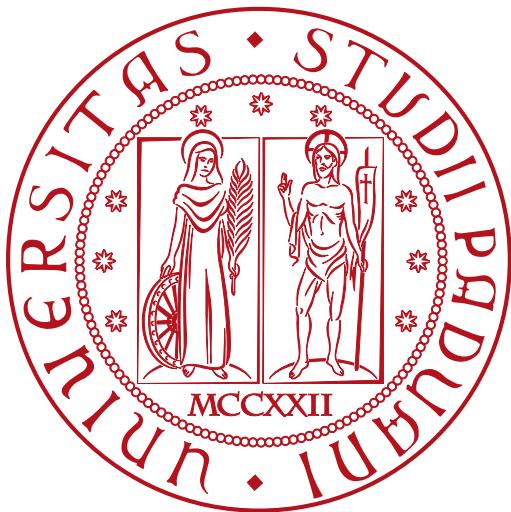


Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

CORSO DI LAUREA IN INFORMATICA



**Sviluppo di un progetto di Game Design
con elementi a tema di Machine Learning
e Intelligenza Artificiale**

Tesi di Laurea

Relatore

Prof. Vardanega

Laureando

Filippo Sabbadin

Matricola 2010008

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Vardanega, relatore della mia tesi, per il grande aiuto ed il continuo supporto fornитоми durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori e i miei parenti per il loro sostegno e per essermi stati vicini durante gli anni di studio, continuando ad incoraggiarmi nell'affrontare le sfide ed andare avanti.

Desidero ringraziare poi i miei amici, per avermi sostenuto durante tutto il percorso universitario ed avermi aiutato anche nei momenti più difficili, rimanendo sempre al mio fianco.

Ci tengo infine a ringraziare i colleghi di Zucchetti S.p.A. e il tutor aziendale Gregorio Piccoli per avermi dato l'opportunità di lavorare ad un progetto di mio personale interesse, permettendomi di crescere professionalmente e di mettere in pratica le mie competenze.

Padova, Settembre 2025

Filippo Sabbadin

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di *stage* curricolare, della durata di circa trecento ore, dal laureando Filippo Sabbadin presso l'azienda Zucchetti S.p.A.. Lo *stage* è stato condotto sotto la supervisione del tutor aziendale Gregorio Piccoli, mentre il prof. Prof. Vardanega ha ricoperto il ruolo di tutor accademico.

Organizzazione del testo

Il primo capitolo presenta l'azienda ospitante, illustrando il contesto organizzativo e produttivo in cui si è svolto lo *stage*, i processi interni adottati e la tipologia di clientela a cui si rivolge. Vengono inoltre descritte le principali tecnologie di supporto utilizzate dal personale e la propensione dell'azienda all'innovazione.

Il secondo capitolo approfondisce il rapporto dell'azienda con gli *stage*, l'interesse verso il progetto svolto e le motivazioni della scelta. Vengono illustrati gli obiettivi e i vincoli concordati con il tutor aziendale, la pianificazione e il calendario delle attività, l'organizzazione del lavoro e le principali tecnologie utilizzate. Infine, si analizza come il progetto si inserisce nel contesto di innovazione e mercato dell'azienda.

Il terzo capitolo descrive in dettaglio l'analisi dei requisiti con i casi d'uso e la lista dei requisiti, l'architettura, *test* e validazione dello *stage*. Seguirà, infine, una descrizione dei risultati che ho raggiunto sul piano qualitativo e quantitativo.

Il quarto capitolo per finire, descrive l'esperienza personale di *stage*, quanti obiettivi soddisfatti rispetto agli obiettivi totali dichiarati nel secondo capitolo, la maturazione durante lo *stage*, con conoscenze ed abilità acquisite durante il periodo.

Infine, verrà fatto un confronto tra le competenze richieste a inizio *stage* rispetto a quelle erogate dal corso di studi, con eventuali lacune su quest'ultimo su competenze che sarebbero state utili per lo *stage* o mondo del lavoro.

Convenzioni tipografiche

Durante la stesura del testo ho scelto di adottare le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini di uso non comune menzionati vengono definiti nel glossario, situato alla fine del documento (p. 55);

- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *termine_G* e ne viene riportata una breve descrizione del termine a piè di pagina;
- i termini in lingua straniera non di uso comune o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*;
- all'inizio di ogni capitolo viene riportato un breve sommario sugli argomenti principali che il capitolo tratta;
- i nomi di funzioni, variabili o classi appartenenti ad un linguaggio di programmazione vengono scritte con un carattere **monospaziato**;
- le citazioni ad un libro o ad una risorsa presente nella bibliografia (p. 60) saranno affiancate dal rispettivo numero identificativo, es. [1];
- ogni immagine sarà accompagnata da un titolo e verrà elencata nel suo indice apposito a inizio documento, l'immagine 1 mostra un esempio:

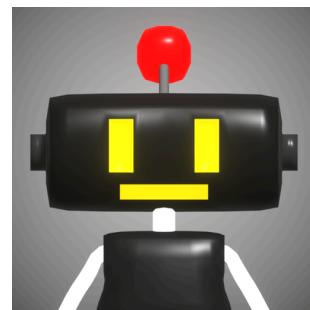


Figura 1: Immagine esempio

- allo stesso modo, ogni tabella sarà seguita da un suo titolo ed inserita nel suo indice apposito. Inoltre ogni riga avrà un colore diverso dalle righe vicine per renderla più accessibile, la tabella 1 mostra un esempio:

Titolo 1	Titolo 2
Valore 1-1	Valore 1-2
Valore 2-1	Valore 2-2

Tabella 1: Tabella esempio

- i blocchi di codice sono rappresentati come il blocco di Codice 1:

```
1 float Q_rsqrt( float number ){
2     long i;
3     float x2, y;
4     const float threehalfs = 1.5F;
5     x2 = number * 0.5F;
6     y = number;
7     i = * (long * ) &y;
8     i = 0x5f3759df - (i>>1);
9     y = * (float * ) &i;
10    y = y * ( threehalfs - ( x2 * y * y ) );
11    return y;
12 }
```

C C

Codice 1: Codice d'esempio.

Indice

1 L’azienda	1.
1.1 Introduzione	1.
1.1.1 Informazioni sull’azienda	1.
1.1.2 Sede dello <i>stage</i>	1.
1.2 Contesto organizzativo e produttivo	2.
1.3 Processi interni utilizzati	3.
1.3.1 Sviluppo, manutenzione ed organizzazione del lavoro	3.
1.3.2 Tecnologie di supporto	3.
1.4 Clientela e prodotti	4.
1.4.1 Tipologia di clientela	4.
1.4.2 Prodotti e servizi	5.
1.5 Propensione per l’innovazione	5.
2 Lo <i>stage</i>	7.
2.1 Rapporto dell’azienda con gli <i>stage</i>	7.
2.2 Interesse personale e dell’azienda verso lo <i>stage</i>	7.
2.2.1 Proposta del progetto	7.
2.2.2 Motivazioni personali	7.
2.2.3 Scelta dell’azienda	8.
2.2.4 Supporto dell’azienda verso il progetto	8.
2.3 Descrizione del progetto	9.
2.3.1 Struttura del gioco	9.
2.3.2 Rapporto del progetto con l’innovazione	10.
2.3.3 Aspettative	11.
2.4 Obiettivi	11.
2.5 Vincoli	12.
2.5.1 Vincoli temporali e tecnologici	12.
2.5.2 Pianificazione	12.
2.5.3 Calendario	13.
2.5.4 Organizzazione del lavoro	14.
2.5.5 Tecnologie scelte	17.
2.5.6 Analisi dei rischi	20.
3 Il progetto	25.
3.1 Analisi dei requisiti	25.
3.1.1 Attori	25.

3.1.2	Casi d'uso	25.
3.1.3.	Requisiti	30.
3.2.	Architettura	31.
3.2.1.	Concetti chiave di <i>Godot</i>	31.
3.2.2.	Funzioni comuni	33.
3.2.3.	Classi del giocatore	34.
3.2.4.	Entità <i>interagibili</i>	37.
3.2.5.	Struttura base di un livello	38.
3.2.6.	<i>LRCannon</i>	38.
3.2.7.	<i>LinearRegressionGraph</i>	39.
3.2.8.	Livello Albero di decisione	41.
3.2.9.	Livello <i>Causalità</i>	43.
3.3.	Verifica e validazione	44.
3.3.1.	Nomenclatura <i>test</i>	44.
3.4.	Risultati ottenuti	45.
3.4.1.	Meccaniche dei livelli	45.
3.4.2.	Requisiti soddisfatti	46.
3.4.3.	<i>Test</i> superati	46.
3.4.4.	Quantità di prodotti	47.
4	Conclusioni	49.
4.1	Obiettivi stage soddisfatti	49.
4.1.1	Grado di soddisfazione degli obiettivi	49.
4.1.2	Obiettivi non superati	50.
4.2	Esperienze acquisite	52.
4.2.1	Competenze tecniche	52.
4.2.2	Competenze trasversali	52.
4.2.3	Gestione dei rischi	52.
4.3	Differenza tra <i>stage</i> e percorso studi	54.
4.4	Pensieri finali	54.
Glossario	55.	
Bibliografia	60.	

Elenco delle Figure

Figura 1	Immagine esempio	vi
Figura 2	Settori e temi di cui si occupa l'azienda. Fonte: Zucchetti	4.
Figura 3	Livello <i>Regressione lineare</i>	9.
Figura 4	Livello <i>Albero di decisione</i>	10.
Figura 5	Livello <i>Causalità</i>	10.
Figura 6	Attore principale	25.
Figura 7	Diagramma <i>UML use case</i> sul movimento	26.
Figura 8	Diagramma <i>UML</i> sull'interazione automatica con un' <i>entità</i>	26.
Figura 9	Diagramma <i>UML</i> sull'interazione manuale con un' <i>entità</i>	27.
Figura 10	Diagramma <i>UML</i> sul lasciare un oggetto	28.
Figura 11	Diagramma <i>UML</i> sull'interazione con una macchina <i>LR</i>	28.
Figura 12	Diagramma <i>UML</i> sull'accensione di un'unità esterna di un condizionatore	29.
Figura 13	<i>Scena</i> del personaggio del giocatore	32.
Figura 14	Diagramma <i>UML</i> delle classi del giocatore	34.
Figura 15	Diagramma <i>UML</i> sulla struttura della macchina di stati	35.
Figura 16	Diagramma sul flusso degli stati del giocatore	36.
Figura 17	Diagramma <i>UML</i> degli oggetti con cui il giocatore può interagire	37.
Figura 18	Diagramma delle classi di un livello base	38.
Figura 19	Diagramma sul funzionamento di un grafico <i>Linear Regression</i> nel gioco	38.
Figura 20	Diagramma sul funzionamento dell'Albero di decisione	41.
Figura 21	Diagramma del livello <i>Causalità</i>	43.
Figura 22	Immagine del cannone <i>LR</i> per posizionare nuovi punti nel grafico della Regressione lineare	45.
Figura 23	Immagine della scena di intermezzo del livello <i>Causalità</i>	45.
Figura 24	Grafico del tempo di compilazione di ogni <i>frame</i> del gioco	50.

Elenco delle Tabelle

Tabella 1	Tabella esempio	vi
Tabella 2	Obiettivi del progetto	11.
Tabella 3	Vincoli del progetto	12.
Tabella 4	Pianificazione del lavoro in ore	13.
Tabella 5	tabella dei documenti	15.
Tabella 6	Linguaggi di programmazione scelti	17.
Tabella 7	<i>Softwares</i> scelti	18.
Tabella 8	Strumenti e servizi utilizzati	18.
Tabella 9	Tipi di <i>file</i> scelti	19.
Tabella 10	Errata pianificazione dei tempi	20.
Tabella 11	Impegni personali o universitari	21.
Tabella 12	Mancanza di competenze tecniche	21.
Tabella 13	Tecnologie non adeguate	21.
Tabella 14	Cambio dei requisiti	22.
Tabella 15	Errore nella progettazione dell'architettura	23.
Tabella 16	Totale requisiti	30.
Tabella 17	Totale <i>test</i> eseguiti	44.
Tabella 18	Tabella con grado di superamento requisiti	46.
Tabella 19	Tabella con grado di superamento <i>test</i> eseguiti	46.
Tabella 20	Tabella delle applicazioni prodotte nello <i>stage</i>	47.
Tabella 21	Obiettivi del progetto	49.

Elenco dei Codici Sorgente

Codice 1 Codice d'esempio.	vii
Codice 2 Funzione <code>calculate_a_b()</code>	39.

Capitolo 1

L’azienda

In questo capitolo descrivo l’azienda, il contesto organizzativo in cui sono stato inserito, i processi interni utilizzati e la tipologia di clientela a cui si rivolge.

1.1 Introduzione

1.1.1 Informazioni sull’azienda

L’azienda Zucchetti Spa opera nel settore informatico da oltre 45 anni ed offre una vasta gamma di soluzioni *software* e servizi per le aziende, mantenendosi sempre aggiornati su tematiche come il diritto civile, contabilità, fiscalità, diritto del lavoro e previdenza.

Zucchetti ha un organico di circa 9.000 persone, con oltre 2.000 di queste dedicate a ricerca e sviluppo, dimostrando una forte attenzione all’innovazione tecnologica e al miglioramento continuo dei propri prodotti.

L’azienda investe costantemente in nuove tecnologie e nella formazione del personale, favorendo un ambiente dinamico e orientato alla crescita professionale.

1.1.2 Sede dello *stage*

La sede dell’azienda dove ho svolto lo *stage* si trova a Padova, in un’area ben collegata e facilmente raggiungibile.

L’ufficio è situato in un edificio moderno e funzionale, dotato di spazi di lavoro aperti e aree dedicate alla collaborazione tra i membri del *team*.

L’ambiente di lavoro si è rivelato collaborativo e stimolante, con una particolare attenzione alla condivisione delle conoscenze e al supporto reciproco tra colleghi. Ho potuto osservare da vicino l’organizzazione del lavoro e l’interazione tra il personale.

Questa esperienza mi ha permesso di comprendere meglio le dinamiche aziendali e di apprezzare l’importanza dell’innovazione continua all’interno dell’azienda.

1.2 Contesto organizzativo e produttivo

Durante lo *stage* sono stato per lo più indipendente, tuttavia qualora avessi avuto bisogno di aiuto, potevo chiedere ad un gruppo composto da professionisti con competenze eterogenee, tra cui sviluppatori, analisti e *project manager*. Ho potuto osservare come la collaborazione e il confronto tra colleghi fossero elementi fondamentali per il buon andamento dei progetti.

L'ambiente lavorativo era caratterizzato da una forte attenzione alla qualità del prodotto e al rispetto delle scadenze, con un approccio orientato al miglioramento continuo dei processi produttivi.

Questo mi ha aiutato a comprendere l'importanza di lavorare in un contesto organizzato e strutturato, dove ogni componente contribuisce al raggiungimento degli obiettivi comuni.

1.3 Processi interni utilizzati

1.3.1 Sviluppo, manutenzione ed organizzazione del lavoro

Durante tutto il periodo di *stage*, ho svolto le mie attività seguendo i processi interni decisi dall’azienda, che prevedevano una gestione strutturata del progetto e una chiara suddivisione dei compiti da eseguire. I processi interni dell’azienda comprendevano fasi distinte per lo sviluppo, la manutenzione e l’organizzazione del lavoro.

- Durante l’organizzazione del lavoro, ho potuto notare una forte comunicazione tra il personale in ufficio ed anche con i membri che lavoravano in *smart working* durante alcuni giorni.
- Per lo sviluppo, ognuno aveva compiti specifici e responsabilità ben definite, con delle eventuali piccole discussioni per chiedere chiarimenti o approfondimenti, ad esempio, sul codice. Molto spesso vedeva due o più membri lavorare insieme su uno stesso argomento, scambiandosi idee e suggerimenti per migliorare il prodotto finale oppure per risolvere eventuali problemi.
- Tutto ciò che veniva sviluppato e completato, era anche mantenuto in base alle esigenze del cliente, in caso di problemi o richieste particolari, come problemi di compatibilità con versioni più vecchie dei *browser*.

1.3.2 Tecnologie di supporto

Per facilitare la comunicazione e la collaborazione tra i membri del *team*, l’azienda ha adottato diverse tecnologie di supporto. La comunicazione tra i membri del *team* avveniva principalmente tramite strumenti digitali di collaborazione, ad esempio *Microsoft Teams*_G, che facilitavano la condivisione delle informazioni e il coordinamento delle attività. Attraverso questa piattaforma, i membri svolgevano anche riunioni o videochiamate, permettendo una comunicazione rapida ed efficace, anche a distanza.

Inoltre, per sincronizzare i cambiamenti e garantire che tutti i membri del *team* fossero aggiornati, venivano utilizzati sistemi di versionamento e *repository* condivisi, in un *database*_G interno. Questi strumenti hanno reso possibile una gestione efficiente delle attività e una rapida risoluzione dei problemi, contribuendo a mantenere un flusso di lavoro fluido e produttivo.

database: insieme organizzato di dati, generalmente memorizzato e gestito in modo da facilitarne l’accesso e la manipolazione

Microsoft Teams: piattaforma di comunicazione e collaborazione sviluppata da Microsoft.

1.4 Clientela e prodotti

1.4.1 Tipologia di clientela

Zucchetti si rivolge a una clientela molto diversificata, che comprende sia piccole e medie imprese, sia grandi aziende, tutte queste private.

Questa varietà di clientela rappresenta uno stimolo costante all'innovazione e all'adattamento dei prodotti alle evoluzioni del mercato, contribuendo a mantenere un'offerta sempre aggiornata e competitiva.



market@zucchetti.it - www.zucchetti.it



Figura 2: Settori e temi di cui si occupa l'azienda. Fonte: Zucchetti

Ho potuto osservare una discussione molto aperta e libera tra i membri del team riguardo alle esigenze e alle aspettative dei clienti, modificando i prodotti in base alle richieste e necessità di questi ultimi.

1.4.2 Prodotti e servizi

L'azienda offre una vasta gamma di prodotti con funzionalità diverse.

Alcuni esempi sono:

- soluzioni gestionali per la contabilità;
- la gestione del personale;
- la gestione della produzione;
- *software* verticali progettati per rispondere alle esigenze di settori specifici come:
 - ▶ sanità;
 - ▶ *retail*;
 - ▶ logistica;
 - ▶ produzione industriale.

Questi prodotti sono pensati per supportare le aziende nell'ottimizzazione dei processi, nella gestione efficiente delle risorse e nell'adeguamento alle normative vigenti.

Oltre ai *software*, Zucchetti fornisce anche servizi, che possono essere di:

- consulenza;
- assistenza tecnica;
- formazione.

In questo modo, Zucchetti garantisce un supporto completo ai propri clienti durante tutte le fasi di adozione e utilizzo delle soluzioni offerte.

1.5 Propensione per l'innovazione

Zucchetti investe costantemente in ricerca e sviluppo, con oltre 2.000 persone dedicate a queste attività. Questo impegno si traduce in una continua introduzione di nuove tecnologie, metodologie di lavoro innovative e aggiornamenti dei prodotti offerti. L'azienda promuove attivamente la formazione del personale e la sperimentazione di soluzioni all'avanguardia, favorendo un ambiente in cui l'innovazione è parte integrante della cultura aziendale.

Durante il mio *stage*, ho potuto osservare come le idee innovative vengano accolte con interesse e valutate attentamente, sia a livello di processo che di prodotto. Ho avuto modo, inoltre, di assistere ad una sessione di *brainstorming* dimostrazione di come l'azienda sia aperta a nuove idee e approcci.

Argomento principale delle ricerche che il personale dell'azienda stava svolgendo, erano gli *LLM*, tema ancora molto nuovo ed inesplorato nel mondo dell'informatica.

brainstorming: tecnica di generazione di idee in gruppo, in cui i partecipanti sono incoraggiati a esprimere liberamente le proprie idee.

LLM - Large Language Model: modello di intelligenza artificiale progettato per comprendere e generare testo in linguaggio naturale.

Per la maggior parte, il personale in azienda si occupava di test e addestramento dei vari modelli, cambio dei parametri, ad esempio, la *temperatura_G*, analizzando gli *output* che questi generavano, la correttezza di questi, e molto altro.

Questa collaborazione contribuisce a generare nuove soluzioni e a mantenere elevato il livello di competitività sul mercato.

La propensione all'innovazione dell'azienda si riflette nella rapidità con cui adotta strumenti digitali e tecnologie emergenti, garantendo così un costante miglioramento dei servizi e delle soluzioni offerte ai clienti.

temperatura: parametro che controlla la casualità delle risposte generate da un LLM.

Capitolo 2

Lo *stage*

In questo capitolo approfondisco il rapporto con l'azienda ospitante verso gli stage in generale, come ha supportato il mio stage, il perché della mia scelta e gli obiettivi e vincoli decisi con il tutor aziendale. Infine verrà messo a confronto il tema dello stage con l'innovazione ed il mercato dove viene inserito il progetto.

2.1 Rapporto dell'azienda con gli *stage*

Da molti anni l'azienda Zucchetti Spa si presenta all'evento *StageIT_G*, incontrando un vasto numero di studenti, proponendo loro nuovi temi da approfondire per i progetti di *stage*.

Oltre a questi progetti, l'azienda è sempre disponibile a valutare nuove idee di progetti proposti dagli studenti, ascoltando le loro esigenze e suggerimenti.

Ho potuto osservare questa disponibilità anche durante lo *stage*, dove erano presenti dei miei colleghi, con alcuni di loro che stavano svolgendo progetti che non erano stati proposti dall'azienda, ma decisi da loro, segno che l'azienda è sempre aperta a nuove idee e proposte.

2.2 Interesse personale e dell'azienda verso lo *stage*

2.2.1 Proposta del progetto

Sempre durante l'evento *StageIT* 2025, ho avuto la possibilità di incontrare il *tutor* aziendale, il quale, prima di presentare i progetti proposti dall'azienda, mi ha dato la possibilità di proporre un tema di progetto diverso.

La mia proposta è stata un progetto di *Game Design_G*.

2.2.2 Motivazioni personali

Il tema di *Game Design* è un argomento che mi appassiona da tempo e su cui ho già avuto modo di lavorare in progetti personali, seppure di piccole dimensioni. Inoltre, il *Game Design* offre l'opportunità di esplorare la creatività e di sviluppare nuove idee in un contesto ludico.

Game Design: disciplina che si occupa della progettazione e dello sviluppo di giochi.

StageIT: evento orientato al lavoro, dedicato agli studenti per aiutarli a trovare aziende dove svolgere l'attività di *stage*.

Questo tema mi ha permesso di combinare la mia passione per lo sviluppo di applicazioni videoludiche con l'apprendimento di nuove competenze tecniche e di sviluppo, rendendo l'esperienza di *stage* più coinvolgente e stimolante.

2.2.3 Scelta dell'azienda

Il motivo principale della mia scelta sull'azienda è stata la disponibilità aperta ad esplorare nuove idee ed approcci.

Ulteriore motivo della mia scelta è stata la posizione della sede dove ho svolto lo *stage*, che ho trovato molto comoda da raggiungere, attraverso i mezzi di trasporto disponibili.

2.2.4 Supporto dell'azienda verso il progetto

Nonostante ritenessi che la mia proposta avesse poco valore, il *tutor* ha mostrato un interesse genuino e ha incoraggiato la mia idea, portandomi a svilupparla ulteriormente.

Successivo all'evento *StageIT*, ho avuto la possibilità di parlare con il *tutor* aziendale, attraverso una videochiamata, il quale mi ha supportato nella definizione del progetto, aiutandomi a capire come svilupparlo al meglio e mostrandomi anche esempi di progetti passati svolti sullo stesso argomento.

Visto che l'argomento del momento nel mondo dell'informatica era l'uso degli *LLM*, il *tutor* ha proposto di usare gli argomenti comuni sul tema dell'Intelligenza Artificiale e **Machine Learning (ML)_G** per arricchire il progetto, ed usarli per creare nuove meccaniche nel gioco per creare livelli unici, che il giocatore deve completare. In questo modo, tramite il gioco, l'utente impara nuovi argomenti riguardo al mondo dell'Intelligenza artificiale e *Machine Learning* ed apprende il loro funzionamento.

Tra questi temi, sono spiccati di più:

- *Regressione lineare_G* ;
- *Alberi di decisione_G* ;
- causalità;
- *Nearest Neighbor_G* ;
- *Support Vector Machines_G*.

alberi di decisione: modello predittivo che rappresenta le decisioni e le loro possibili conseguenze sotto forma di un albero.

Machine Learning: disciplina che si occupa dello sviluppo di algoritmi che permettono ai computer di apprendere dai dati e migliorare le proprie prestazioni nel tempo senza essere esplicitamente programmati.

Nearest neighbor: algoritmo di classificazione che assegna un'etichetta a un campione in base alle etichette dei suoi vicini più prossimi nel dataset.

Regressione lineare: tecnica statistica utilizzata per modellare la relazione tra una variabile dipendente e una o più variabili indipendenti, assumendo una relazione lineare.

Support Vector Machines: classe di algoritmi di apprendimento che cercano di trovare l'iperpiano ottimale che separa le classi nel dataset.

2.3 Descrizione del progetto

2.3.1 Struttura del gioco

Il progetto si tratta di un videogioco educativo, che ha come obiettivo quello di insegnare i concetti base dell'Intelligenza Artificiale e *Machine Learning* in modo semplice e divertente.

Oltre al livello *tutorial* ed al livello principale, dove il giocatore può scegliere il livello che vuole affrontare, sono presenti 3 livelli, ognuno dei quali insegna un concetto diverso:

- il livello della **regressione lineare**, dove la linea nel grafico diventa un ponte su cui il personaggio può camminare, tuttavia la direzione non è corretta e bisogna modificarla aggiungendo nuovi punti nel grafico, modificando la direzione della linea;

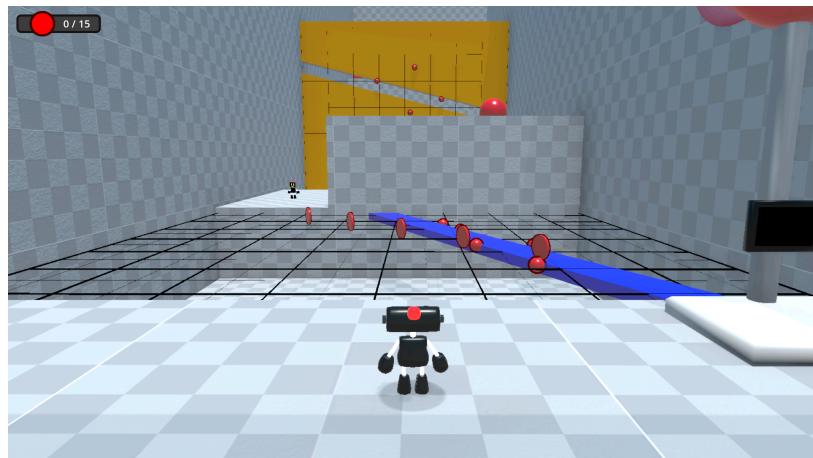


Figura 3: Livello *Regressione lineare*

tutorial: guida passo-passo che insegna come utilizzare un software o completare un'attività specifica.

- il livello dell'**albero di decisione**, dove l'utente dovrà classificare diverse razze di cani in base ai valori, già assegnati, in un albero di decisione;



Figura 4: Livello *Albero di decisione*

- il livello della **causalità**, dove l'obiettivo principale è cercare la giusta causa di quello che sta avvenendo nel livello, ed in caso di risposta corretta, l'utente viene avvisato e premiato.



Figura 5: Livello *Causalità*

2.3.2 Rapporto del progetto con l'innovazione

Il progetto si inserisce in un mercato in crescita, dove l'innovazione rappresenta un elemento chiave per attrarre e coinvolgere nuovi utenti.

La possibile clientela *target* del progetto è costituita principalmente da giovani e appassionati di tecnologia e videogiochi, interessati sia all'aspetto ludico sia a quello educativo. Il prodotto si rivolge a chi desidera apprendere concetti di Intelligenza Artificiale e *Machine Learning* in modo interattivo e coinvolgente, offrendo un'esperienza che unisce apprendimento e divertimento.

2.3.3 Aspettative

Prima di iniziare lo *stage* avevo delle aspettative riguardo al progetto, che si sono rivelate, alla fine, corrette.

Tra queste aspettative, vi erano:

- imparare nuove tecnologie e strumenti per lo sviluppo di giochi;
- migliorare le mie capacità di programmazione e *problem solving*;
- acquisire esperienza pratica nel lavoro di squadra e nella gestione di progetti;
- ricevere *feedback* costruttivo sul mio lavoro e sulle mie idee.

2.4 Obiettivi

Nella tabella 2, ho elencato gli obiettivi personali fissati, insieme al *tutor* aziendale, suddivisi in obbligatori e desiderabili.

Gli obiettivi sono elencati con un codice, costituito da un prefisso e un numero. Il prefisso indica con:

- **O**: gli obiettivi obbligatori, rappresentano le competenze e i risultati minimi da raggiungere durante lo *stage*;
- **D**: gli obiettivi desiderabili, sono traguardi aggiuntivi che arricchiscono ulteriormente il progetto e l'esperienza formativa.

Obbligatori	
O-1	Progettazione e realizzazione delle principali meccaniche di gioco (movimento 3D, interazione con altri oggetti...)
O-2	Implementazione degli argomenti di <i>AI</i> e <i>ML</i> al gioco (Regressione lineare, Alberi di decisione, Causalità...)
O-3	Implementazione di salvataggi e caricamenti dei dati attraverso file di tipo <i>.json</i> oppure <i>.ini</i>
Desiderabili	
D-1	Supporto della lingua inglese oltre all'italiano, con opzione di cambiare lingua di gioco
D-2	Implementazione di <i>shaders</i> , utilizzando <i>script</i> di tipo <i>OpenGLG</i>
D-3	Uso dei linguaggi <i>C#</i> o <i>C++</i> per migliorare le prestazioni
D-4	Implementazione di un modello di <i>LLM</i> per conversazioni tra personaggi all'interno del gioco

Tabella 2: Obiettivi del progetto

OpenGL: linguaggio di programmazione grafica utilizzato per creare applicazioni 3D e 2D.

shader: modello di sviluppo software che promuove la collaborazione e la condivisione del codice sorgente.

2.5 Vincoli

2.5.1 Vincoli temporali e tecnologici

Durante il secondo colloquio con il *tutor* aziendale, abbiamo definito dei vincoli obbligatori del progetto da rispettare.

Ho indicato nella tabella 3 i vincoli, con il prefisso:

- **VTM**: vincoli temporali;
- **VTC**: vincoli tecnologici;

Temporali	
VTM-1	Il progetto deve essere realizzato in un tempo massimo di 320 ore
VTM-2	Il numero di ore settimanali non può essere superiore a 40
Tecnologici	
VTC-1	Il progetto deve essere sviluppato utilizzando il <i>motore di gioco_G</i> <i>open source_G</i> <i>Godot</i>
VTC-2	L'applicazione finale deve essere un eseguibile, senza bisogno di installazione o altri strumenti
VTC-3	Il gioco deve utilizzare una grafica 3D con un movimento del personaggio in terza persona, tridimensionale

Tabella 3: Vincoli del progetto

2.5.2 Pianificazione

Il gioco contiene 3 livelli, per ognuno di questi livelli sono state dedicate due settimane.

Nella tabella 4 ho riportato le ore totali pianificate del progetto:

open source: modello di sviluppo software che promuove la collaborazione e la condivisione del codice sorgente.

motore di gioco: software progettato per facilitare lo sviluppo di videogiochi, fornendo strumenti e funzionalità per la gestione della grafica, della fisica, dell'audio e di altre componenti del gioco.

Durata (ore)	Descrizione attività
24	Pianificazione struttura applicazione Pianificazione stutura livelli Pianificazione implementazione meccaniche di gioco
63	Stesura della documentazione
24	Stesura documentazione relativa ad analisi e progettazione
16	Stesura delle metriche di qualità
15	Stesura delle norme e piano di progetto
8	Stesura del Manuale Utente
177	Sviluppo dei livelli
59	Sviluppo primo livello
59	Sviluppo secondo livello
59	Sviluppo terzo livello
40	<i>Test e verifica dell'applicazione</i>
Totale ore: 304	

Tabella 4: Pianificazione del lavoro in ore

2.5.3 Calendario

Il periodo di *stage* è suddiviso in 8 periodi, la cui lunghezza di ognuno corrisponde a una settimana. Qui sotto ho elencato le attività pianificate per ogni settimana:

- **Settimana 1 | 18/06 - 20/06 | 24 ore:**
 - Incontro con il personale dell'azienda per discutere i requisiti dell'applicazione da sviluppare.
 - Verifica credenziali e strumenti di lavoro assegnati
 - Pianificazione e progettazione dell'applicazione.
 - Inizio sviluppo.
- **Settimana 2 | 23/06 - 27/06 | 40 ore:**
 - Approfondimento sul tema *Regressione lineare*.
 - Sviluppo degli oggetti principali del primo livello, implementando gli elementi della *Regressione lineare*.

- **Settimana 3 | 30/06 - 04/07 | 40 ore:**
 - Approfondimento sul tema *Alberi di decisione*.
 - Sviluppo degli oggetti principali del secondo livello.
- **Settimana 4 | 07/07 - 11/07 | 40 ore:**
 - Approfondimento sull'argomento *Causalità*.
 - Sviluppo degli oggetti principali del terzo livello.
 - Compilazione del *PoC_G*.
- **Settimana 5 | 14/07 - 18/07 | 40 ore:**
 - Sviluppo del primo livello, sul tema *Regressione Lineare*, con gli oggetti creati nella seconda settimana.
- **Settimana 6 | 21/07 - 25/07 | 40 ore:**
 - Sviluppo del secondo livello, sul tema *Alberi di decisione*, implementando gli oggetti creati nella terza settimana.
- **Settimana 7 | 28/07 - 01/08 | 40 ore:**
 - Sviluppo del terzo livello, sul tema *Causalità*, implementando gli oggetti creati nella quarta settimana.
- **Settimana 8 | 04/08 - 08/08 | 40 ore:**
 - Stesura dei *test*.
 - Compilazione dell'*MVP_G*.

2.5.4 Organizzazione del lavoro

- **Organizzazione generale:**

Ogni giorno, sceglievo le attività da svolgere in base allo stato di avanzamento del progetto, tenendo conto delle priorità e delle scadenze.

Utilizzavo strumenti digitali per la gestione delle attività, come sistemi di versionamento o uso di *ticket_G*, per aiutarmi a tenere traccia dei compiti svolti. Ogni fase del lavoro, dalla raccolta dei requisiti allo sviluppo e alla verifica, veniva documentata e tracciata per garantire trasparenza ed efficienza.

Questi processi interni mi hanno permesso di acquisire una maggiore consapevolezza sull'importanza dell'organizzazione e della comunicazione all'interno di un contesto lavorativo strutturato.

Durante lo sviluppo, sono state imposte le seguenti regole per garantire corretta organizzazione. Le regole sono suddivise in base all'attività.

- **Documentazione:**

Dovevo documentare in modo dettagliato il processo di fornitura, in modo da garantire la tracciabilità delle attività svolte e delle decisioni che prendevo.

I documenti che dovevo scrivere erano:

PoC: descrive una dimostrazione pratica che ha lo scopo di verificare la fattibilità o il potenziale di un'idea, concetto o soluzione.

MVP: descrive la versione minima di un prodotto che include solo le funzionalità essenziali per essere utilizzato dagli utenti.

ticket: segnalazione o richiesta registrata in un sistema di tracciamento che descrive un'attività da svolgere.

Nome	Descrizione
Analisi dei requisiti	Definisce tutti gli i casi d'uso e i requisiti funzionali del progetto. Questi sono stati raccolti in collaborazione con il tutor aziendale e sono stati utilizzati come base per la progettazione e lo sviluppo del <i>software</i>
Glossario	Contiene la definizione dei termini utilizzati nel progetto
Piano di progetto	Definisce le attività da svolgere e i tempi previsti per lo sviluppo del <i>software</i> , viene descritto in dettaglio ogni periodo di sviluppo, con una retrospettiva delle attività svolte e una pianificazione delle attività future
Norme di progetto	Definisce le regole e le convenzioni da seguire durante lo sviluppo del <i>software</i> , come la nomenclatura dei <i>file</i> , la struttura del codice e le pratiche di programmazione da seguire
Piano di qualifica	Definisce le metriche che vengono usate per garantire la qualità del prodotto <i>software</i> . Vengono inoltre scritte le modalità di <i>test</i> e verifica del <i>software</i> , in modo da garantire che il prodotto soddisfi i requisiti stabiliti
Specifica tecnica	Describe in dettaglio l'architettura del sistema, i componenti <i>software</i> e le loro interazioni
Manuale utente	Fornisce istruzioni dettagliate su come utilizzare il <i>software</i> all'utente e garantirne il corretto funzionamento

Tabella 5: tabella dei documenti

- **Codifica:**

Tutti i *file* contenenti codice del gioco sono salvati come file *.gd*, e sono scritti con il linguaggio di programmazione *GDScript*_G. Ho salvato i nomi delle classi con una nomenclatura *PascalCase*_G, mentre i nomi dei *file* e delle variabili usano *snake_case*_G.

GDScript: linguaggio di programmazione specifico per il motore di gioco *Godot*, progettato per essere semplice e intuitivo.

PascalCase: pratica di scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole.

snake_case: pratica di scrivere parole composte separando le parole tramite trattino basso, con tutte le lettere minuscole.

Per maggiori dettagli sulla nomenclatura, ho seguito le convenzioni della documentazione ufficiale:

https://docs.godotengine.org/it/4.x/tutorials/scripting/gdscript/gdscript_styleguide.html

- **Modellazione:**

Ho esportati tutti i modelli nel formato **.glb_G**. Esportavo il **materiale_G** insieme al modello 3D senza immagini, come un *placeholder*, dato che veniva rimpiazzato dal materiale presente nei *file* del gioco.

Nel caso il modello 3D presentava animazioni, queste venivano esportate insieme al modello.

- **Animazione:**

Le animazioni sono incluse nel modello durante l'esportazione. Per semplificare l'attività, ho usato un **rig_G** che dispone di **IK_G**. Le animazioni sono già separate prima dell'esportazione e possono essere trovate nella sezione **NLA_G** del *software* **Blender_G** e selezionate individualmente premendo la linea con il mouse e modificarle usando la scorciatoia *Shift+TAB*.

- **Creazione e modifica di *texture*:**

Ho salvato le **texture_G** come semplici immagini di tipo **.png_G**.

Entrambe le dimensioni della *texture* (larghezza e altezza) dovevano essere una potenza di 2.

Risoluzioni esempio:

- 256x256;
- 512x512;
- 1024x1024 (1K);
- 2048x2048 (2K).

Di norma, 1024 *pixels* corrispondono a 1 metro.

.glb: formato standard di un modello tridimensionale che legge il modello 3D come un file binario.

.png: formato di immagine raster senza perdita di qualità, ampiamente utilizzato per la grafica web e il design digitale.

Blender: *software* di modellazione ed animazione 3D usato per creare modelli 3D ed animazioni. **IK - Inverse Kinematics:** descrive il processo di calcolo della posizione delle articolazioni di un modello 3D in base alla posizione finale di una parte del corpo.

NLA - Nonlinear Animations: sistema di gestione delle animazioni in *Blender* che consente di combinare e sovrapporre diverse animazioni in modo non lineare.

rig: struttura scheletrica applicata a un modello 3D.

materiale: insieme di proprietà che definiscono l'aspetto visivo di un oggetto 3D, come colore, riflessione, trasparenza.

texture: immagine applicata a un modello 3D per fornire dettagli visivi, come colori e *pattern*.

- **Verifica e validazione:**

Il processo di verifica aveva lo scopo di garantire che il *software* sviluppato soddisfi i requisiti stabiliti e che sia conforme agli *standard* di qualità richiesti. Ho svolto due tipologie di verifica, ognuna è focalizzata sulla verifica di vari aspetti dell'applicazione:

analisi statica: l'analisi statica comportava il controllo del codice prima della sua esecuzione. Applicavo questo tipo di verifica non solo al codice, ma anche ai documenti del progetto.

Casi dove ho applicato l'analisi statica:

- individuazione di *bug* nel codice;
- individuazione di errori di battitura nei documenti;
- verifica della coerenza e completezza della documentazione prodotta;

analisi dinamica: l'analisi dinamica veniva eseguita all'esecuzione del *software*. Usavo l'analisi dinamica per controllare se erano presenti errori durante l'esecuzione dell'applicazione e dei suoi componenti.

Questo tipo di verifica mi permetteva di individuare malfunzionamenti, errori logici o comportamenti inattesi che potevano emergere solo durante l'esecuzione reale del *software*.

Le principali attività di analisi dinamica includevano:

- esecuzione di *test* di unità e di integrazione per verificare il corretto funzionamento delle singole componenti e della loro interazione;
- monitoraggio delle prestazioni e dell'utilizzo delle risorse durante l'esecuzione;
- individuazione e correzione di *bug* che si manifestano solo in fase di runtime.

2.5.5 Tecnologie scelte

Nome	Descrizione	Versione
GDScript	Linguaggio di programmazione di alto livello, con sintassi simile a <i>Python_G</i> , viene integrato con il motore di gioco <i>Godot</i>	(Legata a <i>Godot</i>)
GDShader	Linguaggio simile a <i>GLSL ES_G</i> 3.0, usato per la creazione di materiali e <i>shader</i> più complessi	(Legata a <i>Godot</i>)
Typst	Linguaggio utilizzato per la stesura dei documenti	0.13.1

Tabella 6: Linguaggi di programmazione scelti

GLSL ES - OpenGL Shading Language for Embedded Systems: linguaggio di shading utilizzato per scrivere *shader*.

Python: linguaggio di programmazione di alto livello, noto per la sua sintassi semplice e leggibile.

Nome	Descrizione	Versione
<i>Godot</i>	Il motore di gioco <i>open source</i> per lo sviluppo del videogioco	4.5-beta3-mono
<i>Blender</i>	<i>Software</i> di modellazione ed animazione 3D usato per creare i modelli 3D ed animazioni nel gioco	4.4.3

Tabella 7: *Softwares* scelti

Nome	Descrizione	Versione
<i>Git</i>	Servizio per il controllo della versione	2.50.1
<i>GitHub</i>	Servizio di <i>hosting</i> _G per i progetti <i>software</i> , utilizzato per la gestione del codice sorgente	-
<i>GitHub Actions</i>	Servizio di integrazione continua e distribuzione continua (<i>CI</i> _G / <i>CD</i> _G), utilizzato per compilare i documenti ad ogni <i>push</i> _G	-
<i>Notion</i>	Applicazione per la gestione dei progetti e la collaborazione	2.53

Tabella 8: Strumenti e servizi utilizzati

CI - Continuos Integration: processo di integrazione continua delle modifiche del codice in un *repository* condiviso, garantendo che il codice sia sempre in uno stato funzionante e testato.

CD - Continuos Delivery: processo di rilascio continuo delle modifiche del codice in produzione, garantendo che il *software* sia sempre in uno stato utilizzabile.

hosting: descrive il servizio che consente di archiviare e rendere accessibili *online* siti *web*, applicazioni o progetti *software*.

push: descrive l'azione di inviare le modifiche del codice a un *repository* remoto.

Nome	Descrizione	Versione
*.csv	« <i>Comma separated values</i> », <i>file</i> utilizzato per memorizzare le frasi nelle lingue diverse supportate dal gioco	-
*.ini	Tipo di <i>file plain-text</i> utilizzato per salvare i dati del gioco	-
*.glb	« <i>GLTF Binary</i> », <i>file</i> utilizzato per memorizzare i modelli 3D e le loro animazioni in formato binario, in modo da risparmiare spazio e migliorare le prestazioni	2.0.1

Tabella 9: Tipi di *file* scelti

2.5.6 Analisi dei rischi

2.5.6.1 Rischi organizzativi

Errata pianificazione dei tempi	
Descrizione	Un'errata pianificazione dei tempi può portare a ritardi nello sviluppo del progetto, con conseguente rischio di non rispettare le scadenze stabilite.
Probabilità	Alta
Pericolosità	Alta
Rilevamento	Monitoraggio delle attività pianificate e dei tempi di esecuzione ogni settimana
Piano di contingenza	Controllare le attività svolte tramite uno strumento di gestione del progetto (ad esempio <i>diagrammi di Gannt</i> _G e uso di <i>checklist</i> su <i>Notion</i>) e rivedere la pianificazione delle attività in caso di ritardi.

Tabella 10: Errata pianificazione dei tempi

diagrammi di Gannt: strumento di gestione dei progetti che rappresenta graficamente le attività pianificate nel tempo, mostrando la durata.

Impegni personali o universitari	
Descrizione	Impegni personali o universitari possono influenzare il tempo a disposizione per lo sviluppo del progetto, causando ritardi o interruzioni nello sviluppo.
Probabilità	Alta
Pericolosità	Media
Rilevamento	Monitoraggio delle attività pianificate e dei tempi di esecuzione ogni settimana
Piano di contingenza	Pianificare le attività in modo da tenere conto degli impegni personali o universitari, e rivedere la pianificazione delle attività in caso di imprevisti.

Tabella 11: Impegni personali o universitari

2.5.6.2 Rischi tecnici

Mancanza di competenze tecniche	
Descrizione	La mancanza di competenze tecniche può influenzare la qualità del prodotto software, causando ritardi nello sviluppo e problemi di integrazione
Probabilità	Media
Pericolosità	Alta
Rilevamento	Monitoraggio delle attività pianificate e dei tempi di esecuzione ogni settimana
Piano di contingenza	Formazione sulle tecnologie utilizzate e revisione della progettazione in caso di problemi tecnici

Tabella 12: Mancanza di competenze tecniche

Tecnologie non adeguate	
Descrizione	L'uso di tecnologie non adeguate può influenzare la qualità del prodotto <i>software</i> , causando problemi di prestazioni basse o <i>bug</i> .
Probabilità	Alta
Pericolosità	Alta
Rilevamento	Monitoraggio delle attività svolte e dei tempi di esecuzione ogni settimana
Piano di contingenza	Valutazione delle tecnologie utilizzate e revisione della progettazione in caso di problemi tecnici. In caso di problemi con le tecnologie utilizzate, si valuterà la possibilità di modificare la progettazione del gioco per adattarsi alle tecnologie disponibili

Tabella 13: Tecnologie non adeguate

2.5.6.3 Rischi di analisi e progettazione

Cambio dei requisiti	
Descrizione	Un cambiamento dei requisiti può influenzare la progettazione del sistema, causando ritardi nello sviluppo
Probabilità	Bassa
Pericolosità	Alta
Rilevamento	Comunicazione frequente con il relatore del progetto per garantire che i requisiti siano chiari e stabili
Piano di contingenza	Rivedere la progettazione del sistema in caso di cambiamenti dei requisiti, e valutare l'impatto sui tempi di sviluppo. In caso di cambiamenti significativi dei requisiti, si valuterà la possibilità di modificare la pianificazione delle attività per tenere conto dei nuovi requisiti

Tabella 14: Cambio dei requisiti

Errore nella progettazione dell'architettura	
Descrizione	Un errore nella progettazione dell'architettura può influenzare la qualità del prodotto <i>software</i> , causando ritardi nello sviluppo e problemi di integrazione
Probabilità	Media
Pericolosità	Alta
Rilevamento	Monitoraggio delle attività svolte e dei tempi di esecuzione ogni settimana, valutazione della progettazione dell'architettura
Piano di contingenza	Rivedere la progettazione dell'architettura in caso di problemi, e valutare l'impatto sui tempi di sviluppo

Tabella 15: Errore nella progettazione dell'architettura

Capitolo 3

Il progetto

In questo capitolo approfondisco tutti i processi del progetto: sviluppo, test e validazione. In pratica descrivo cosa ho fatto di preciso, e come l'ho svolto

3.1 Analisi dei requisiti

3.1.1 Attori

Nella stesura dei casi d'uso, ho riportato solo un attore, il **giocatore**, cioè l'utente che interagisce con il videogioco, controllando il personaggio e prendendo decisioni durante il gioco.

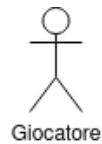


Figura 6: Attore principale

Nei casi d'uso del progetto, l'attore principale sarà sempre il giocatore.

3.1.2 Casi d'uso

3.1.2.1 Introduzione

Nelle seguenti sezioni descrivo i casi d'uso dell'applicazione, evidenziando le possibili azioni che il giocatore può svolgere all'interno del videogioco, e come può interagire con gli elementi o *entità* dell'ambiente circostante. Ad esempio, il giocatore può muovere il personaggio, saltare, prendere oggetti, interagire con altri personaggi, etc...

UC1 - Movimento

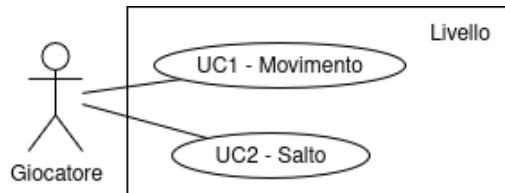


Figura 7: Diagramma UML use case sul movimento

Descrizione: il giocatore può muovere il personaggio in avanti, indietro, a sinistra e a destra utilizzando i tasti direzionali della tastiera o lo *stick* analogico sinistro del del *joypad_G*.

Precondizioni: il giocatore deve essere in un livello del gioco.

Postcondizioni: il personaggio si muove nella direzione desiderata e interagisce con l'ambiente circostante.

UC2 - Salto

Descrizione: il giocatore può far saltare il personaggio utilizzando un tasto specifico.

Precondizioni: il giocatore deve essere in un livello del gioco e deve essere libero di muoversi.

Postcondizioni: il personaggio esegue il salto.

UC3 - Interazione con *entità* automatica

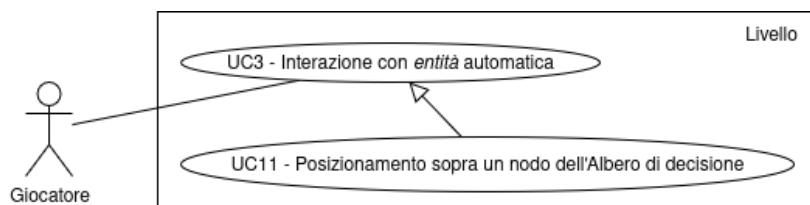


Figura 8: Diagramma UML sull'interazione automatica con un'*entità*

Descrizione: il giocatore si avvicina ad un'*entità* e l'interazione avviene automaticamente, mostrando un messaggio.

Precondizioni: il giocatore deve essere vicino ad un'*entità*.

Postcondizioni: il giocatore interagisce con l'*entità*.

Generalizzazioni: - Posizionamento sopra un nodo dell'Albero di decisione.

joypad: dispositivo di *input* dotato di pulsanti, levette e altri controlli per interagire con il gioco.

UC4 - Interazione con *entità* manuale

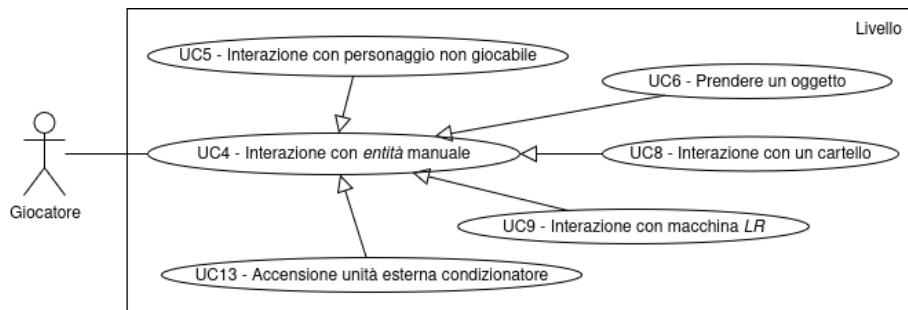


Figura 9: Diagramma UML sull'interazione manuale con un'*entità*

Descrizione: il giocatore si avvicina a un'*entità*, che può essere un cartello o un personaggio non giocabile, e vede l'*input* che deve premere per interagire.

Precondizioni: il giocatore deve essere vicino all'*entità*.

Postcondizioni: il giocatore ha premuto l'*input* per interagire.

Generalizzazioni:

- Interazione con un cartello.

- Interazione con macchina *LR*.
- Accensione unità esterna condizionatore.
- Interazione con personaggio non giocabile.
- Prendere un oggetto.

UC5 - Interazione con personaggio non giocabile

Descrizione: il giocatore si avvicina a un personaggio non giocabile e vede l'*input* che deve premere per interagire.

Precondizioni: il giocatore deve essere vicino a un personaggio non giocabile.

Postcondizioni: il giocatore ha premuto l'*input* per interagire.

UC6 - Prendere un oggetto

Descrizione: il giocatore può prendere un oggetto e poi muoversi con esso.

Precondizioni: il giocatore deve essere in un livello del gioco ,deve esserci un oggetto che può raccogliere davanti ad esso e non deve averne già uno.

Postcondizioni: il giocatore tiene l'oggetto e può muoversi con esso.

UC7 - Lasciare un oggetto

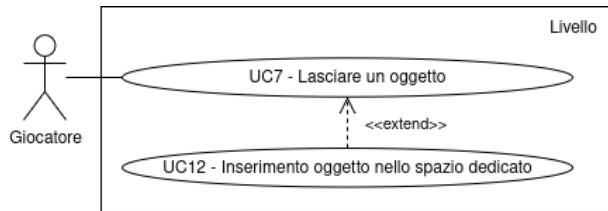


Figura 10: Diagramma UML sul lasciare un oggetto

Descrizione: il giocatore può lasciare un oggetto che sta portando.

Precondizioni: il giocatore deve essere in un livello del gioco e sta portando un oggetto.

Postcondizioni: il giocatore lascia l'oggetto e questo rimane nella posizione dove viene lasciato.

Estensioni: - Inserimento oggetto nello spazio dedicato.

UC8 - Interazione con un cartello

Descrizione: il giocatore si avvicina ad un cartello e vede l'*input* che deve premere per leggerlo.

Precondizioni: il giocatore deve essere vicino ad un cartello.

Postcondizioni: il giocatore ha premuto il tasto e legge il contenuto del cartello.

UC9 - Interazione con macchina LR

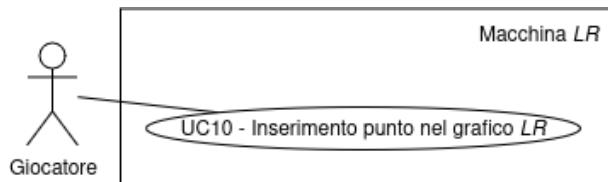


Figura 11: Diagramma UML sull'interazione con una macchina LR

Descrizione: il giocatore vuole interagire con la macchina *LR* (chiamata anche *Cannone LR*) per posizionare dei punti sul grafico della Regressione lineare.

Precondizioni: il giocatore deve trovarsi in un'area per interagire con la macchina.

Postcondizioni: il giocatore sta usando la macchina *LR*.

UC10 - Inserimento punto nel grafico LR

Descrizione: il giocatore vuole posizionare un punto sul grafico *LR*, mentre sta usando la macchina *LR*.

Precondizioni: il giocatore deve essere in utilizzo di una macchina *LR*.

Postcondizioni: il punto viene posizionato sul grafico.

UC11 - Posizionamento sopra un nodo dell'Albero di decisione

Descrizione: il giocatore si posiziona sopra un nodo dell'Albero di decisione.

Precondizioni: il giocatore deve essere in un livello del gioco e deve essere presente un Albero di decisione.

Postcondizioni: il giocatore è posizionato sopra un nodo dell'Albero di decisione.

UC12 - Inserimento dell'oggetto nello spazio dedicato

Descrizione: il giocatore posiziona l'oggetto che sta portando in uno spazio apposito.

Precondizioni: il giocatore deve portare un oggetto ed essere sopra un nodo finale dell'*albero*.

Postcondizioni: se l'oggetto è giusto, il giocatore viene avvisato ed il nuovo oggetto viene mostrato nell'apposito cartello.

UC13 - Accensione unità esterna condizionatore

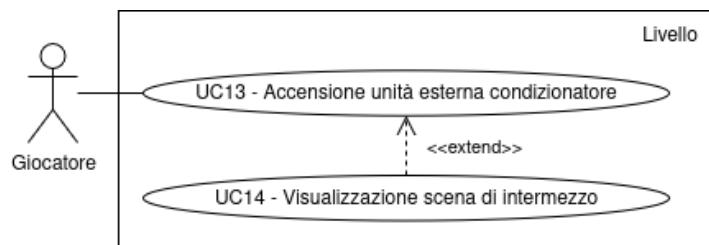


Figura 12: Diagramma UML sull'accensione di un'unità esterna di un condizionatore

Descrizione: il giocatore accende un'unità esterna di un condizionatore.

Precondizioni: il giocatore deve essere in un livello.

Postcondizioni: l'unità esterna del condizionatore viene accesa ed il giocatore non può più interagire con l'*entità*. Se sono state accese tutte le unità, parte la scena di intermezzo.

Estensioni: - Visualizzazione scena di intermezzo.

UC14 - Visualizzazione scena di intermezzo

Descrizione: il giocatore visualizza una scena di intermezzo.

Precondizioni: il giocatore deve soddisfare certe condizioni.

Postcondizioni: il giocatore visualizza la scena di intermezzo.

3.1.3. Requisiti

Ho diviso i requisiti in base al **tipo** e alla **priorità**.

- **Tipo:** possono essere
 - **Funzionali:** indicano una funzionalità del sistema;
 - **Qualità:** indicano le caratteristiche della qualità del prodotto, come un sistema deve essere o come il sistema deve esibirsi;
 - **Accessibilità:** indicano una caratteristica da soddisfare per rendere il gioco accessibile ad un numero maggiore di utenti.
- **Priorità:** possono essere
 - **Obbligatori:** necessari per considerare il prodotto completo;
 - **Desiderabili:** non strettamente necessari ma sono un valore aggiunto.

La tabella 16 mostra il totale dei requisiti, sempre divisi per tipo e priorità.

Tipologia	Obbligatori	Desiderabili	Totale
Funzionali	43	10	53
Qualità	4	-	4
Accessibilità	3	3	6
Totale			63

Tabella 16: Totale requisiti

3.2. Architettura

3.2.1. Concetti chiave di *Godot*

3.2.1.1. Nodi

I nodi sono i blocchi fondamentali del gioco. Sono come ingredienti in una ricetta. Ci sono dozzine di tipi che possono mostrare un'immagine, riprodurre un suono, rappresentare una camera, e molto altro. Tutti i nodi hanno le seguenti caratteristiche:

- un nome;
- proprietà modificabili;
- ricevono *callback* per aggiornarsi ad ogni $frame_G$
- si possono estendere con nuove proprietà e funzioni;
- si possono aggiungere a un altro nodo come figlio. [1]

Inoltre ad ogni nodo posso assegnare uno script, che estende il tipo di quel nodo e aggiunge nuove funzionalità.

I principali tipi di nodi forniti da *Godot* che ho utilizzato in questo progetto sono:

- **Node**: nodo base da cui vengono estesi tutti gli altri nodi, in questo progetto l'ho usato principalmente per assegnare classi e inserirle come figlie in altri nodi.
- **Node3D**: rappresenta un oggetto nello spazio tridimensionale.
 - **CharacterBody3D**: rappresenta un personaggio che si può muovere nel gioco, gestendo la sua posizione e interazioni.
 - **Camera3D**: rappresenta una telecamera nello spazio tridimensionale, che può essere utilizzata per visualizzare la *scena*.
 - **MeshInstance3D**: rappresenta un oggetto tridimensionale.
 - **CollisionShape3D**: rappresenta una forma di collisione nello spazio tridimensionale, l'ho utilizzata per gestire le interazioni fisiche tra gli oggetti.
 - **Area3D**: rappresenta un'area nello spazio tridimensionale. Questa classe l'ho principalmente utilizzata per gestire le interazioni tra gli oggetti che entrano ed escono da essa.
- **AnimationPlayer**: gestisce le animazioni degli oggetti nella *scena*, permettendo di riprodurre animazioni sul modello 3D, telecamere e altri nodi.
- **Control**: rappresenta un nodo UI_G , utilizzato per gestire gli elementi dell'interfaccia utente del gioco.

3.2.1.2. Scene

Un insieme di più nodi in un albero, come il personaggio nella figura 13, viene chiamato *scena*. Una volta salvata, la *scena* si presenta come un nuovo nodo nell'*editor*, dove posso aggiungerlo come figlio di un nodo esistente.

frame: unità di misura temporale utilizzata nei videogiochi e nelle animazioni. Tipicamente sono 60 in un secondo.

UI - User Interface: interfaccia grafica che consente all'utente di interagire con un'applicazione o un videogioco.

In questo caso, l'istanza della *scena* appare come nodo singolo con interni nascosti.

Le scene di consentono di strutturare il codice del gioco in qualunque modo si voglia. L'*editor* offre la possibilità di comporre nodi per creare nodi personalizzati e complessi, come un personaggio di gioco che si muove e salta, una barra della vita, una cesta con cui puoi interagire, e molto altro. [1]

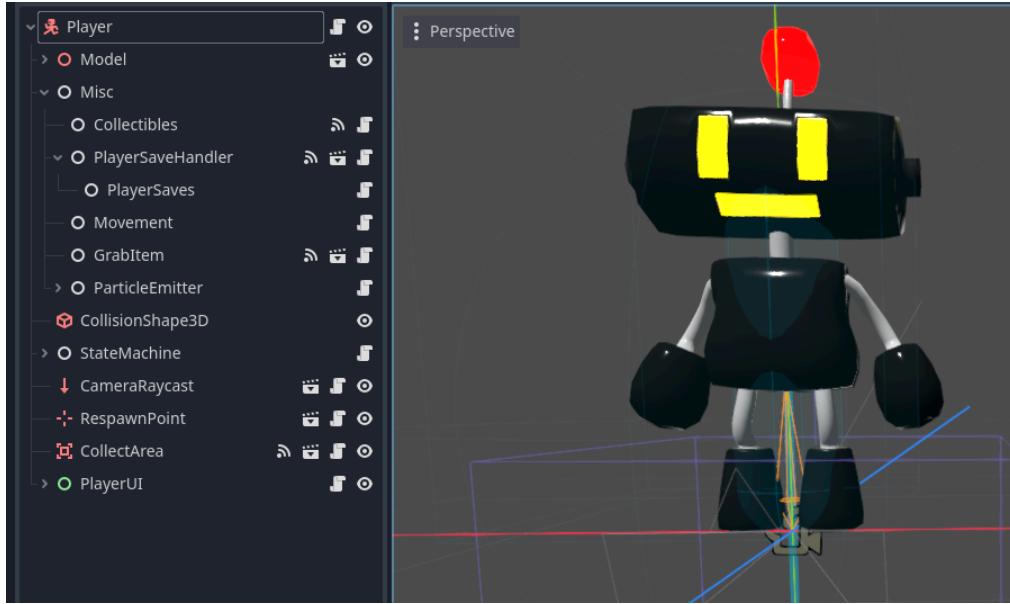


Figura 13: *Scena* del personaggio del giocatore

Nel corso dello *stage* ho utilizzato anche altre caratteristiche dei nodi.

- Posso salvare una *scena* sul disco locale e caricarla in seguito.
- Posso creare quante più istanze di una *scena*. Ad esempio, in alcuni livelli del gioco, ho inserito più personaggi non giocabili, ognuno con i suoi parametri.

3.2.1.3. Segnali

I segnali sono un modo per far comunicare i nodi in maniera asincrona in *Godot*. Ogni classe presenta dei segnali preimpostati ed emessi in determinati momenti, ad esempio quando un nodo viene caricato, questo emette il segnale **ready()**, oppure quando un bottone viene premuto, viene emesso il segnale **pressed()**. I segnali inoltre possono anche contenere dei parametri, che possono essere utilizzati per passare informazioni tra i nodi.

Infine, si possono creare segnali personalizzati, che possono essere emessi in qualsiasi momento dal nodo che li ha definiti tramite il metodo **signal.emit(...)**. Ho utilizzato molto questa funzionalità per far comunicare i nodi all'interno dei livelli e mandare variabili in maniera asincrona.

Ci sono due modi per collegare un segnale ad un altro nodo:

- tramite l'*editor*: selezionando il nodo che emette il segnale e trascinandolo sul nodo che deve ricevere il segnale, e selezionando il metodo che deve essere chiamato quando il segnale viene emesso;
- tramite codice: utilizzando il metodo `signal.connect(callable: Callable)` del nodo che emette il segnale, passando come parametro il nome del metodo che deve essere chiamato quando il segnale viene emesso.

3.2.2. Funzioni comuni

Molte classi del progetto presentano delle funzioni virtuali comuni, che vengono fornite dalle classi base del motore di gioco:

`+ready(): void`

Questa funzione viene chiamata quando il nodo entra nella scena, ovvero quando tutti i nodi figli sono stati caricati e il nodo è pronto per essere utilizzato.

In questa funzione è possibile inizializzare le variabili, collegare i segnali e impostare le proprietà del nodo.

È importante notare che questa funzione viene chiamata solo una volta, quando il nodo viene caricato per la prima volta nella scena, e non ad ogni *frame* del gioco.

`+process(delta: float): void`

Questa funzione viene chiamata ad ogni *frame* del gioco e permette di aggiornare lo stato della classe, ad ogni *frame* di inattività.

Un *frame* di inattività corrisponde ad un *frame* effettivo del gioco.

Il parametro `delta` rappresenta il tempo trascorso dall'ultimo *frame* di inattività, ed è utile per gestire le animazioni e le interazioni in modo fluido e coerente. [2]

`+physics_process(delta: float): void`

Questa funzione viene chiamata ad ogni *frame* di fisica del gioco, che di default è fisso 60 volte al secondo, anche nel caso il numero di *frame* al secondo del gioco sia inferiore o superiore.

Questa funzione è utile per gestire le interazioni fisiche tra gli oggetti, come ad esempio la gestione delle collisioni e la gestione della gravità.

Il parametro `delta` rappresenta il tempo trascorso dall'ultimo *frame* di fisica. [2]

`+_input_(event: InputEvent): void`

Funzione chiamata ogni volta che il gioco rileva un qualsiasi *input*, sia da tastiera che dal *joypad*. Il parametro `event` rappresenta l'*input* che chiama la funzione.

3.2.3. Classi del giocatore

3.2.3.1. Player

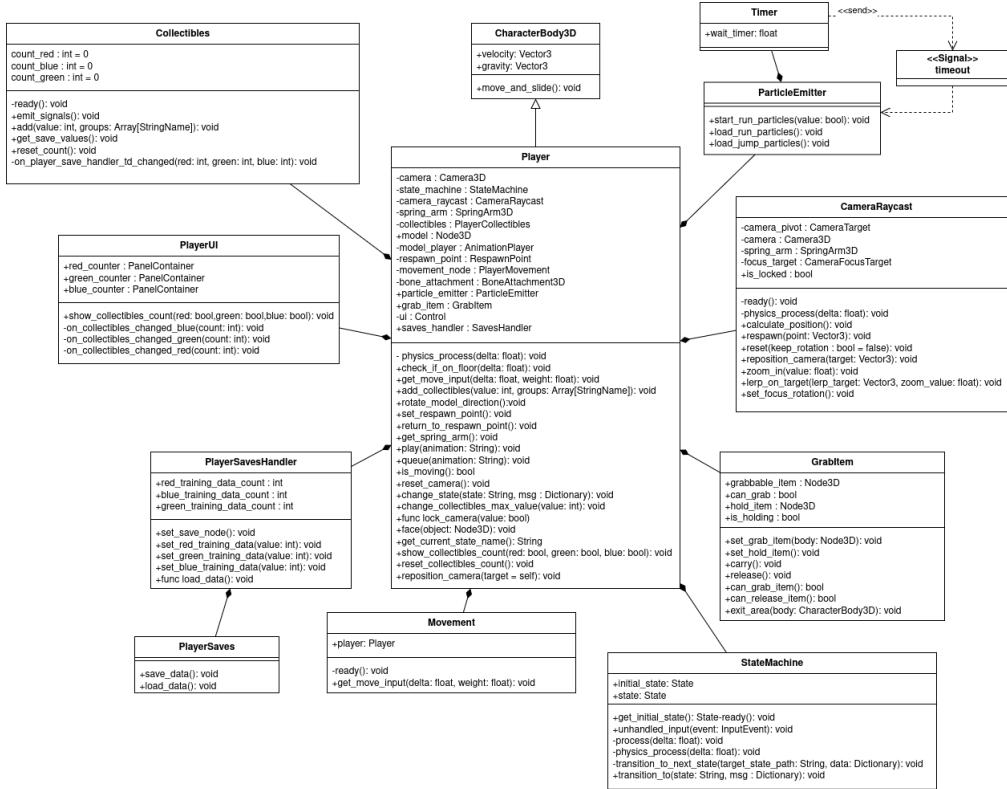


Figura 14: Diagramma UML delle classi del giocatore

Il giocatore può essere considerata la classe principale di tutta l'applicazione, attraverso il quale l'utente può interagire con la maggior parte del gioco.

Nonostante ci sia solo un giocatore presente nel gioco, questo non è un *singleton*, poiché per implementare un *singleton* in *Godot* è richiesto che questo sia caricato come *autoload* in ogni scena, e non c'è motivo di caricare il giocatore nel menu principale, all'avvio del gioco.

Molte variabili presenti nel giocatore sono riferimenti ai suoi nodi figli presenti nella scena, queste variabili sono precedute dalla parola chiave `@onready` nel codice. Similmente, molte funzioni della classe servono solo per accedere alle variabili dei suoi nodi figli.

Ho associato alla classe del giocatore delle classi che offrono funzionalità diverse,

autoload: meccanismo di *Godot* che consente di caricare automaticamente una risorsa all'avvio del gioco e mantenerla sempre attiva.

singleton: *design pattern* che garantisce ci sia solo un'istanza di una classe in tutto il gioco, rendendola accessibile da qualsiasi parte del codice.

3.2.3.2. StateMachine

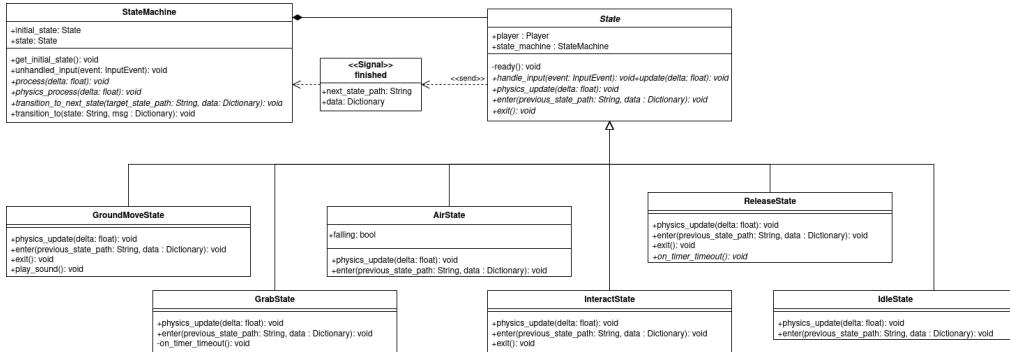


Figura 15: Diagramma UML sulla struttura della macchina di stati

Ho utilizzato la macchina di stati per controllare meglio i diversi stati in cui il personaggio del giocatore può trovarsi, ad esempio: il movimento, salto... L’uso della macchina di stati, inoltre, mi ha garantito una gestione più semplice del personaggio del giocatore e ha reso più facile aggiungere o modificare funzionalità a questo.

La classe **StateMachine** si occupa di gestire la transizione degli stati.

L’attributo **state** indica lo stato corrente del personaggio del giocatore. Quando riceve il segnale **finished(...)** dallo stato in cui si trova, si occupa di passare allo stato indicato dal segnale, passando gli eventuali dati contenuto nel **Dictionary** allo stato successivo.

Tutti gli stati ereditano dalla classe base astratta **State**. Questa include un riferimento al giocatore ed alla macchina di stati. Di seguito sono descritti tutte le classi degli stati del personaggio:

- **IdleState**: stato iniziale del personaggio. Questo stato viene chiamato quando è fermo per terra. Può passare a tutti gli altri stati in base agli input premuti in diverse condizioni. Ad esempio se il personaggio sta portando qualcosa ed il giocatore preme il tasto di interazione, il personaggio passa allo stato *Release*, ma se non sta portando niente, allora non succede niente. Invece se preme lo stesso tasto dentro un’area specifica, il personaggio passa allo stato *Interact*.
- **GroundMovementState**: il personaggio del giocatore passa allo stato *Ground-Movement* quando viene premuto un input per spostarsi rimanendo per terra. Come lo stato *Idle*, si può passare a tutti gli altri stati anche da questo, seguendo le stesse condizioni dello stato *Idle*.
- **AirState**: si può passare a questo stato in due condizioni: il personaggio cade da una piattaforma, o il giocatore preme il tasto per saltare. Nell’ultimo caso, lo stato precedente manda un valore **jump = true** all’interno del **Dictionary**. Lo stato controlla se è presente il medesimo valore non appena il personaggio entra in quello stato, ed in caso positivo, esegue il salto, caricando la rispettiva animazione e modificando la velocità verticale.

- **InteractState:** lo stato *Interact* indica che il personaggio del giocatore è impegnato ad interagire con un'altra entità, ad esempio mentre parla con un personaggio non giocabile o legge un cartello.
- **GrabState:** il personaggio del giocatore passa allo stato *Grab* quando il giocatore preme il tasto per prendere un oggetto vicino. Importante notare che questo stato rappresenta solo quando il personaggio prende un oggetto, dopo aver svolto l'azione, il personaggio torna allo stato *Idle*, cambiando le animazioni in modo che rispecchino il fatto che sta portando un oggetto.
- **ReleaseState:** quando il giocatore preme di nuovo il tasto per prendere un oggetto mentre il personaggio sta portando un oggetto, questo passa allo stato *Release* e lascia l'oggetto. Come il suo stato opposto, una volta lasciato l'oggetto, il giocatore torna allo stato *Idle*.

La figura 16 mostra il flusso degli stati.

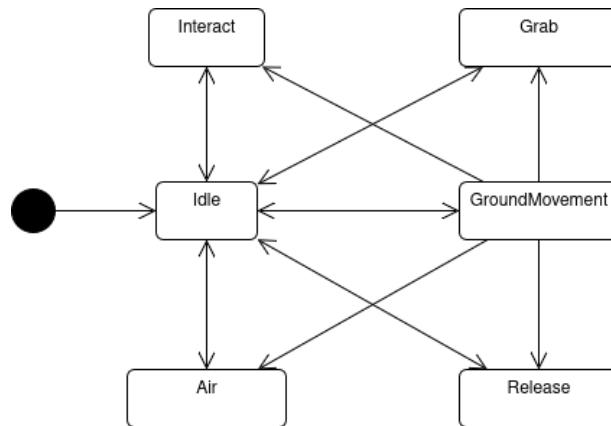


Figura 16: Diagramma sul flusso degli stati del giocatore

3.2.4. Entità interagibili

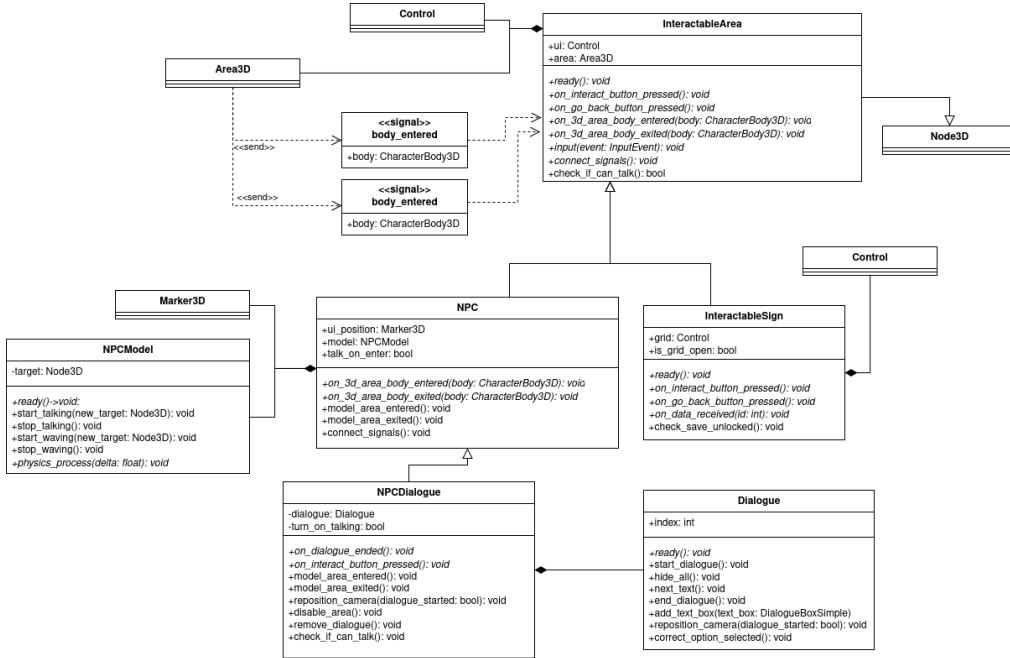


Figura 17: Diagramma UML degli oggetti con cui il giocatore può interagire

Nei livelli sono presenti diverse *entità* con cui il giocatore può interagire. Di seguito descrivo i diversi tipi di *entità*, e le classi che le compongono:

- **InteractableArea**: classe base astratta che fornisce i metodi alle classi figlie. La classe è composta da un'Area3D, che invia i segnali quando il giocatore entra ed esce, e da una classe Control che rappresenta la UI che il giocatore visualizza quando entra.
- **NPC**: rappresenta un personaggio non giocabile che ha assegnato una semplice frase come messaggio. Questa frase viene visualizzata in una classe SimpleProjectLabel. Presenta anche una classe Marker3D che segna la posizione della UI, e una classe NPCModel che gestisce le animazioni del modello 3D del personaggio.
- **InteractableSign**: rappresenta un cartello che il giocatore può leggere. Il cartello può contenere diverse informazioni, come una lista o un grafico. Il contenuto del cartello è inserito in un'altra classe Control.
- **NPCDialogue**: rappresenta un personaggio non giocabile che, a differenza della classe NPC, presenta un dialogo. Il giocatore può interagire con il personaggio e visualizzare il dialogo premendo il rispettivo tasto.

3.2.5. Struttura base di un livello

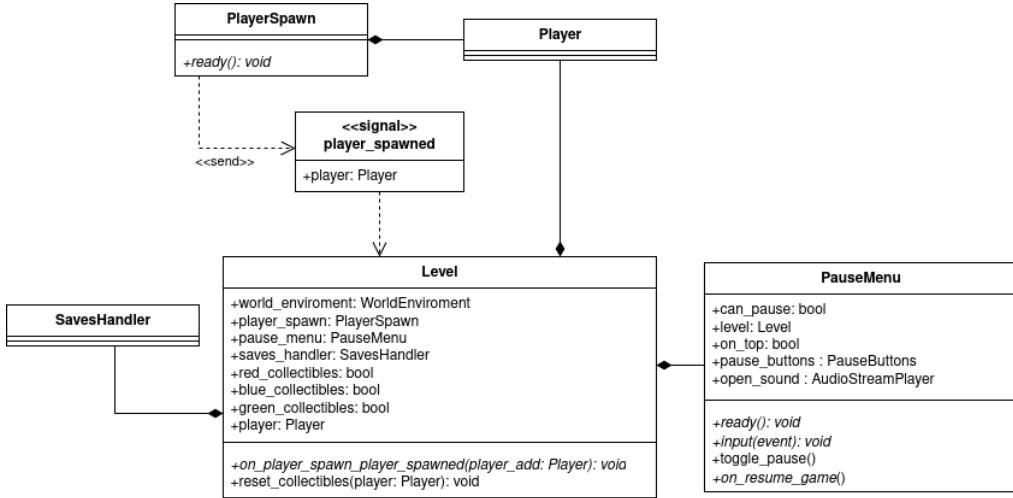


Figura 18: Diagramma delle classi di un livello base

- **Level**: classe del livello, non svolge molte funzioni visto che i componenti possono comunicare tra di loro attraverso i segnali. Gli attributi booleani **red_collectibles**, **blue_collectibles** e **green_collectibles** stabiliscono quali tipi di *collezionabili* sono presenti nel livello, e quindi quali far visualizzare nella *UI* del giocatore.
- **PlayerSpawn**: classe che si occupa di generare il giocatore nella posizione in cui si trova. Appena generato il giocatore viene assegnato alla classe **Level**
- **PauseMenu**: il menu di pausa, questo viene caricato quando il giocatore preme il rispettivo tasto, mettendo in pausa tutta la scena.

3.2.6. LRCannon

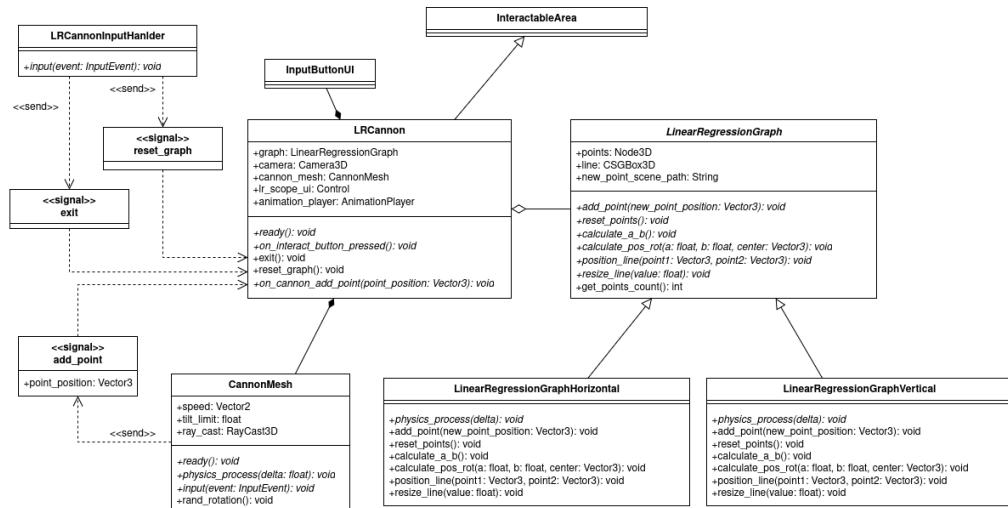


Figura 19: Diagramma sul funzionamento di un grafico *Linear Regression* nel gioco

La classe **LRCannon** rappresenta il cannone nel livello per aggiungere nuovi dati nel grafico della Regressione lineare.

Eredita da **InteractableArea** e infatti il giocatore può interagirci quando entra dentro l'area apposita.

Quando il giocatore preme l'*input* per interagire, la telecamera viene cambiata ed il giocatore entra nello stato *Interact*. La classe è composta da:

- **CannonMesh**: si occupa della rotazione del cannone quando questo è attivo e quando viene inserito un nuovo punto nel grafico.
- **LRCannonInputHandler**: gestisce gli *input* del giocatore quando questo sta controllando il *cannone LR*.

3.2.7. LinearRegressionGraph

Classe base astratta usata per i due tipi di grafico presenti nel livello: orizzontale e verticale. Si occupa di svolgere le operazioni di regressione lineare per ottenere la formula della retta $y = a + bx$.

Tuttavia, non posso applicare la formula direttamente ad un oggetto 3D.

Il metodo **calculate_a_b()** si occupa di calcolare le variabili **a** e **b** della formula della retta con le seguenti formule:

$$a = \frac{\sum y \sum x^2 - \sum x \sum xy}{n(\sum x^2) - (\sum x)^2} \quad b = \frac{n \sum xy - (\sum x)(\sum y)}{n \sum x^2 - (\sum x)^2}$$

Dove x e y sono le coordinate dei punti, ed n è uguale al numero di punti che abbiamo nel grafico.

Tutte le sommatorie sono state calcolate in un ciclo *for* e dopo inserite come variabili nelle formule.

```
1 func calculate_a_b() -> void:
2     var sum_x : float = 0.0
3     var sum_x2 : float = 0.0
4     var sum_y : float = 0.0
5     var sum_xy : float = 0.0
6     var num : float = 0.0
7
8     for i in points.get_child_count():
9         num += 1
10        sum_x += points.get_child(i).global_position.x
11        sum_y += points.get_child(i).global_position.z
```

 GDScript

```

12     sum_x2 += (points.get_child(i).global_position.x)**2
13     sum_xy += (points.get_child(i).global_position.z)*\
14         (points.get_child(i).global_position.x)
15
16 #Vengono poi applicate le formule

```

Codice 2: Funzione `calculate_a_b()`

Il metodo `calculate_pos_rot(a: float, b: float)`, poi, prende la formula della retta, e posiziona due punti nel grafico lungo la retta calcolata dal metodo precedente. Questi due punti vengono passati al metodo successivo.

Il metodo `position_line(pos1: Vector3, pos2: Vector3)`, infine, posiziona il modello 3D della retta in mezzo ai due punti e lo ruota in modo che li intersechi.

Le trasformazioni globali vengono poi modificate in base al tipo della classe:

- **LinearRegressionGraphHorizontal**: il grafico orizzontale, parallelo al terreno.
- **LinearRegressionGraphVertical**: il grafico verticale, perpendicolare al terreno.

3.2.8. Livello Albero di decisione

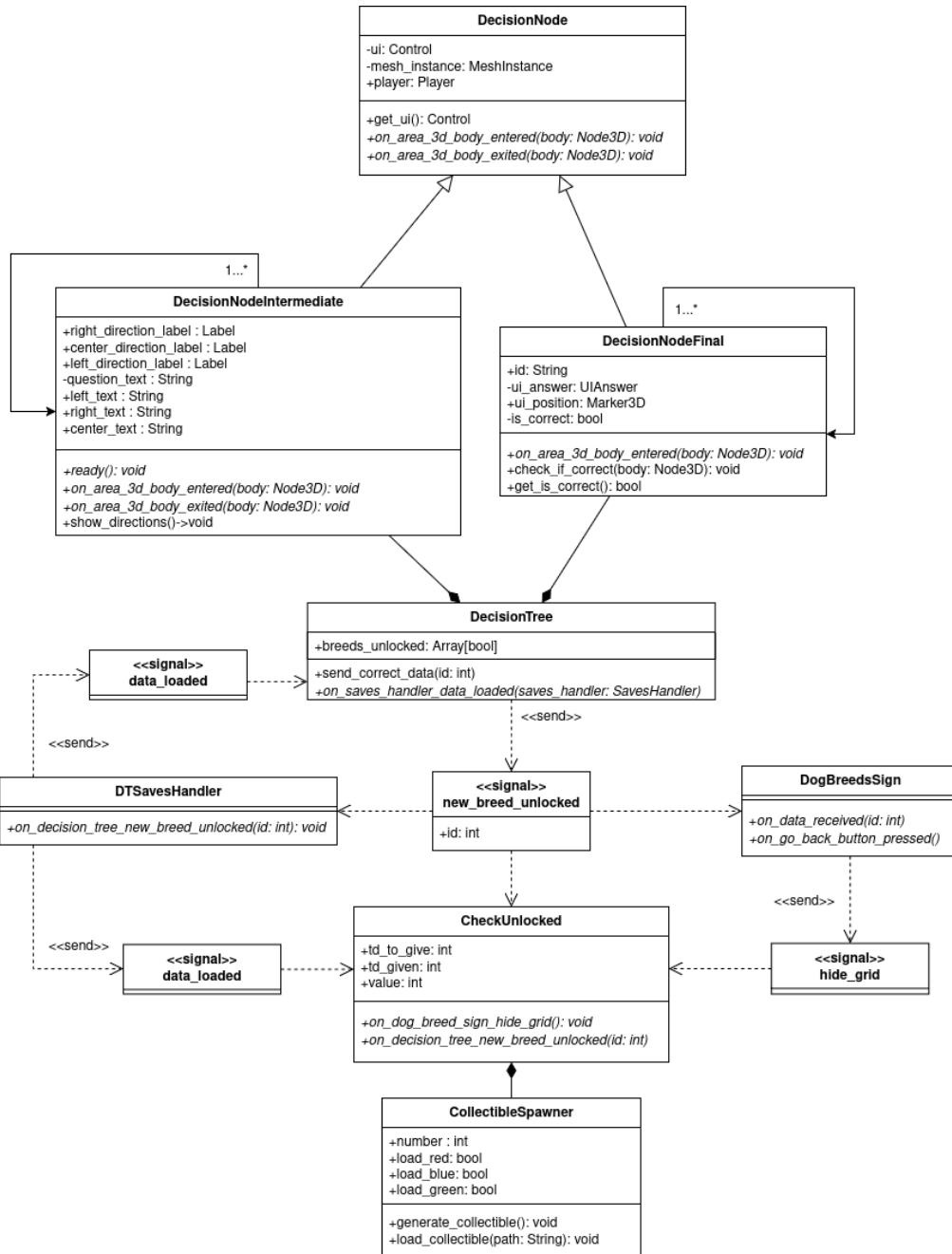


Figura 20: Diagramma sul funzionamento dell'Albero di decisione

- **DecisionTree**: la classe che rappresenta un Albero di decisione nel livello composta da più istanze di **DecisionNodeFinal** e **DecisionNodeIntermediate**, inserite tutte come nodi figli nella *scena*. Si occupa di inviare i segnali agli altri nodi presenti nel livello.

- **DecisionNode**: classe base astratta per i due tipi di nodi presenti nell'albero: *intermediate* e *final*. Fornisce i metodi virtuali `on_area_3d_body_entered(body: CharacterBody3D)` e `on_area_3d_body_exited(body: CharacterBody3D)` che vengono chiamati quando entra un oggetto di tipo `CharacterBody3D` nell'area sopra la piattaforma.

Il comportamento poi viene modificato dalle classi figlie.

- **DecisionNodeIntermediate**: quando entra il giocatore nell'area, viene visualizzata la *UI* con le indicazioni da seguire;
- **DecisionNodeFinal**: quando entra un cane nell'area, controlla se l'`id` di questo corrisponde all'`id` associato all'istanza.
- **DogBreedsSign**: oltre all'Albero di decisione nel livello è presente anche un cartello con cui il giocatore può interagire e visualizzare le razze die cani che ha indovinato.

La classe `DogBreedsSign` rappresenta questo cartello. Questa, è composta da una classe `DogSignUI` che è il contenuto del cartello, contenente tutte le razze dei cani che il giocatore ha indovinato.

Quando il cartello viene chiuso, emette il segnale `hide_grid()` che chiama il metodo `on_dog_breed_sign_hide_grid()` nella classe `CheckUnlocked`.

- **CheckUnlocked**: classe che si occupa di controllare le razze di cani sbloccate e tenere il conto di quelle nuove che il giocatore non ha ancora controllato, nell'attributo `td_to_give`.

Il valore di questo attributo viene modificato all'inizio del caricamento del livello e quando il giocatore indovina una nuova razza nell'Albero di decisione, ed è la differenza tra l'attributo `value` e il valore `td_given`.

Al caricamento del livello, riceve il segnale `data_loaded()` dal nodo che gestisce i salvataggi `DT Saves Handler`, assegna il valore dell'attributo `td_given`. Quando riceve il segnale `new_breed_unlocked` dall'Albero di decisione, `value` aumenta di 1, ed aggiorna il valore di `td_to_give`.

Quando riceve il segnale `hide_grid` dal cartello, chiama la funzione per generare i *collezionabili* tanti quanti il valore di `td_to_give`.

3.2.9. Livello *Causalità*

3.2.9.1. Struttura del livello

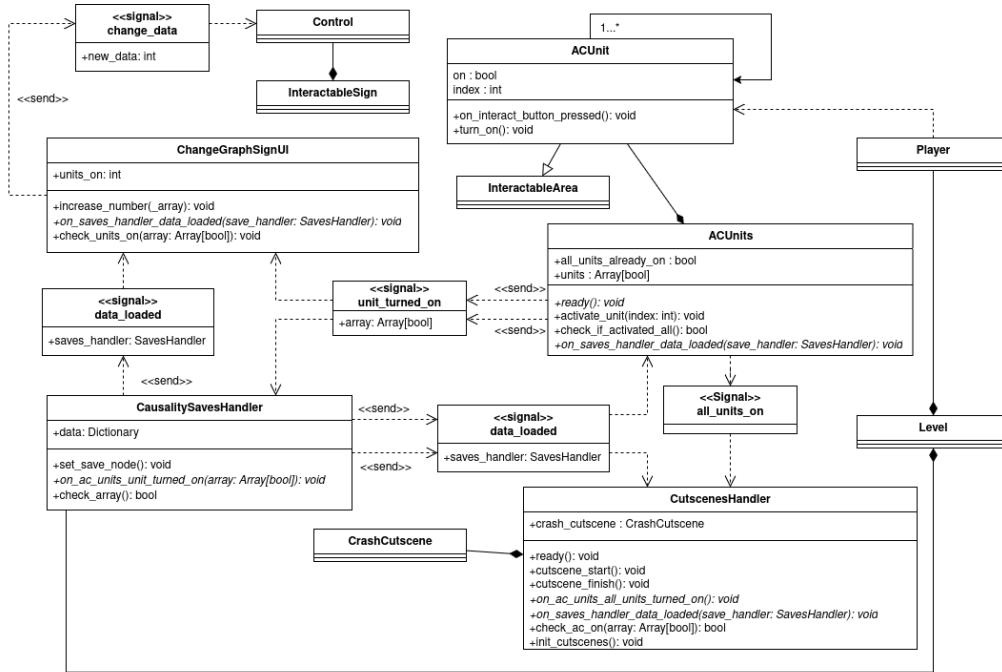


Figura 21: Diagramma del livello *Causalità*

- **ACUnit**: rappresenta un condizionatore. Eredita da **InteractableArea**. Il giocatore, quando entra nell'area, può premere il tasto di interazione per accenderlo. Quando viene acceso, il valore dell'`array` nodo padre, in questo caso **ACUnits**, viene aggiornato con il giusto indice.
- **ACUnits**: si occupa di gestire tutte le istanze di **ACUnit**, inserite come nodi figli nella scena. Quando viene acceso un condizionatore, emette il segnale `unit_turned_on(index: int)`, passando direttamente l'array aggiornato come argomento nel segnale. Quando tutti i condizionatori sono stati accesi, manda il segnale `all_units_on()`, usato in questo caso per far iniziare la scena di intermezzo.
- **CutsscenesHandler**: si occupa di gestire le scene di intermezzo nel livello, inserite come nodi figli nella scena. Nonostante avessi pianificato di usare la classe per gestire più scene di intermezzo, alla fine ne è presente solo una. Questa classe svolge anche il ruolo da mediatore, ricevendo i segnali dal livello e mandandoli ai nodi figli, gestendo il traffico dei segnali.
- **ChangeNPCIceCreamBehaviour**: si occupa di cambiare il comportamento del rispettivo personaggio non giocabile. Viene assegnata ad un nodo figlio del nodo del personaggio.

Funziona nello stesso modo della classe descritta prima, però avviene solo un cambio del dialogo e non c'è la modifica del comportamento.

- **NPCIceCreamSave:** lo scopo della classe è quello di caricare il gruppo di persone davanti alla gelateria nel caso il livello venga caricato quando già tutti i condizionatori sono stati accesi.

Le persone vengono caricate quando la classe riceve il segnale `change_specific_values()`, in quanto vengono caricate solo ed esclusivamente al caricamento del livello.

- **ChangeSignUI:** questa classe si occupa di cambiare il contenuto del rispettivo cartello. Viene assegnata ad un nodo figlio del nodo del cartello.

Funziona nello stesso modo delle classi che cambiano il dialogo o comportamento dei personaggi. Quando riceve il segnale `change_values()`, cambia il contenuto del cartello, rimpiazzandolo con l'istanza assegnata alla classe.

3.3. Verifica e validazione

3.3.1. Nomenclatura *test*

Di seguito sono elencate le metodologie di *testing* che ho utilizzato per verificare e validare il prodotto *software*. Le metodologie di *testing* sono suddivise in quattro categorie:

- ***test* di unità:** *test* che verificano il corretto funzionamento di singole unità del codice, ho svolto questi *test* con l'*add-on* della community *GUT - Godot Unit Test*;
- ***test* di integrazione:** *test* che verificano il corretto funzionamento dell'interazione tra più unità dell'applicazione, ho svolto anche questi con l'*add-on GUT*;
- ***test* di sistema:** *test* che verificano il corretto funzionamento del sistema nel suo complesso, inclusi i requisiti funzionali e non funzionali, comprendono anche *test* sulle prestazioni, e li ho svolti utilizzando gli strumenti forniti da *Godot*;
- ***test* di accettazione:** *test* che verificano se il prodotto è pronto per essere rilasciato.

Tipologia	Eseguiti
Unità	44
Integrazione	37
Sistema	10
Accettazione	4
Totale	95

Tabella 17: Totale *test* eseguiti

add-on: estensione che migliora o amplia le funzionalità di un *software* esistente.

3.4. Risultati ottenuti

3.4.1. Meccaniche dei livelli

In questa sezione mostro delle immagini raffiguranti le meccaniche principali dei livelli.

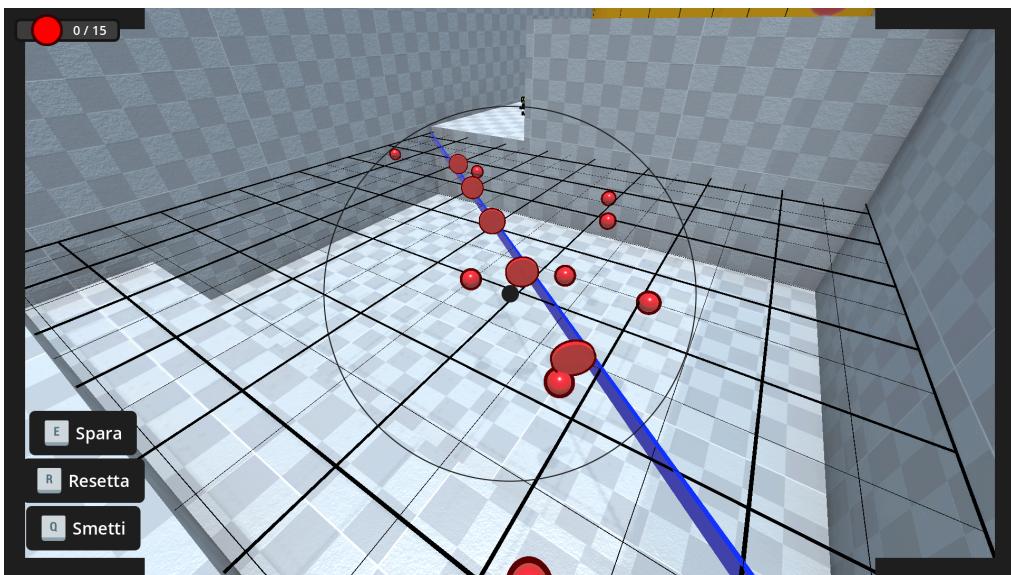


Figura 22: Immagine del cannone *LR* per posizionare nuovi punti nel grafico della Regressione lineare



Figura 23: Immagine della scena di intermezzo del livello *Causalità*

3.4.2. Requisiti soddisfatti

Tipologia	Obbligatori	Desiderabili	Totale	Soddisfatti
Funzionali	43	10	53	53
Qualità	4	-	4	4
Accessibilità	3	3	6	4
Totale			63	61

Tabella 18: Tabella con grado di superamento requisiti

La tabella 18 mostra il grado di superamento dei requisiti funzionali, di qualità e di accessibilità, con un totale di 63 requisiti. Non sono riuscito a soddisfare alcuni dei requisiti di accessibilità desiderabili per i seguenti motivi:

- «**Il gioco deve supportare il sistema operativo MacOS**»: nonostante *Godot* permetta di compilare le applicazioni in un eseguibile per il sistema operativo MacOS, non ho potuto testare il gioco in una macchina con sistema operativo MacOS, quindi non potevo garantire il suo corretto funzionamento;
- «**Il gioco deve supportare una risoluzione bassa, ad esempio 640×360px**»: sotto una certa risoluzione, gli elementi della *UI* vengono tagliati dalla finestra, rendendo difficile la navigazione nei menu o leggere le istruzioni sugli *input* da premere.

3.4.3. Test superati

Tipologia	Eseguiti	Superati
Unità	44	44
Integrazione	37	37
Sistema	10	10
Accettazione	4	4
Totale	95	95

Tabella 19: Tabella con grado di superamento *test* eseguiti

La tabella 19 mostra il totale dei *test* eseguiti, suddivisi per tipologia, con un totale di 95 *test* eseguiti e superati con successo.

3.4.4. Quantità di prodotti

A fine *stage* ho completato la stesura di 8 documenti:

- Analisi dei requisiti;
- Glossario;
- Piano di progetto;
- Piano di qualifica;
- Manuale utente;
- Specifica tecnica.

Inoltre, durante il periodo di *stage* ho sviluppato 2 applicazioni dello stesso progetto. La tabella 20 riporta queste applicazioni.

Prodotto	Descrizione
PoC Proof of Concept	Demo del progetto, che ha come scopo principale quello di dimostrare la fattibilità del concetto, implementando le funzionalità principali
MVP Minimum Viable Product	Versione minima del prodotto che si può considerare pronta per il rilascio. Rappresenta il prodotto finale dello <i>stage</i> e include tutte le funzionalità principali richieste nel progetto

Tabella 20: Tabella delle applicazioni prodotte nello *stage*

Capitolo 4

Conclusioni

In questo capitolo effettuo una retrospettiva sul progetto e sulla mia esperienza di stage, analizzando le esperienze acquisite durante il periodo. Infine metto a confronto gli argomenti insegnati dal percorso di studi e quelli richiesti per lo sviluppo del progetto.

4.1 Obiettivi stage soddisfatti

4.1.1 Grado di soddisfazione degli obiettivi

Come descritto precedentemente nella sezione 2.4, avevo prefissato degli obiettivi insieme al relatore. Nella tabella 21 ho riportato questi obiettivi con il loro codice ed il loro stato di completamento.

Obiettivo	Stato
O-1	✓
O-2	✓
O-3	✓
D-1	✓
D-2	✓
D-3	
D-4	

Tabella 21: Obiettivi del progetto

Sempre dalla tabella 21 possiamo notare che non sono riuscito a soddisfare gli ultimi due obiettivi desiderabili. Questi due obiettivi non sono stati soddisfatti a causa di limitazioni tecniche e di risorse.

In generale, sono soddisfatto del lavoro svolto e degli obiettivi raggiunti, anche se ci sono stati alcuni ostacoli lungo il cammino.

4.1.2 Obiettivi non superati

4.1.2.1 D-3 | Uso di linguaggi come C++ per migliorare le prestazioni

Durante lo sviluppo del *PoC*, ho utilizzato il linguaggio *GDScript* per sviluppare il progetto. Andare a rimpiazzare il codice, anche se parzialmente, con un altro linguaggio di programmazione come *C++*, avrebbe consumato una gran parte del tempo a disposizione, senza garantire un miglioramento significativo delle prestazioni.

Per dimostrare quest'ultima parte, ho inserito nella figura 24 un grafico preso all'interno dell'*editor* durante l'esecuzione del gioco che mostra il tempo di *rendering_G* per ogni *frame* in millisecondi.



Figura 24: Grafico del tempo di compilazione di ogni *frame* del gioco

Dal grafico possiamo notare diverse cose:

- I tempi di compilazione e *rendering* tra un *frame* e l'altro, sono generalmente stabili, con alcune eccezioni presenti durante la transizione di un livello ad un altro a causa del caricamento delle risorse.
- Il grafico è diviso in 2 sezioni, la prima, a sinistra, mostra il tempo richiesto dalla *CPU_G*, la seconda, a destra, mostra il tempo richiesto dalla *GPU_G*. Come possiamo vedere, la *GPU* richiede più tempo rispetto alla *CPU*. Cambiare il linguaggio di programmazione aumenterebbe le prestazioni solo della *CPU*, visto che *GDScript* e *C++* sono linguaggi di alto livello letti e compilati dalla *CPU*. Ma come possiamo vedere dalla figura 24, questo non aumenterebbe le prestazioni, poiché i lavori sono svolti in parallelo, e la *GPU* rimarrebbe il *bottleneck_G* in questo caso.

bottleneck: punto nel sistema che limita le prestazioni complessive, spesso a causa di risorse insufficienti o sovraccarico di lavoro.

CPU - Central Processing Unit: unità di elaborazione centrale del computer, responsabile dell'esecuzione delle istruzioni e del controllo delle operazioni.

GPU - Graphics Processing Unit: unità di elaborazione grafica, specializzata nel rendering delle immagini e nella gestione delle operazioni grafiche.

rendering: processo di generazione dell'immagine finale di oggetti 3D, che coinvolge il calcolo della luce, delle ombre ed altro.

4.1.2.2 D-4 | Implementazione di un modello di *LLM*

Durante lo sviluppo del progetto, ho provato ad implementare un *LLM* per generare automaticamente i dialoghi dei personaggi non giocabili nel gioco.

Nonostante la comunità di *Godot* abbia sviluppato diversi *add-on* per integrare gli *LLM* locali nei giochi, ho incontrato diversi fattori che hanno reso difficile l'implementazione:

- **Largo uso della *VRAM*:** il problema principale degli *LLM* locali è il fatto che richiedono molta *VRAM*, risorsa che nei videogiochi viene già utilizzata per gestire le texture, i modelli 3D e altri asset grafici. Questo porta rapidamente all'esaurimento della memoria disponibile, causando rallentamenti o crash dell'applicazione, soprattutto su hardware non di fascia alta.
- **Limitazioni della macchina personale:** la macchina su cui ho svolto il progetto aveva a disposizione 1GB di *VRAM*, limitando molto l'utilizzo di un modello. Durante un *test*, ho provato a integrare un *LLM* “leggero”, con mezzo miliardo di parametri, in modo da usare meno *VRAM*. Il risultato che ho ottenuto è stata una stringa di frasi e parole incomprensibili.
- **Lingua:** dato che avevo prefissato l'obiettivo di supportare la lingua italiano ed inglese nel gioco, anche le frasi generate dal modello dovevano essere in italiano o in inglese. Gli unici *LLM* “leggieri” che sono riuscito a trovare erano solo in lingua inglese, e se supportavano più lingue significava che i parametri effettivamente usati per l'inglese e italiano erano ancora di meno.
- **Dimensioni:** un *LLM* locale ha dimensioni che variano da centinaia di MB a diversi *gigabyte*. Integrare un modello del genere avrebbe aumentato notevolmente la dimensione finale del gioco, rendendolo meno accessibile per gli utenti con connessioni *internet* lente, limitate o con poca memoria nel dispositivo. Nel caso avessi voluto usare un *LLM* diverso per ogni lingua, le dimensioni dell'applicazione finale sarebbero aumentate ulteriormente.

4.2 Esperienze acquisite

4.2.1 Competenze tecniche

Durante lo *stage*, ho avuto l'opportunità di approfondire le mie conoscenze in diversi ambiti tecnici, in particolare nello sviluppo di giochi con *Godot*.

Ho approfondito le conoscenze che avevo sul linguaggio *GDScrip*t e a sfruttare le funzionalità del motore per implementare meccaniche di gioco complesse ed originali.

Ho imparato a lavorare con modelli 3D, animazioni, sistemi di particelle ed effetti visivi per creare un'esperienza di gioco coinvolgente ed interessante.

Inoltre, ho acquisito competenze nella gestione delle risorse e nell'ottimizzazione delle prestazioni del gioco.

4.2.2 Competenze trasversali

Durante lo *stage*, ho anche sviluppato competenze trasversali, come la capacità di comunicare efficacemente con i membri del *team*.

Ho imparato a gestire il tempo in modo più efficiente, utilizzando strumenti di gestione del progetto per pianificare le attività, controllare le scadenze e monitorare i progressi.

4.2.3 Gestione dei rischi

4.2.3.1 Introduzione

Durante lo *stage*, ho dovuto affrontare dei problemi che avevo previsto una possibilità nel presentarsi nella sezione 2.5.6.

Ho dovuto affrontare questi problemi e trovare delle soluzioni per superarli, in modo da poter completare il progetto nei tempi previsti e senza compromettere la qualità del lavoro.

4.2.3.2 Errata pianificazione dei tempi

Lo sviluppo dell'applicazione ha avuto un andamento più veloce di quello che avevo previsto durante la pianificazione e ho finito per avere più tempo a disposizione rispetto a quello pianificato. Quindi, per occupare le ore di scarto ho aggiunto nuove funzionalità non obbligatorie al gioco. Ad esempio ho aggiunto un livello *tutorial* ed il supporto per i *joypad* com dispositivo di *input*, con cambio di visualizzazione dei tasti nella guida *UI*, tutte funzionalità che non avevo pianificato di sviluppare.

4.2.3.3 Impegni personali o universitari

Durante il periodo di *stage*, mi mancava ancora un esame da recuperare. Ho dovuto sostenere l'esame due volte, in due giorni diversi, e questo mi ha fatto perdere alcune ore di *stage*. Nonostante le ore perse, sono comunque riuscito a completare il progetto nei tempi previsti.

4.2.3.4 Tecnologie non adeguate

Visto che *Godot* è un motore di gioco relativamente nuovo rispetto alla concorrenza attuale (*Unity*, *Unreal Engine...*), non è ancora molto diffuso e non ha una grande comunità di sviluppatori.

Questo mi ha reso più difficile trovare risorse e supporto durante lo sviluppo del progetto, ed alcune di queste risorse avevano un uso limitato o non erano aggiornate.

Ad esempio, volevo implementare un sistema di *CI/CD* (Continuous Integration/Continuous Deployment) per automatizzare i *test*, tuttavia le risorse che ho trovato erano per lo più obsolete o non funzionanti, dunque ho svolto i *test* manualmente. Ho avuto lo stesso problema anche per implementare uno strumento di *code coverage*.

Al contrario, invece, alcuni *add-on* per *Godot* si sono rivelati molto utili e ben documentati, ad esempio *GUT*, che mi ha facilitato lo svolgimento dei *test* di unità e di integrazione.

4.2.3.5 Errore nella progettazione dell'architettura

Durante lo sviluppo del *PoC* avevo come priorità svolgere un'applicazione che dimostrasse le funzionalità principali, ma ho trascurato alcuni aspetti architettonici che si sono rivelati problematici in seguito. Ad esempio, non avevo ben definito la gestione dei cambi degli stati del giocatore e delle transizioni tra le diverse schermate, portando a un codice più complesso e difficile da mantenere. Ho dovuto quindi rivedere parte dell'architettura per migliorare la modularità e la manutenibilità del codice.

code coverage: unità di misura che indica la percentuale di codice sorgente coperto da test automatici.

4.3 Differenza tra *stage* e percorso studi

Alcuni temi che ho dovuto affrontare durante lo *stage*, erano temi che avevo già affrontato durante il percorso di studi.

- *GDSCript* è un linguaggio di programmazione orientato agli oggetti, fondamento su cui si basa anche *C++*, che ho dovuto imparare per svolgere un altro progetto durante il percorso di studi.
- I temi del principale del gioco di *Machine Learning*, l'i avevo già approfonditi in precedenza durante un corso di studi. Nello specifico, mi ha aiutato molto a trovare temi su cui sviluppare le meccaniche del gioco come l'Albero di decisioni e la Regressione lineare ed il loro funzionamento.
- Durante il progetto del corso di *Ingegneria del software*, ho avuto modo di approfondire gli *LLM* insieme ad altri miei colleghi. Anche se alla fine non ho implementato questi nel gioco, mi hanno fornito una base teorica utile per comprendere meglio le tecnologie di intelligenza artificiale.

Al contrario, altri temi non erano stati trattati durante il percorso di studi, o erano stati trattati, ma in modo superficiale.

Questo riguardo tecnologie più specifiche e non molto di uso comune, come l'uso di un motore di gioco, o tecnologie di ambito diverso da quello insegnato nel corso di studi, come modellazione 3D.

Non ritengo che inserire questi argomenti nel percorso di studi sarebbe utile, dato che il campo dell'informatica è molto vasto e in continua evoluzione, e non è possibile coprire tutti gli argomenti in modo approfondito.

4.4 Pensieri finali

L'esperienza di *stage* è stata molto positiva e formativa, permettendomi di mettere in pratica sia le conoscenze acquisite durante il percorso di studi, sia quelle che ho acquisito personalmente, e di sviluppare nuove competenze tecniche e trasversali.

Ho avuto l'opportunità di lavorare su un progetto stimolante e innovativo, che mi ha permesso di approfondire conoscenze di mio interesse personale su *Game Design* e modellazione 3D in un ambiente professionale e collaborativo.

Ho avuto l'occasione di confrontarmi con professionisti del settore, imparando da loro e ricevendo *feedback* costruttivi sul mio lavoro.

Ritengo che questa esperienza mi abbia fornito una base solida per affrontare future sfide nel campo dello sviluppo di giochi e grafica 3D, e sono entusiasta di continuare a crescere e continuare ad imparare nuove tecnologie e metodologie in questo settore, sia in maniera personale che professionale.

Glossario

.glb – GLTF (Graphics Library Transmission Format) Binary: formato standard di un modello tridimensionale che legge il modello 3D come un file binario, permettendo una lettura e *rendering* più veloce e minimizzando lo spazio occupato dal *file*. 16.

.png – Portable Network Graphics: formato di immagine raster senza perdita di qualità, ampiamente utilizzato per la grafica web e il design digitale. 16.

add-on: estensione o componente aggiuntivo che migliora o amplia le funzionalità di un *software* esistente. 44.

Alberi di decisione: modello predittivo utilizzato in statistica e *Machine Learning*, che rappresenta le decisioni e le loro possibili conseguenze sotto forma di un albero, facilitando l'interpretazione e la visualizzazione delle scelte. 8.

autoload: meccanismo di *Godot* che consente di caricare automaticamente una risorsa all'avvio del gioco, rendendola disponibile in tutte le scene senza doverla caricare manualmente. 34.

Blender: *software* di modellazione ed animazione 3D *open source* usato per creare modelli 3D ed animazioni. 16.

bottleneck: in italiano, *collo di bottiglia*, termine utilizzato in informatica per descrivere un punto di congestione o limitazione nelle prestazioni di un sistema, che può influenzare negativamente l'efficienza complessiva. 50.

brainstorming: tecnica di generazione di idee in gruppo, in cui i partecipanti sono incoraggiati a esprimere liberamente le proprie idee senza giudizio, al fine di stimolare la creatività e trovare soluzioni innovative a un problema. 5.

CD – Continuos Delivery: pratica di sviluppo *software* che consente di rilasciare frequentemente e in modo affidabile nuove versioni del *software*, garantendo che il codice sia sempre in uno stato pronto per la produzione. 18.

CI – Continuos Integration: pratica di sviluppo *software* che consente di integrare frequentemente le modifiche del codice in un repository condiviso, garantendo che il codice sia sempre in uno stato funzionante e testato. 18.

code coverage: metodologia di analisi del codice sorgente che misura la percentuale di codice eseguito durante i test, aiutando a identificare le aree non testate e migliorare la qualità del software. 53.

CPU – Central Processing Unit: unità di elaborazione centrale di un computer, responsabile dell'esecuzione delle istruzioni e della gestione delle operazioni aritmetiche e logiche. 50.

database: insieme organizzato di dati, generalmente memorizzato e gestito in modo da facilitarne l'accesso e la manipolazione. In ambito *software*, i database sono utilizzati per archiviare informazioni in modo strutturato, consentendo operazioni di ricerca, aggiornamento e gestione dei dati. 3.

diagrammi di Gannt: strumento di gestione dei progetti che rappresenta graficamente le attività pianificate nel tempo, mostrando la durata, le dipendenze e le scadenze delle attività in un formato visivo facile da comprendere. 20.

frame: unità di misura temporale utilizzata nei videogiochi e nelle animazioni, che rappresenta un singolo fotogramma di un'animazione o di un ciclo di gioco. Tipicamente sono 60 in un secondo (60 FPS - *Frames Per Second*). 31.

Game Design: disciplina che si occupa della progettazione e dello sviluppo di giochi, sia da tavolo che digitali, considerando aspetti come la meccanica di gioco, la narrazione, l'estetica e l'interazione con il giocatore. 7.

GDScript: linguaggio di programmazione specifico per il motore di gioco Godot, progettato per essere semplice e intuitivo, con una sintassi simile a *Python*. Viene utilizzato per scrivere *script* che controllano la logica del gioco, le interazioni e le funzionalità. 15.

GLSL ES – OpenGL Shading Language for Embedded Systems: linguaggio di shading utilizzato per scrivere shader per applicazioni embedded, come giochi e grafica in tempo reale. 17.

GPU – Graphics Processing Unit: unità di elaborazione grafica di un computer, progettata per gestire e accelerare la creazione di immagini e video, nonché per eseguire calcoli complessi legati alla grafica 3D. 50.

hosting: servizio che consente di archiviare e rendere accessibili online siti web, applicazioni o progetti software, fornendo le risorse necessarie per il loro funzionamento e la loro distribuzione. 18.

IK – Inverse Kinematics: soluzione usata nell'ambito dell'animazione 3D. Si tratta di semplificare l'animazione calcolando il movimento di altre ossa o articolazioni in base all'ultimo osso della catena. Ad esempio, automatizza il movimento del braccio muovendo solo la mano, anziché ruotare singolarmente braccio, avambraccio e mano. Questo metodo risulta anche molto più simile a come ci si muove naturalmente. 16.

joypad: dispositivo di input utilizzato principalmente per come dispositivo di *input* nei videogiochi, dotato di pulsanti, leve e altri controlli per interagire con il gioco. 26.

LLM – Large Language Model: modello di intelligenza artificiale progettato per comprendere e generare testo in linguaggio naturale, addestrato su grandi quantità di dati testuali per svolgere compiti come la traduzione, la risposta a domande e la generazione di contenuti. 5.

ML – Machine Learning: disciplina che si occupa dello sviluppo di algoritmi e modelli statistici che permettono ai computer di apprendere dai dati e migliorare le proprie prestazioni nel tempo senza essere esplicitamente programmati. 8.

materiale: insieme di proprietà che definiscono l'aspetto visivo di un oggetto 3D, come colore, riflessione, trasparenza e altre caratteristiche ottiche. In *Godot*, i materiali possono essere applicati a modelli 3D per controllare il loro aspetto durante il *rendering*. 16.

Microsoft Teams: piattaforma di comunicazione e collaborazione sviluppata da Microsoft, che fornisce chat, videoconferenze, condivisione di file e lavoro di gruppo integrato con gli strumenti forniti da Microsoft. 3.

motore di gioco: software progettato per facilitare lo sviluppo di videogiochi, fornendo strumenti e funzionalità per la gestione della grafica, della fisica, dell'audio e di altre componenti del gioco. 12.

MVP – Minimum Viable Product: Versione minima di un prodotto che include solo le funzionalità essenziali per essere utilizzato dagli utenti. 14.

Nearest Neighbor: algoritmo di apprendimento automatico utilizzato per la classificazione e la regressione, che si basa sull'idea di trovare i punti dati più vicini a un dato punto di input e fare previsioni in base a questi punti. 8.

NLA – Nonlinear Animation: sistema di gestione delle animazioni in *Blender* che consente di combinare e sovrapporre diverse animazioni in modo non lineare, permettendo una maggiore flessibilità e controllo sulle animazioni dei modelli 3D. 16.

open source: modello di sviluppo software che promuove la collaborazione e la condivisione del codice sorgente, consentendo a chiunque di utilizzare, modificare e distribuire il software liberamente. 12.

OpenGL – Open Graphics Language: linguaggio di programmazione grafica utilizzato per creare applicazioni 3D e 2D, fornendo un'interfaccia standardizzata per l'interazione con la scheda grafica del computer. 11.

PascalCase: pratica di scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole. 15.

PoC – Proof of Concept: una dimostrazione pratica che ha lo scopo di verificare la fattibilità o il potenziale di un'idea, concetto o soluzione. È spesso utilizzato nelle fasi iniziali di un progetto per validare il funzionamento teorico e pratico, incluso il modo in cui diverse componenti del sistema possono integrarsi tra loro per raggiungere l'obiettivo prefissato. 14.

push: operazione che consente di inviare le modifiche locali del codice a un *repository* remoto, aggiornando così la versione del codice condiviso con altri membri del *team*. 18.

Python: linguaggio di programmazione di alto livello, noto per la sua sintassi semplice e leggibile, ampiamente utilizzato in vari ambiti come lo sviluppo web, l'analisi dei dati, l'intelligenza artificiale e la scienza dei dati. 17.

Regressione lineare: modello statistico utilizzato per analizzare la relazione tra una variabile dipendente e una o più variabili indipendenti, assumendo che questa relazione sia lineare. 8.

rendering: processo di generazione dell'immagine finale di oggetti 3D, che coinvolge il calcolo della luce, delle ombre, dei riflessi, dei materiali ed altro. 50.

rig: struttura scheletrica applicata a un modello 3D che consente di animarlo tramite la manipolazione di ossa e articolazioni. 16.

shader: programma che calcola l'aspetto visivo di un oggetto 3D, determinando come la luce interagisce con le superfici. 11.

singleton: design pattern garantisce che esista un'unica istanza di una classe, garantendo un punto di accesso globale a essa. 34.

snake_case: pratica di scrivere parole composte separando le parole tramite trattino basso, solitamente con le prime lettere delle singole parole in minuscolo. 15.

StageIT: evento orientato al lavoro organizzato dall'Università degli Studi di Padova, dedicato agli studenti per aiutarli a trovare aziende dove svolgere l'attività di *stage*. 7.

Support Vector Machines: algoritmo di apprendimento automatico utilizzato per la classificazione e la regressione, che cerca di trovare il margine ottimale che separa le diverse classi nel piano. 8.

temperatura: parametro che controlla la casualità delle risposte generate da un LLM. Valori più bassi rendono le risposte più conservative e focalizzate, mentre valori più alti aumentano la creatività e la varietà delle risposte. 6.

termine: termine esempio per dimostrare come funziona il glossario. vi

texture: immagine bitmap applicata a un modello 3D per fornire dettagli visivi, come colori e pattern. Ne esistono di vario tipo e possono essere utilizzate, ad esempio, per dare colore al modello 3D o modificare il valore della luce riflessa da questo. 16.

ticket: segnalazione o richiesta registrata in un sistema di tracciamento (come *Github Issues* o *Jira*) che descrive un problema, una funzionalità da implementare o un'attività da svolgere all'interno di un progetto *software*. 14.

tutorial: concetto usato per indicare una guida introduttiva ad un determinato argomento. In questo caso, rappresenta il livello introduttivo di un videogioco, progettato per insegnare all'utente i concetti base del gioco, ad esempio i comandi. 9.

UI – User Interface: interfaccia grafica che consente all'utente di interagire con un'applicazione o un videogioco. Può essere composta da pulsanti, menu, finestre di dialogo o altri elementi interattivi. 31.

Bibliografia

- [1] Juan Linietsky, Ariel Manzur, e comunità di Godot, «Nodi e Scene», 2014. [Online]. Disponibile su: https://docs.godotengine.org/it/4.x/getting_started/step_by_step/nodes_and_scenes.html
- [2] Juan Linietsky, Ariel Manzur, e comunità di Godot, «Processi di inattività e fisica», 2014. [Online]. Disponibile su: https://docs.godotengine.org/it/4.x/tutorials/scripting/idle_and_physics_processing.html