scopo di mantenere alcune informazioni in memoria accessibili da qualunque altra scena. Ciò torna molto utile per rendere sempre disponibili variabili che hanno rilevanza per l'intera partita come il numero di monete raccolte o la quantità di salute del giocatore rimasta.

3.5.2 Passaggio di Scena

Un'altra questione di fondamentale importanza da tenere in considerazione è che, grazie al sistema di gestione della memoria di Godot, passare da una scena ad un'altra usando i metodi standard messi a disposizione dall'engine comporta l'eliminazione della scena precedente con tutti i suoi valori. Questa meccanica è molto utile per automatizzare la liberazione della memoria da dati non più necessari

ma comporta anche degli svantaggi. Tra tutti, quello più rilevante è sicuramente il passaggio di parametri tra una scena e l'altra che diventa impossibile se non tramite l'utilizzo di un Singleton apposito. Tuttavia, tale soluzione risulta poco elegante e va contro ogni pattern utilizzato durante la progettazione e lo sviluppo del software.

Un altro problema dei metodi messi a disposizione da Godot per il cambio della scena è quello dell'utilizzo di un singolo thread. Se ciò in progetti non particolarmente pesanti può non essere un problema, già solo per giochi che sfruttano le tre dimensioni e che, di conseguenza, necessitano di caricare ed elaborare un maggior numero di informazioni, rappresenta un ostacolo non da poco. Infatti, senza l'uso di un thread ausiliario, il gioco prova a caricare tutti gli elementi di una scena bloccando qualunque altra operazione. Il risultato è un software che non riesce a mostrare la benché minima animazione di caricamento e che appare, agli occhi di qualunque sistema operativo, come un'applicazione che “non risponde” per tutto il tempo di caricamento.

Per ovviare a tali problemi si è deciso di sviluppare un sistema di cambio scena partendo da zero che seguia questi semplici passi quando viene richiamato:

- quando si deve effettuare un cambio scena, la scena attuale chiama il metodo `change_scene_from_map` dello `SceneSwitcher` passando il nome della scena successiva e un dizionario che contenga i dati che vuole inviare;
- lo `SceneSwitcher` aggiunge all'albero delle scene una nuova scena, chiamata `TransitionScene`, che si occupa di mostrare al giocatore la schermata di caricamento;
- viene creato un nuovo thread che carica la scena successiva;
- durante il caricamento, il sistema può mostrare l'avanzamento dei lavori tramite un nodo di tipo “`ProgressBar`” della scena del caricamento (non è obbligatorio);
- Una volta terminato il caricamento, la nuova scena viene istanziata e, se presente, viene chiamato il suo metodo `_import_data_from_previous_scene` che gli permette di ricevere i dati passati dalla scena precedente;
- Dato che Godot non permette a Thread che non siano il principale di aggiungere o eliminare nodi dall'albero delle scene, vengono caricati da un nuovo Thread anche i nodi che andrebbero aggiunti a tempo di compilazione con l'istanziazione di una scena (ad esempio: la scena della battaglia non conosce quale sia il nemico da caricare se non prima di ricevere il suo id e altri dati dalla scena della mappa) per poi essere aggiunti anch'essi all'albero delle scene dal thread principale;

- Una volta che tutti gli elementi sono stati caricati e aggiunti all'albero, Godot attiva l'animazione di “fade out” della TransitionScene e permette al giocatore di visualizzare la nuova scena.

3.5.3 Gestione della Mappa

La MapScene è responsabile interamente della gestione della mappa durante una partita. Questo implica che, dopo aver selezionato un livello, è incaricata di generare la mappa e di mantenerla costantemente aggiornata mentre il giocatore progredisce nell'avventura.

La mappa è una griglia con un numero variabile di colonne e righe, di solito 5 colonne e 20 righe. Per ogni casella della griglia, il sistema determina se deve apparire un nodo e assegna un tipo di evento. Inoltre, genera tutte le variabili necessarie per gestire quel tipo di evento (ad esempio, per un nodo di tipo battaglia viene scelto dal sistema un nemico ed il suo livello, ovvero un valore che ne determina le statistiche.) Ovviamente, per quanto la generazione della mappa sia basata sulla casualità, sono stati inseriti alcuni vincoli, come ad esempio la presenza di almeno un nodo per ogni riga.

Il generatore della mappa è altamente flessibile grazie a una fase di progettazione e programmazione attentamente pianificata. Questo approccio ha consentito non solo di creare un sistema di generazione casuale e coerente dei nodi, ma anche di garantire la futura adattabilità a situazioni ed esigenze diverse. Più precisamente, tramite la semplice modifica del valore di alcune variabili, aggiornabili tramite l'editor di Godot, come illustrato nella Figura 3.7, è possibile scegliere: la dimensione della mappa specificando il numero di righe e di colonne, i tipi di nodi che possono essere creati e la relativa probabilità di essere scelti dal sistema, la probabilità che un nodo appaia in una casella e di quanto quella probabilità aumenti o diminuisca ogni volta un nodo venga inserito o meno e ancora tanti altri parametri che permettono una grande adattabilità della mappa a situazioni diverse.

Una mappa è composta da *nodi*, ciascuno rappresentato graficamente da un cilindro schiacciato di dimensioni ridotte e da un'icona che ha lo scopo di segnalare l'evento ad esso collegato. Un nodo è composto da:

- uno stato che indica se è accessibile dal giocatore, se è stato precedentemente visitato e altre informazioni rilevanti;
- un tipo, che permette al giocatore di comprendere a quale evento tale nodo dia inizio;
- un id;
- un dizionario di variabili specifiche per il tipo di evento ad esso collegato;

MapScene.gd		
Map Columns	5	◆
Map Rows	20	◆
Types	Dictionary (size 4)	
Type Probability Shifts	Dictionary (size 4)	
Node Probability	40	◆
Node Probability Increase	10	◆
Node Probability Decrease	15	◆
Node Connection Range	2	◆
Path Probability	50	◆
Path Probability Increase	15	◆
Path Probability Decrease	15	◆
Node Area	x 2.5 y 4	
Enemy Level Max Variation	2	◆

Figura 3.7: Lista di parametri personalizzabili per la generazione della mappa.

- un array di numeri rappresentanti la colonna di cui fanno parte i nodi direttamente raggiungibili da questo;

Inoltre, il nodo ha il compito di gestire un eventuale click su di esso. Quando è nello stato "reachable", è possibile selezionarlo per avviare l'evento correlato.

I nodi sono collegati tra loro da percorsi. Anche i percorsi sono generati casualmente (sempre con il rispetto di alcuni vincoli). Ciò significa che anche se due nodi avessero le proprietà per essere collegati, la creazione di un percorso tra di loro non sarebbe per nulla scontato.

Anche i *percorsi* presentano uno stato che influenza il loro aspetto grafico. Questo stato varia in base alla posizione del giocatore sulla mappa e alle strade precedentemente attraversate.

Come è stato precedentemente spiegato, quando si passa da una scena all'altra, sia tramite i metodi che tramite il sistema creato per questo progetto, la scena precedente viene eliminata e con essa tutti i suoi dati. Ciò vuol dire che avviare un evento cliccando su un nodo della mappa e quindi caricare la scena corrispondente equivale ad eliminare la MapScene con tutte le informazioni relative ai nodi e ai percorsi. È risultato dunque necessario trovare una soluzione per salvaguardare la conservazione di tali informazioni, in quanto fondamentali per mostrare nuova-

mente al giocatore la mappa di gioco dopo il completamento di un evento. Per raggiungere tale risultato, sono state individuate varie possibilità:

- Passare la mappa tra le scene: Una soluzione poteva essere quella di passare la mappa come una variabile tra scene ma ciò avrebbe comportato il continuo invio di questa struttura potenzialmente pesante a scene che non ne necessitano se non per ripassarla in un secondo momento alla MapScene;
- Salvare la mappa in Global: Una seconda soluzione era rappresentata dal salvataggio in Global della struttura, il che avrebbe senso dato che questa scena autoload è utilizzata per conservare variabili a cui è necessario accedere durante l'intera partita. Tuttavia, questo avrebbe significato mantenere costantemente in memoria una mole di informazioni che poteva essere esigua come gigantesca a seconda della grandezza della mappa che, è importante ricordare, è completamente modificabile;

Alla fine, si è quindi optato per creare una risorsa contenente tutte le informazioni necessarie che possa essere salvata in una cartella di gioco per poi essere caricata e modificata solo quando necessario.

3.5.4 Generazione della Descrizione del Metodo

Un'altra feature fondamentale di Project Coding che vale la pena esplorare è la generazione della descrizione durante la sezione di programmazione delle mosse del giocatore. Aggiornare la spiegazione ad ogni tassello del puzzle aggiunto o rimosso si è rivelata una sfida molto ardua ma, grazie ad una efficace strutturazione del sistema di gestione delle mosse, il risultato raggiunto risulta più che soddisfacente.

Come altre variabili rilevanti per la durata di intere partite, anche le mosse del giocatore, ovvero le azioni eseguite dal personaggio del giocatore durante una battaglia dopo la pressione del tasto apposito, vengono memorizzate nella scena Global. Più precisamente, in Global è possibile interagire con un dizionario denominato “*moves*” contenente una risorsa “*Move*” per ogni mossa (costruttore, prima mossa e seconda mossa).

La risorsa Move, oltre a svariati metodi, contiene tre variabili:

- time: indica il Reload Time della mossa;
- pieces_register: è un registro che contiene tutti i tasselli di puzzle utilizzati per comporre la mossa;
- paths_register: un dizionario speciale che contiene tutti i possibili “percorsi logici” formati dalle combinazioni dei tasselli.

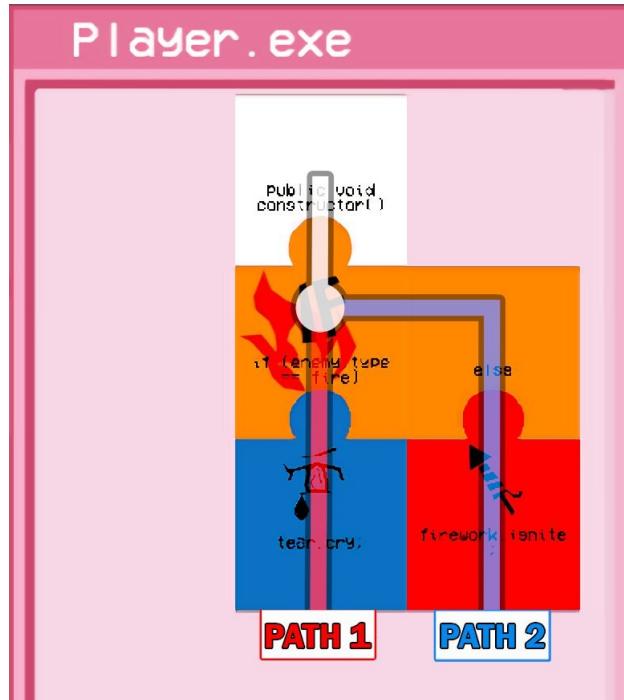


Figura 3.8: Esempio di generazione di percorsi dato un determinato metodo.

La struttura del registro dei percorsi è ciò che rende possibile il continuo aggiornamento della descrizione.

Per capire il suo funzionamento, si consideri la programmazione della mossa tramite il menu per la definizione del metodo presentato precedentemente: qui vengono utilizzati tasselli di puzzle per definire il comportamento di tale mossa a seconda della situazione. Se la mossa non contiene un tassello di tipo “If”, il comportamento di tale metodo risulta sempre lo stesso in qualunque situazione; al contrario, quando sono presenti uno o più “If”, la mossa può avere esiti diversi a seconda del tipo del nemico. Si può comprendere l’effetto di ogni esito seguendo il “percorso logico” che parte dal tassello iniziale fino ad arrivare ad una classe o ad un tassello che non ha ulteriori collegamenti. Per una migliore comprensione, si prenda in considerazione l’esempio mostrato nella Figura 3.8; qui viene illustrata una mossa che ha come primo tassello un “If” che controlla se il nemico è di tipo fuoco, a cui, a sua volta, sono collegate due classi denominate “Tear” e “Firework”, rispettivamente tramite lo slot dell’if e lo slot dell’else. Questa mossa, quindi, presenta due percorsi logici: il primo che viene percorso solo se il nemico è di tipo fuoco e che termina con la classe “Tear” e il secondo che viene percorso in tutti gli altri casi e che termina con la classe “Firework.” Compreso questo ragionamento, si può intuire la funzione di “paths_register”.

“*paths_register*” è un dizionario di elementi che rappresentano i “percorsi logici” che hanno come chiave il “tipo” di cui il nemico deve essere (fuoco, acqua, erba o normale) per scegliere tale percorso, mentre i loro valori sono costituiti: dall’id della classe con cui tale percorso si conclude, dal numero di iterazioni della mossa data dalla quantità e dalle caratteristiche dei pezzi di codice di tipo “for” presenti sul percorso e da un array chiamato “*checked_types*” che viene utilizzato per gestire la combinazione di più tasselli di tipo “if”.

È importante notare che tra le chiavi, oltre ai possibili tipi dei nemici, può essere presente anche la stringa “empty”, che specifica gli effetti della mossa quando non vi sono altri percorsi che abbiano come chiave il tipo del nemico.

Il *paths_register* quindi non contiene esattamente una catena di tasselli di codice come ci si potrebbe immaginare, bensì per ogni possibile esito della mossa calcola e mantiene in memoria solo le informazioni fondamentali.

Tali informazioni vengono poi lette da un metodo del modulo dedicato alla creazione delle mosse facente parte della scena “Luna’s”, in quanto presenti in Global, e trasformate in frasi che formano la descrizione. Ciò è possibile grazie all’accostamento di alcune informazioni dei percorsi, come il tipo del nemico o il numero di iterazioni della mossa, a delle frasi pre-impostate e salvate in un documento di testo che vengono opportunamente recuperate e, in seguito, assemblate dal sistema per creare frasi di senso compiuto.

Le descrizioni sono disponibili sia in italiano che in inglese.

Il *paths_register* di Move non viene utilizzato solo per generare la descrizione durante la creazione della mossa; grazie al fatto che le caratteristiche della mossa vengono facilmente esposte attraverso questa metodologia, le azioni delle mosse stesse durante le battaglie si basano su questa struttura per attivare gli effetti che il giocatore ha programmato.

3.6 Segnali e Componenti

Project Coding fa largo uso dei concetti tipici dell’architettura “Component-Based”, di solito combinati con quelli dell’*observer pattern*, che in Godot è realizzabile tramite l’uso dei segnali, al fine di creare dei componenti che, inseriti come nodi figli di altri nodi, mettono a disposizione di quest’ultimi nuove funzionalità senza la necessità di scrivere ulteriore codice.

Alcuni esempi dell’uso di questi componenti nel progetto sono evidenti nella scena del Luna’s, dove è stato fatto ampio uso dei ButtonComponent. Questi componenti consentono a un nodo di tipo Sprite3D di fungere da pulsante con animazioni sia alla pressione che al rilascio, oltre a lanciare segnali che vengono successivamente intercettati da altri nodi per eseguire azioni correlate a tale input.

I ButtonComponent non sono stati però gli unici ad essere ampiamente utilizzati

nella scena dedicata al Luna's; Ad esempio, è stato sviluppato un DragAndDrop-Component che rende l'oggetto di cui è figlio trascinabile per lo schermo e lancia segnali alla pressione e al rilascio.

La scena dedicata al Luna's mostra già i vantaggi dell'utilizzo dell'architettura basata sui componenti, il cui enorme potenziale, però, diventa decisamente più evidente nella "BattleScene".

In questa scena, molti elementi sono organizzati in modo tale da essere composti esclusivamente da componenti, eliminando la necessità di scrivere codice personalizzato, a meno che non sia richiesta una funzionalità molto specifica. Un personaggio deve avere un sistema che gestisca la salute e per questo viene utilizzato un "*HealthComponent*", ma è necessario anche tener traccia di eventuali potenziamenti momentanei alle statistiche, e ciò viene fatto tramite lo "*StatusComponent*". In sostanza si può creare un component per ogni caratteristica, dalla gestione delle collisioni alle animazioni dei modelli 3D, e ciò permetterebbe l'aggiunta di funzionalità senza la creazione di codice al di fuori da quello dei componenti. Ciò che rende quest'approccio speciale è che i componenti che sono stati creati per il personaggio possono poi essere utilizzati per qualunque altro oggetto di gioco.

Se si vuole tenere traccia della vita di un nemico basta aggiungere al suo albero lo stesso "*HealthComponent*" usato per il personaggio; se si vuole creare un oggetto della scena che al contatto con un oggetto sparisce basta aggiungergli ai nodi figli un "*HitboxComponent*". Così facendo, la creazione di nuovi elementi di gioco diventa estremamente più semplice ed efficace, con la scrittura di nuovo codice che si riduce solo alla programmazione di peculiarità dell'elemento in questione.

Un possibile svantaggio di questo approccio potrebbe essere la complessità nel facilitare la comunicazione tra i componenti affinché eseguano le loro funzioni in maniera precisa.

Come accennato in precedenza, un modo eccellente per facilitare la comunicazione tra i componenti e promuovere il "low coupling" è l'uso dei segnali. Tuttavia, se i segnali non vengono collegati ai ricevitori in anticipo, il sistema non funzionerà come previsto. Ed è qui che entrano in gioco alcune architetture comunemente utilizzate nell'ambito di Godot.

Una soluzione iniziale potrebbe essere quella di collegare l'emittente del segnale e il ricevitore tramite l'editor stesso di Godot. Tuttavia, in questo caso, considerando che il personaggio, il nemico e tutti gli oggetti di gioco vengono creati durante l'esecuzione del gioco (a runtime), non è possibile conoscere in anticipo quali componenti dovranno comunicare tra di loro.

È qui che entrano in gioco quelli che sono stati denominati "*Global Signals*". Come deducibile anche dal nome, questo approccio combina la scene autoload con il concetto di segnale: gli emittenti dei segnali non inviano più il segnale direttamente ai ricevitori, bensì utilizzano una classe globale a cui, da ora in poi, ci si riferirà

con il nome di SignalManager. Allo stesso modo gli emittenti non si collegheranno più direttamente al nodo emittente, bensì alla nuova scena globale. Quando un emittente lancia un segnale, il SignalMangaer lo riceve e lancia a sua volta un segnale con lo stesso nome che verrà poi ricevuto da tutte le classi che hanno fatto richiesta di ricevere tale segnale.

In generale questo approccio semplifica di molto la gestione dei componenti, che così non necessiteranno di alcun riferimento ai ricevitori per funzionare. Tuttavia, nel caso specifico della scena della battaglia questo approccio potrebbe creare delle grosse complicazioni. Come citato precedentemente, lo stesso componente (non la stessa istanza ma la stessa scena) può essere figlio di più di un nodo (personaggio, nemico o oggetto generico di gioco.) Dato che i componenti condividono lo stesso codice, tutti faranno richiesta di collegarsi allo stesso segnale e ciò può provocare comportamenti indesiderati. Si consideri, ad esempio, una situazione di battaglia tra il personaggio del giocatore e un nemico. Quando il nemico subisce un colpo, l'HitboxComponent lancia un segnale specifico che viene intercettato dal SignalManager e inoltrato a tutti i ricevitori interessati. In questa situazione, si potrebbe desiderare che l'HealthComponent risponda al segnale quando il nemico perde salute a causa del contatto con un oggetto. Tuttavia, se tutti gli HealthComponent si collegano allo stesso segnale, essi risponderanno tutti alla situazione, coinvolgendo sia il componente responsabile della gestione della salute del nemico che quello del giocatore.

Per risolvere questa situazione sono state individuate due soluzioni:

- Dato che i segnali hanno la capacità di passare dei parametri, sarebbe possibile inviare anche un riferimento o l'id dell'emittente in modo che, quando un componente riceve il segnale, può scegliere se ignorarlo o meno a seconda di tale valore. Questa soluzione sicuramente risolve il problema iniziale ma ne crea uno prestazionale: se si vuole lanciare un singolo segnale verso il componente di un determinato elemento, questo segnale verrà moltiplicato ed inviato a tanti nodi quanti sono quelli che contengono lo stesso componente. In altre parole, per inviare un singolo segnale si rischia di lanciarne molti di più, soprattutto per situazioni con molti elementi a schermo;
- Si potrebbe modificare il SignalManager in modo da gestire i segnali dei singoli elementi separatamente. Più precisamente si potrebbe creare un dizionario nella scena SignalManagerArchive che contenga una risorsa avente le stesse funzionalità del vecchio SignalManager per ogni oggetto a schermo. Ciò permetterebbe ai componenti di registrarsi presso il SignalManager dedicato accedendogli passando l'id dell'istanza dell'elemento di cui fanno parte al dizionario del SignalManagerArchive ed eliminando definitivamente il problema dei segnali multipli della precedente soluzione. Tuttavia, anche

in questo caso è necessario fare delle considerazioni: creare un SignalManager per ogni elemento equivale ad occupare maggiore spazio in memoria ed è per questo che diventa fondamentale liberare spazio ogni volta che ce n'è l'opportunità. Quindi, quando un elemento di gioco viene eliminato, diventa di grande importanza gestire anche l'eliminazione del relativo SignalManager all'interno del dizionario.

In conclusione, sia la prima che la seconda soluzione presentano vantaggi e svantaggi da considerare. Anche se, data la natura del gioco, non dovrebbero mai verificarsi situazioni che prevedano la presenza in contemporanea di un numero di elementi talmente elevato da rendere rilevante il peso di questi approcci, sono state analizzate attentamente entrambe le alternative. Tuttavia, per affrontare la sfida della gestione dei segnali, la seconda soluzione è stata preferita. Nonostante la sua complessità aggiuntiva, essa risulta essere più organizzata e richiede meno risorse della CPU, il che la rende la scelta migliore in questo contesto.

3.7 Dati Statici e Salvataggi

3.7.1 Gestione dei Dati Statici

Durante lo sviluppo di Project Coding, è emersa la crescente necessità di implementare un sistema per la gestione dei dati statici al fine di organizzare in modo efficiente una vasta gamma di dati di natura diversa. Il gioco Project Coding include numerosi oggetti di diverso tipo, ciascuno con un insieme significativo di caratteristiche da gestire. Un esempio ne sono i pacchetti, i quali hanno un nome, un prezzo, una rarità, delle descrizioni in più lingue e le classi contenute. Ogni pacchetto è diverso dall'altro, il che si traduce nella necessità di sviluppare una struttura dati per conservare tutte le informazioni di questi oggetti.

L'esempio è stato fatto con i pacchetti ma ci sono molti altri oggetti con caratteristiche uniche di cui tener conto come: pezzi di codice, classi, nemici, personaggi giocabili e tanto altro.

Dopo aver esaminato diverse alternative, è stato deciso di adottare l'uso di file JSON per archiviare questi dati. Questa scelta consente l'utilizzo di software esterni per l'inserimento rapido ed efficiente degli elementi. Inoltre, Godot offre un solido supporto per i file JSON, che sono facilmente leggibili e possono essere convertiti in dizionari utilizzando la classe JSON. Questa classe permette anche la creazione di file JSON da zero e l'inserimento di variabili di gioco al loro interno.

3.7.2 Gestione dei Salvataggi

Godot non presenta una vera e propria funzione di salvataggio, lasciando al programmatore l'implementazione della soluzione che meglio si adatti alle sue esigenze.

Molti sviluppatori suggeriscono di utilizzare una risorsa apposita contenente tutti i dati necessari da salvare in una cartella di sistema in modo da poterla aggiornare e caricare velocemente. Questa è stata anche la soluzione adottata per Project-Coding, che, ad ogni cambio di scena, carica, modifica e salva i dati della risorsa SaveData che contiene tutte le informazioni necessarie per riprendere una partita in un secondo momento e le variabili generali non legate ad una singola partita come il livello del personaggio o il numero di monete totali.

È necessario, tuttavia, fare una considerazione: per quanto salvare i dati in una risorsa sia la soluzione più facile e più veloce in termini di prestazioni, potrebbe non essere ideale per tutti i giochi. Più specificamente, il problema delle risorse usate come file di salvataggio consiste nella loro capacità di poter eseguire codice arbitrario. Se a questo si aggiunge che un'applicazione creata con Godot può far leggere al sistema operativo qualunque tipologia di codice tramite l'utilizzo della classe OS, ci si può facilmente rendere conto dei pericoli di questo metodo.

In generale, se non si prevede di permettere al giocatore di importare salvataggi di altre persone, il problema non si pone. Tuttavia, se ciò fosse possibile, una persona potrebbe scaricare i dati di gioco caricati su internet di un altro utente che contengono del codice malevolo ed esporsi così a potenziali attacchi informatici.

L'approccio più sicuro quindi è, al momento, l'utilizzo di un file JSON, che non permette l'esecuzione di codice arbitrario ed elimina del tutto i rischi. Tuttavia, un file JSON è ben più pesante di una risorsa, quindi per grandi quantità di dati non ne è consigliabile l'utilizzo.

Nei forum si sta discutendo dell'utilizzo di tecniche più complicate che integrino la sicurezza dei file JSON con la praticità delle risorse; di conseguenza, è probabile che presto verrà trovata una soluzione a tale problema.

Detto ciò, Project Coding non intende incentivare la condivisione dei salvataggi in quanto, tramite questo trucchetto, sarebbe possibile saltare intere sezioni di gioco, minando il percorso di apprendimento. Per tale motivo, l'applied game fa uso di un sistema di salvataggio basato sulle risorse.

Capitolo 4 - Risultati

Questo capitolo esamina come Project Coding abbia integrato efficacemente le informazioni ottenute dall’analisi dei giochi educativi precedentemente discussi nella sezione ‘Related Works’ del Capitolo 2, dedicando a ciascuno di essi un sottocapitolo. Questo processo ha permesso di sviluppare e implementare una serie di funzionalità volte a risolvere le criticità emerse da tali analisi, sfruttando al contempo le opportunità e le caratteristiche positive identificate in quei prodotti.

4.1 Robocode

Nel mare dei giochi educativi relativi alla programmazione, Robocode riesce a distinguersi dagli altri grazie alla sua struttura di gioco. Spesso, molti prodotti inducono il giocatore a risolvere enigmi attraverso la programmazione, rappresentata in diverse modalità. Tuttavia, Robocode esce da questa convenzionale via tracciata, distintiva di molti titoli con lo stesso obiettivo, suddividendo il gameplay in due fasi: una dedicato alla programmazione del proprio carrarmato e l’altra dove è possibile effettivamente giocare, vedendo i risultati del proprio lavoro tramite i movimenti e le reazioni della propria “creatura”.

Questa struttura si dimostra altamente motivante, poiché il giocatore non programma solo per superare un livello e ricominciare da zero per affrontare il successivo. Al contrario, la programmazione del carrarmato persiste anche dopo ogni battaglia e può essere utilizzata in quanti scontri si desidera. L’aspirante programmatore ha la possibilità di continuare a modificare e migliorare il codice, partendo dal lavoro precedentemente svolto. Questo permette di percepire il risultato del proprio lavoro come qualcosa di durevole nel tempo e, di conseguenza, di valore.

Project Coding fa suoi questi concetti strutturali del gioco del 2000, modificandoli opportunamente per adattarli ai propri scopi. Come Robocode, infatti, anche questo progetto si basa su una struttura doppia, dove in una prima fase, all’interno del Luna’s, si richiede al giocatore di “programmare” le proprie mosse per poi, durante la battaglia, poter utilizzare il frutto del proprio lavoro per combattere contro i nemici. Anche in questo caso, il giocatore non è chiamato a scrivere un codice specifico per risolvere una singola situazione, come accade ad esempio in “Rabbids: Coding!”, ma anzi, è esortato a prepararsi ad ogni evenienza: un giocatore non sa quale nemico si ritroverà davanti o quali sfide lo attendano lungo il suo percorso ed è per questo che deve prepararsi ad ogni evenienza durante la fase di programmazione. Tale concetto è evidente se si analizza il ruolo del pezzo di codice “if” nell’economia di gioco: questo pezzo può essere utilizzato per far mutare il comportamento di una mossa in base al tipo del nemico incontrato in battaglia e

permettersi così di porsi in una posizione di vantaggio in qualunque situazione. Un giocatore non è a conoscenza di che tipo saranno i nemici che incontrerà durante la partita, ma può utilizzare gli “if” per avere sempre a disposizione una mossa che risulti molto efficace contro ogni tipologia di avversario.

Un grosso problema di Robocode precedentemente discusso è quello relativo alla totale assenza di guide in merito al linguaggio Java o alla programmazione in generale. Tale questione rende praticamente impossibile, per un neofita, affacciarsi al mondo di gioco senza una guida esterna, come un insegnante o una serie di tutorial recuperabili su YouTube. Questo aspetto scoraggia fin da subito qualunque giocatore non abbia una volontà abbastanza forte da spendere ore a documentarsi e a studiare per usufruire del software, il che risulta in una grossa limitazione del potere educativo del videogame.

Project Coding ha l’obiettivo di far avvicinare alla programmazione anche persone lontane da questo mondo e quindi è risultato fondamentale correggere tale aspetto. La risposta a questa problematica è data, in primo luogo, dalla semplificazione della sezione di programmazione che sostituisce il codice puro con dei tasselli di puzzle, ma soprattutto dalla presenza di una descrizione nella sezione di programmazione della mossa che sia sempre in grado di fornire al giocatore una spiegazione degli effetti della mossa che sta componendo. Così facendo, il giocatore che non vuole informarsi sul funzionamento di certi costrutti (tramite delle spiegazioni presenti nel gioco o usufruendo di aiuti esterni) può semplicemente trascinare alcuni pezzi di codice nel puzzle e vedere i loro effetti nella descrizione. Con questo approccio, il giocatore inizialmente fa affidamento completamente alla descrizione, sperimentando combinazioni di vari pezzi per capire quale possa restituire la soluzione migliore. Dopo numerose partite però, il giocatore capisce quali effetti sono collegati ai vari pezzi e non perde più tempo a combinare tasselli in modo casuale, ma può valutare la situazione direttamente analizzando gli oggetti a sua disposizione; quando ciò accade, il giocatore ha acquisito le competenze relative a quei costrutti senza neanche accorgersene e l’obiettivo del gioco è raggiunto.

Un altro punto debole di Robocode è il lato estetico che, oltre ad essere poco chiaro nell’illustrare all’utente le funzionalità del software, rende chiaro l’anno di rilascio del titolo tramite schermate dall’aspetto obsoleto. Se ciò può non essere un problema per un videogame che punta su una community numerosa ma storica, per un progetto come Project Coding che punta principalmente ad un pubblico giovane non curare il lato estetico sarebbe inaccettabile.

Project Coding presenta un certo studio sotto questo punto di vista rendendo chiara ogni funzione del gioco, inserendo personaggi che possano accompagnare il giocatore di schermata in schermata e, in generale, facendo in modo che ogni componente del gioco risulti gradevole all’occhio.

L’idea di Robocode di suddividere la sua struttura di gioco in due fasi è risultata

vincente. Tuttavia, la sezione del combattimento, in cui i giocatori osservano passivamente la battaglia sullo schermo senza la possibilità di interagire attivamente, potrebbe apparire antiquata in un contesto videoludico moderno. Per quanto abbia senso nel contesto in cui è sviluppata, tale meccanica va contro il concetto stesso di videogioco, basato principalmente sull'interattività tra il giocatore e il mondo di gioco. Per questo motivo, durante la fase di battaglia, Project Coding non si limita a far osservare al giocatore i risultati del codice creato in precedenza, ma lo chiama vivamente all'azione. Infatti, l'utente deve evitare tutti gli attacchi che il nemico vuole scagliargli contro e deve attuare strategie precise per utilizzare le mosse in modo vincente.

Il giocatore non assume mai un ruolo "passivo" durante la partita, ma è costantemente coinvolto in azioni, sia istintive come schivare gli attacchi nemici, sia riflessive come ideare strategie per utilizzare al meglio le mosse e sconfiggere il nemico. Questa struttura accentua l'importanza di una corretta programmazione delle mosse: avere a disposizione una mossa efficace contro il nemico consente di concludere lo scontro rapidamente, di esporsi meno agli attacchi nemici e di ridurre il tempo in cui il giocatore prova la sensazione di tensione data dal desiderio di non perdere preziosi punti vita.

4.2 Rabbids: Coding!

Come mostrato in precedenza, Rabbids: Coding! sostituisce il codice Java con una struttura che possa risultare più familiare a utenti che non hanno mai avuto esperienze di programmazione: il puzzle.

Essendo un'ottima soluzione per avvicinare i giovani alla programmazione e ormai consolidata come meccanica di base in numerosi programmi didattici informatici, la metafora "programmazione - puzzle" viene impiegata anche in Project Coding per semplificare il processo di programmazione delle mosse.

Una delle funzionalità che più colpisce di "Rabbids: Coding!" è sicuramente la volontà di premiare i giocatori che riescono ad ottimizzare il proprio codice nel miglior modo possibile. Nel gioco sviluppato da Ubisoft, ciò avviene attraverso l'assegnazione di stelle. Tuttavia, come precedentemente discusso, queste stelle hanno un impatto minimo sull'esperienza complessiva del giocatore, attenuando l'efficacia di questa buona idea.

Project Coding vuole raggiungere lo stesso obiettivo ma usando metodi differenti. Mentre in Rabbids: Coding! per superare un livello è necessario creare una sequenza di blocchi che funzioni solo per una determinata situazione, in Project Coding la situazione è nettamente diversa. Programmare in modo inefficiente una mossa si traduce nella incapacità, in battaglia, di infliggere notevoli danni al nemico. Ciò non rende impossibile sconfiggerlo, ma per arrivare a tale obiettivo sarebbero necessari tempi decisamente più lunghi, il che si tradurrebbe nell'esporsi a molti

più rischi, aumentando le possibilità di ricevere ingenti danni.

Si consideri, ad esempio, una mossa composta da 4 tasselli: un pezzo di codice "for 2" che itera due volte il codice ad esso collegato, un pezzo di codice "if fire" che specifica un'azione nel caso il nemico sia di tipo fuoco, e due classi, "Pistola d'Acqua" e "Lanciafiamme", collegate rispettivamente al primo e al secondo slot dell'"if"; la posizione del "for" può fare una differenza significativa. Se inserito all'inizio del codice, prima dell'"if", il codice verrebbe iterato due volte, consentendo al giocatore di lanciare due attacchi consecutivi invece di uno, indipendentemente dal tipo del nemico. D'altra parte, se il "for" fosse attaccato al primo slot dell'"if", il giocatore avrebbe la possibilità di lanciare due volte lo stesso attacco solo nel caso la condizione dell'"if" fosse vera, depotenziando di fatto il proprio potere offensivo e rallentando della metà i tempi di abbattimento dei nemici aventi un elemento diverso dal fuoco.

In altre parole, anche Project Coding premia i giocatori che riescono ad ottimizzare il proprio codice, ma non direttamente attraverso assegnazione di premi, bensì indirettamente, seguendo la naturale logica del linguaggio Java e di molti altri linguaggi di programmazione.

4.3 Outcore: Desktop Adventure

Nel corso dell'analisi del mini-gioco Idle Game di Outcore: Desktop Adventure, è stata evidenziata una notevole differenza in termini di efficacia tra la raccolta di monete in questo titolo e la conquista di stelle in "Rabbids: Coding!". Per quanto i sistemi di ricompensa siano diversi anche solo concettualmente, non si può fare a meno di notare che le due valute impattano sui giocatori in maniera decisamente diversa. Le monete del primo titolo permettono di acquistare potenziamenti, accedere ad altri livelli e di svelare segreti, mentre le stelle del titolo di Ubisoft non hanno uno scopo al di fuori di sbloccare una nuova modalità solo dopo aver completato, con il massimo dei voti, tutti i livelli del titolo. In altre parole, in Rabbids: Coding! la ricompensa arriva solo e unicamente al completamento dell'intera esperienza di gioco mentre Outcore premia il giocatore continuamente e in modi sempre diversi.

Appreso ciò, anche Project Coding ha introdotto una valuta di gioco. Alla fine di ogni battaglia, gestendo in modo adeguato gli imprevisti o apprendo tesori, è possibile ottenere monete che possono essere successivamente spese nel Luna's per acquistare pacchetti il cui contenuto è avvolto nel mistero, per far propri pezzi di codice in grado di potenziare gli attacchi o perfino per recuperare punti vita in previsione delle prossime battaglie. Le monete possono essere anche usate in specifici imprevisti, ma non sempre privarsi dei propri risparmi può rivelarsi la mossa migliore. Alla fine di una partita, inoltre, a seconda delle prestazioni del giocatore, è possibile ottenere una certa quantità di punti spendibili per aumentare

permanentemente il livello del personaggio, aumentandone così le statistiche. In altre parole, anche Project Coding cerca di ricompensare quando necessario il giocatore per fornirgli la motivazione di continuare il proprio viaggio ed apprendere così nuove nozioni legate al linguaggio Java.

Il motivatore cardine di Outcore: Desktop Adventure, però, è principalmente legato al rapporto che si instaura tra il giocatore e il personaggio principale dell'intera avventura: Lumi. Il giocatore è portato a voler sapere sempre di più di quella ragazza che ha invaso il suo desktop e, per farlo, è pronto ad affrontare tutte le sfide che il gioco gli lancia.

Per quanto le premesse narrative di Project Coding siano decisamente diverse da quelle di Outcore, che è un'esperienza estremamente singolare che punta molto sulla rottura della quarta parete, anche questo progetto cerca di inserire personaggi con cui sia possibile instaurare un rapporto emotivo. Attualmente, è presente solo Luna, la proprietaria del Luna's, che accompagna il giocatore durante ogni operazione all'interno del locale. Il suo ruolo consiste nell'assistere il giocatore in ogni operazione e intervenire quando necessario. Tuttavia, c'è ancora molto lavoro da fare sotto questo aspetto per ottenere i risultati desiderati, così come ci sono molte altre aree del progetto che necessitano di maggiori cure. Questo argomento verrà approfondito nel prossimo capitolo.

Capitolo 5 - Conclusioni

In quest'ultimo capitolo si analizzano e si discutono i risultati ottenuti dallo sviluppo di questo progetto.

Inizialmente, si considerano i successi raggiunti dal lavoro svolto e il suo contributo potenziale per il futuro dei giochi educativi. Successivamente, si esplorano possibili miglioramenti per gli aspetti già presenti in Project Coding, valutando le funzionalità necessarie per un eventuale progetto completo futuro.

5.1 Risultati Raggiunti

Dato che lo sviluppo del progetto è giunto al termine, risulta necessario analizzare i risultati raggiunti.

Inizialmente, uno dei primi argomenti affrontati per rendere possibile la creazione di un gioco educativo è stato l'individuazione del motivo del fallimento dei numerosi videogiochi educativi che hanno tentato di fornire nuove nozioni ai giocatori tramite il media videoludico. Alla fine delle numerose ricerche, il motivo di tali fallimenti è stato individuato nella loro scarsa componente ludica data da una generale mancanza di cura degli aspetti legati al Game Design. Per questo motivo, Project Coding, videogioco che ha come scopo quello di far avvicinare alla programmazione Java anche persone lontane da questo mondo, necessitava di una struttura che risultasse divertente e che non forzasse al suo interno i concetti di Java, bensì li amalgamasse alla sua struttura di gioco.

Si può dire che, per quanto l'impianto di gioco risulti ancora acerbo data la natura di prototipo del progetto, questo risultato sia stato, almeno teoricamente, raggiunto. Infatti, Project Coding si presenta come un titolo che verte sui concetti della programmazione Java, come classi, metodi e pacchetti, ma che incorpora tali nozioni in un ambiente ludico senza mai forzarli ma, al contrario, presentandoli al giocatore come elementi assolutamente centrali all'interno del gioco. Ovviamente, mutare la classica programmazione tramite codice in un'esperienza ludica che potesse risultare intuitiva anche a chi non è mai entrato in contatto con il mondo dell'informatica si è rivelata una sfida parecchio complessa, ma grazie alla trasformazione dei costrutti in tasselli di un puzzle e, soprattutto, grazie all'implementazione della descrizione del metodo, Project Coding è riuscito in questo ambizioso intento. Tali meccaniche, infatti, permettono al giocatore di prendere dimestichezza con questi elementi autonomamente, senza dover necessariamente studiare l'argomento. Il funzionamento della fase di apprendimento si basa sul fatto che un utente che inizialmente non ha la benché minima conoscenza del linguaggio Java, può semplicemente comprare gli oggetti che più attirano il suo

interesse nel negozio, anche solo per l'aspetto grafico, per poi provare ad utilizzarli all'interno della programmazione del metodo senza un criterio preciso. Tramite la continua sperimentazione, facendo uso della descrizione, la conoscenza del giocatore degli effetti degli elementi di gioco aumenterà di partita in partita fino a quando non sarà completamente in grado di programmare i propri attacchi senza più dover leggere gli effetti.

È un concetto che è già stato espresso nei capitoli precedenti, ma è fondamentale ribadirlo perché, per quanto prezioso al fine di raggiungere i risultati del progetto, è il fulcro di un problema che ha portato alla decisione di non effettuare alcun tentativo di testing.

Il problema del testing sta nel fatto che i frutti dell'apprendimento tramite Project Coding su un giocatore si manifestano dopo numerose partite: dapprima il giocatore non conosce nulla e fa affidamento sulla descrizione ma giocando prende manualità con i concetti base e non perde più tempo a creare combinazioni di tasselli per comprendere come creare il miglior metodo possibile. Di conseguenza, per effettuare un testing completo sarebbe stata necessaria la collaborazione di un gruppo di persone senza conoscenze legate al linguaggio Java per un periodo di tempo prolungato. Ciò però comporta che il prototipo di Project Coding potesse offrire un'esperienza longeva, formata da vari livelli e che presentasse una quantità di situazioni varie in modo da non annoiare mai il giocatore. Purtroppo, raggiungere tale risultato in tempi brevi risulta inimmaginabile dato che richiede lunghe tempistiche e numerose risorse. Attualmente, Project Coding è composto da un solo livello e, in generale, da una quantità di contenuti che non risulta sufficiente per verificare gli effetti dell'applied game tramite la prova diretta da parte di una persona. Nonostante la generazione casuale della mappa permetta di rendere le partite, anche dello stesso livello, molto diverse tra loro, non si considera questa funzione abbastanza impattante per mantenere l'esperienza di gioco divertente anche dopo numerose partite.

Un altro settore del videogioco trascurato a causa della mancanza di tempo è il comparto audio, fondamentale per rendere l'esperienza gradevole. Infatti, Project Coding al momento non presenta né effetti sonori, né musiche, il che impatta negativamente sull'esperienza di gioco.

5.2 Sviluppi futuri

Per quanto l'ossatura di Project Coding sia solida, il progetto è lontano da una versione definitiva. In questa sottosezione verranno approfondite tutte le migliorie che possono essere apportate ad aspetti già esistenti e tutte le possibili aggiunte per la buona riuscita di un applied game completo.

Questa prima versione di Project Coding è stata sviluppata non solo allo scopo di mostrare il potenziale degli applied game tramite una breve esperienza, ma anche

per creare l'ossatura del sistema in modo da semplificare lo sviluppo futuro. Di conseguenza, le tempistiche ridotte hanno lasciato spazio solo alla creazione di un singolo livello di gioco, purtroppo incompleto sotto molti aspetti data la sua natura di prototipo. Risulta quindi naturale comprendere che uno degli sviluppi futuri auspicabili sia la creazione di numerosi livelli. Ciò comporterebbe:

- la creazione di nuovi nemici, tutti contraddistinti da movimenti, animazioni, fattezze e mosse uniche in modo da non annoiare mai il giocatore;
- la creazione di nuovi scenari, sia per quanto riguarda le battaglie, sia per quanto riguardale mappe;
- nuovi pacchetti contenenti classi dai comportamenti variegati che permettano al giocatore di spaziare tra strategie completamente diverse tra loro;
- nuovi pezzi di codice, che amplino il ventaglio di possibilità di personalizzazione delle mosse del giocatore e, di conseguenza, le sue conoscenze dell'universo Java.

Inoltre, sarebbe interessante rendere più profondo il concetto di “*boss fight*” (la battaglia contro il boss di fine livello.) Al momento, tale evento prevede semplicemente uno scontro contro un nemico più forte dei precedenti, ma si potrebbe pensare di creare una selezione di nemici dalle mosse particolarmente curate che magari impattino anche su altri elementi del gioco come le arene delle battaglie, in modo da generare sfide ardue ma memorabili. Collegare tali scontri ad un eventuale componente narrativa innalzerebbe notevolmente il tasso di coinvolgimento del giocatore, rendendo quest'ultimo molto più predisposto ad impegnarsi e ad affrontare qualsiasi avversità.

E, a tal proposito, un elemento che si voleva inserire già all'interno del prototipo era la componente narrativa. Purtroppo, ciò non è stato possibile a causa delle limitate tempistiche a disposizione. Tuttavia, nel progetto completo, si riconosce l'importanza di una trama che guida il giocatore durante il suo percorso.

La componente narrativa può essere fondamentale per la buona riuscita del progetto e non è un caso che esistano numerosi articoli o addirittura interi libri che trattino l'argomento. Ciò non implica che un gioco debba obbligatoriamente avere una trama per avere valore; alla fine, ciò che conta di più è l'esperienza che il titolo offre al giocatore. Tuttavia, è essenziale riconoscere il notevole impatto motivazionale che una trama ben scritta può generare.

Non è lo scopo di questo capitolo approfondire ulteriormente l'argomento ma è necessario comprendere un concetto fondamentale: all'interno del contesto video-ludico, una componente narrativa di valore, sia essa attiva o passiva (ovvero, offre o meno al giocatore la possibilità di prendere decisioni che influenzino la trama), è

in grado di trasmettere al giocatore numerose emozioni, spingendolo ad agire [8]. Come sottolineato anche in precedenza, la totale assenza di un comparto audio depotenzia fortemente l'attuale esperienza di Project Coding. Gli effetti sonori riescono a rafforzare notevolmente la percezione del giocatore di ciò che avviene a schermo, risultando fondamentali per la buona riuscita del progetto.

Il ruolo di una buona colonna sonora all'interno di un videogioco è difficile da descrivere in poche righe data la sua importanza. Basti pensare che le cosiddette OST (Original Soundtrack) nel mondo videoludico sono al centro di numerosi articoli e libri. Di conseguenza, sarebbe necessario condurre attenti studi per sviluppare musiche idonee a coinvolgere il giocatore nel mondo di gioco di Project Coding.

Entrambi questi aspetti si configurano come elementi di rilevanza estrema per la buona riuscita del progetto, e nel corso di un eventuale sviluppo della versione completa, richiederebbero la massima attenzione e cura.

Al momento, Project Coding presenta molti elementi “placeholder” dal punto di vista grafico. Basti pensare che alcuni nemici sono semplicemente rappresentati da un rettangolo rosso. Risulta quindi palese che, uno dei compatti da migliorare ulteriormente, sia quello grafico, studiando dapprima dei design affascinanti per nemici, luoghi e UI per poi trasformarli in modelli 3D, sprite ed altri elementi grafici in grado di catturare e deliziare l'occhio del pubblico.

In conclusione, Project Coding aveva come obiettivo quello di mostrare come sia possibile creare un applied game votato all'istruzione (in questo caso, legato al mondo della programmazione Java) che possa essere divertente e coinvolgente, senza ricorrere a lunghe spiegazioni o a lezioni mascherate da esperienza ludica. Si può dire che sia riuscito nel suo intento, dato che la struttura base del gioco è già presente in questo primo prototipo, ma il lavoro è lontano dal suo completamento. Ci sono tanti aspetti da migliorare e completare. Tuttavia, si spera che questo lavoro possa ispirare altri sviluppatori con il sogno di avvicinare giovani menti a vari argomenti tramite l'utilizzo di un mezzo divertente e leggero, sfruttando il potenziale educativo, ancora inespresso, dei videogiochi.

Ringraziamenti

Desidero dedicare questo spazio della mia tesi per ringraziare tutti coloro che mi hanno sostenuto lungo questo percorso.

Innanzitutto, desidero ringraziare il Professore Palomba, relatore della mia tesi, per aver accettato di accompagnarmi nella realizzazione di questo progetto.

Un sentito ringraziamento va alla Dottore Giusy Annunziata e al Dottore Stefano Lambiase, co-relatori di questa tesi, per la loro costante disponibilità, consigli e correzioni che hanno contribuito alla realizzazione del progetto.

Voglio esprimere la mia profonda gratitudine ai miei familiari e in particolare a mia madre, che ha sempre creduto in me incondizionatamente e mi ha sostenuto in ogni fase della mia vita, compreso il percorso universitario.

Infine, ringrazio i miei amici e tutte le persone che ho incontrato durante questo percorso e che hanno contribuito a farmi crescere sia come individuo che come informatico.

Bibliografia

- [1] Robert Haworth e Kamran Sedig. “The importance of design for educational games”. In: *Education in a technological world* (2011), pp. 518–522.
- [2] Ariel Manzur Juan Linietsky e the Godot community. *Godot Documentation*. 2014. URL: <https://docs.godotengine.org/en/stable/#>.
- [3] Gabriela Kiryakova, Nadezhda Angelova e Lina Yordanova. “Gamification in education”. In: *Proceedings of 9th international Balkan education and science conference*. Vol. 1. 2014, pp. 679–684.
- [4] Teemu H Laine e Renny SN Lindberg. “Designing engaging games for education: A systematic literature review on game motivators and design principles”. In: *IEEE Transactions on Learning Technologies* 13.4 (2020), pp. 804–821.
- [5] Alessio Piro. *Project Coding Demo*. 2023. URL: <https://github.com/AlessioPiro/ProjectCodingDemo>.
- [6] Jan L Plass, Bruce D Homer e Charles K Kinzer. “Foundations of game-based learning”. In: *Educational psychologist* 50.4 (2015), pp. 258–283.
- [7] Jean-Nicolas Proulx, Margarida Romero e Sylvester Arnab. “Learning mechanics and game mechanics under the perspective of self-determination theory to foster motivation in digital game based learning”. In: *Simulation & Gaming* 48.1 (2017), pp. 81–97.
- [8] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC press, 2008.