

Università degli Studi della Basilicata

Dipartimento di Matematica, Informatica ed Economia

Corso di Laurea in Scienze e Tecnologie Informatiche



Tesi di Laurea Triennale

## **“Studio e sviluppo di una soluzione didattica per l'apprendimento basata sul game base learning”**

### **RELATORE**

Prof. Santomauro Michele

### **CANDIDATO**

Venafro Daniele

Matricola 56893

**Anno accademico 2022/2023**



## Indice generale

1 Introduzione.....	1
2 Descrizione Progettazione dell'Applicazione.....	2
2.1 Scelta della Piattaforma.....	2
2.2 Mondo di Gioco, Grafica e Progressione.....	2
2.3 Delineare le Meccaniche di Gioco: ScriptConsole.....	4
2.4 Delineare le Meccaniche di Gioco: Doc.....	9
2.5 Delineare le Meccaniche di Gioco: Modalità esercizio.....	10
2.6 Delineare le Meccaniche di Gioco: Broker MQTT.....	15
2.7 Concept della Storia.....	16
2.8 Menu e Opzioni.....	16
3 Introduzione a Godot Game Engine.....	18
3.1 Interfaccia.....	18
3.2 Nodi.....	19
3.3 Script.....	20
4 Uno Sguardo alla Logica Implementativa.....	23
4.1 Addon Godot.....	23
4.2 Controllo Personaggio e Camera.....	23
4.3 Classi Statiche.....	26
4.4 Modalità esercizio.....	27
4.5 ScriptConsole.....	34
4.6 Doc.....	41
5 I livelli.....	42
5.1 Livello 1.....	42
5.2 Livello 2.....	44
6 Conclusioni.....	48
6.1 Considerazioni Finali.....	48
6.2 Sviluppi Futuri.....	49

# 1 Introduzione

L'obiettivo di questo progetto di tesi è quello di realizzare un "videogioco didattico" per quanto riguarda il mondo della programmazione, ovvero un videogioco che possa essere uno strumento utile sia per i docenti, che potranno utilizzarlo come eserciziario all'interno delle loro lezioni, sia per gli studenti, che potranno imparare divertendosi.

Per la realizzazione del progetto ho utilizzato il game engine Godot, un engine Open Source che permette la creazione di videogiochi 2D o 3D per molteplici piattaforme: PC, console e anche mobile. Per questo progetto è stato scelto il PC come piattaforma di riferimento. Il linguaggio di programmazione utilizzato è C#, anche se l'engine supporta anche altri linguaggi come: GDScript(un linguaggio proprietario dell'engine) oppure C++.

L'intento principale del progetto era proprio quello di creare una applicazione che potesse essere di supporto agli insegnanti, ma soprattutto agli studenti accompagnandoli nel loro percorso di crescita come programmatori, inizialmente attraverso piccole interazioni con le meccaniche di gioco, che portano l'utente a familiarizzare con la programmazione, fino ad arrivare al punto in cui gli studenti stessi possono mettere mani al codice sorgente per creare nuovi livelli o intere nuove meccaniche di gioco.

Tutto il progetto è stato infatti sviluppato con l'idea di essere completamente Open Source permettendo a tutti non solo di utilizzare l'applicazione, ma anche di mettere mani al progetto della stessa per modificare o aggiungere codice, permettendo quindi al progetto di evolversi ed arricchirsi nel tempo in modo da diventare sempre più utile nel suo ruolo di *"supporto alla formazione"*.

Pertanto la maggior parte dell'attenzione che ho dedicato al progetto si è rivolta verso la creazione dell'architettura software, in modo tale da creare una solida base che implementasse tutte le API utili e necessarie per semplificare la creazione di funzionalità future, non solo per me, ma anche e soprattutto per chi si troverà a sviluppare su questa applicazione in futuro.

## **2 Descrizione Progettazione dell'Applicazione**

### **2.1 Scelta della Piattaforma**

Inizialmente come piattaforma di sviluppo avevo scelto Unity, un engine gratuito molto famoso su cui già avevo fatto esperienza autonomamente, tuttavia proprio nel periodo in cui stavo per iniziare lo sviluppo, l'azienda proprietaria di Unity ha annunciato delle modifiche dei termini riguardanti le politiche di utilizzo della piattaforma. Queste modifiche non erano state comunicate nel migliore dei modi e lasciavano quindi forti dubbi sulle condizioni di utilizzo reali, per questo motivo ho deciso di orientarmi su un engine Open Source che già da diverso tempo stava aumentando in popolarità, ovvero Godot.

Infatti la sua natura Open Source la rendeva una scelta più sicura nel lungo periodo e permetteva la programmazione in C#, stesso linguaggio di Unity, rendendo più indolore il cambio di piattaforma.

Pensando inoltre alla natura Open Source di questo progetto, Godot sembrava una scelta molto pertinente e dopo aver provato alcuni giochi sviluppati su questo engine per valutarne la bontà, primo fra tutti il gioco "Cassette Beasts" disponibile sul PC Game Pass di Microsoft (ma anche Steam e console), ho deciso di sviluppare su questa piattaforma. L'intero progetto è stato quindi realizzato utilizzando Godot versione 4.1.1 stabile in linguaggio C#.

### **2.2 Mondo di Gioco, Grafica e Progressione**

Con Godot è possibile creare giochi con ambienti 2D o 3D. Per questo progetto ho deciso di creare un gioco 2D, in modo da poter disegnare e realizzare tutti gli elementi grafici in autonomia utilizzando una tavoletta grafica e Photoshop.

Infatti, data la natura Open Source del progetto utilizzare risorse grafiche acquistate sarebbe impossibile in quanto chiunque potrebbe ottenere le medesime visitando la pagina GitHub. Inoltre personalmente ho più esperienza nel disegno 2D che la modellazione 3D, pertanto questo approccio

si è rivelato più semplice da attuare. Per le risorse Audio sono state utilizzate tracce gratuite Royalty free, recuperate online sul sito "[pixabay.com](http://pixabay.com)".



*SpriteSheet dell'avatar del giocatore*

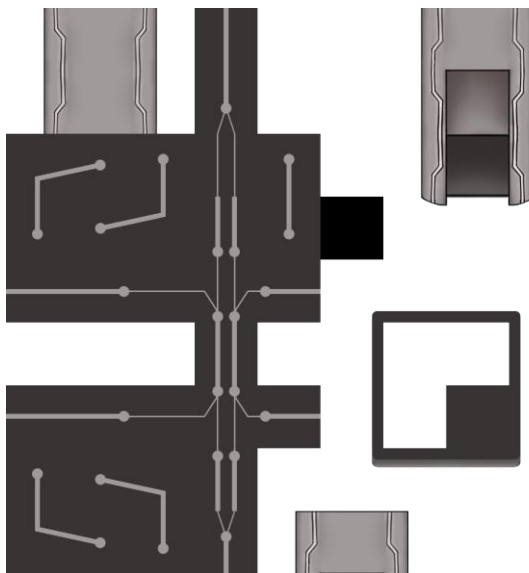
Per prima cosa ho quindi realizzato lo **spriteSheet**, una immagine contenente i disegni dei vari frame dell'animazione dell'avatar del personaggio.

Per il mondo di gioco ho utilizzato il sistema delle **tilemap** che permette di dividere il mondo di gioco in caselle di una dimensione prefissata ed associare ad ognuna di queste caselle una immagine ottenuta da un **tileset**, ovvero una immagine contenente tutte le grafiche possibili per ogni casella.

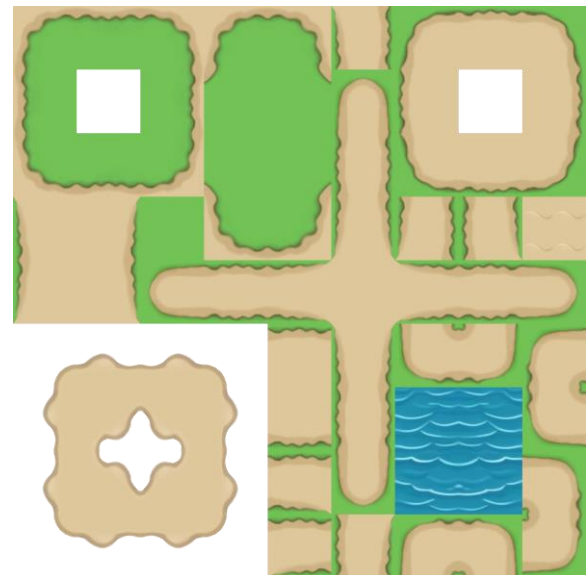
Per questo progetto è stata usata una tilemap con una grandezza di 128x128 pixel, in modo da creare un mondo di gioco ben definito. Spesso in questo tipo di videogiochi si utilizza una grafica in stile **pixel art**, ovvero con dei disegni a bassa densità di pixel in modo da rendere ognuno di questi pixel ben visibili andando così a semplificare il disegno e rendendo più semplice creare animazioni convincenti, in quanto il poco dettaglio rende più semplice mascherare eventuali difetti. Molti giochi adottano questo stile grafico anche per richiamare una sorta di effetto nostalgia nei confronti di giochi più datati.

Un tempo infatti le console avevano molti più limiti di oggi, in particolare legati alla quantità di memoria, per questo la grafica pixel art permetteva di creare modelli di gioco convincenti e molto leggeri da memorizzare. Nel mio caso ho preferito una grafica più definita semplicemente perché non ho molta dimestichezza con la creazione di grafiche in pixel art.

Questo gioco avrà quindi una grafica 2D con visuale isometrica di tipo Top-Down, molto ispirata a quella che avevano giochi come “Pokemon Rosso Fuoco” o “Pokemon Smeraldo” su gameboy advance, con la differenza che in questo caso non verrà usata una grafica “pixellata”. Questi sono i due tileset realizzati per il gioco.



*Tileset livello 1*



*Tileset livello 2*

Per quanto riguarda la progressione, il gioco sarà diviso in piccoli livelli consecutivi. Una volta completato un livello, si potrà accedere al successivo. Riavviando l'applicazione sarà possibile ricominciare dall'inizio oppure giocare uno qualsiasi dei livelli sbloccati in precedenza.

In questo modo gli studenti, una volta presa confidenza con l'engine e la scrittura del codice, potranno cimentarsi nella creazione di nuovi livelli per arricchire l'esperienza di gioco prendendo come esempio quelli già disponibili.

### **2.3 Delineare le Meccaniche di Gioco: ScriptConsole**

Come detto inizialmente l'obiettivo principale del gioco vuole essere quello di far prendere confidenza con il codice e l'ideazione di algoritmi agli studenti

di informatica alle prime armi, come ad esempio gli studenti dei primi anni delle superiori. Per questo motivo inizialmente era stata presa in esame l'idea di permettere agli utenti di scrivere codice durante il **runtime**, ovvero durante l'esecuzione, dell'applicazione.

Tuttavia questa possibilità lasciava spazio a diverse problematiche:

1. Complessità della funzionalità da implementare;
2. Difficoltà legate al prevedere il comportamento dell'utente;
3. Possibilità che codice concettualmente o sintatticamente errato generasse crash indesiderati nell'applicazione;

Se infatti la prima problematica è abbastanza ovvia e comprensibile, le altre due non sono affatto banali o trascurabili.

L'incapacità di prevedere il comportamento dell'utente rende molto difficile il lavoro di **Game Design** (ovvero la creazione delle varie soluzioni di gioco come puzzle da risolvere o comunque meccaniche di gioco necessarie alla progressione nello stesso), inoltre dando tutto quel potere all'utente c'è il forte rischio che faccia degli errori causando crash o andando a modificare dei parametri interni all'applicazione alterandone la progressione.

Se è vero che è possibile arginare la seconda parte del problema scrivendo il codice in modo tale che quello scritto dall'utente non abbia i permessi per interagire con determinati componenti dell'applicazione, è altrettanto vero che gestire un codice sintatticamente corretto ma concettualmente sbagliato può essere molto più problematico.

Guardando la cosa dal punto di vista di un giovane studente: se il gioco con cui sto interagendo crasha o si blocca ad ogni mio errore, costringendomi a riavviare l'applicazione e perdendo il progresso fatto, l'esperienza finirà inevitabilmente con il diventare più frustrante che divertente, facendo venire meno uno degli obiettivi iniziali, ovvero quello di formare divertendo.

È diventato quindi subito ovvio che fosse necessario un ambiente più controllato, che permettesse agli studenti di avere vari approcci per sviluppare le loro capacità di **problem solving**, ma restando comunque in un numero ristretto e prevedibile di possibilità.



Ragionando su tutto questo ho realizzato che la cosa importante non fosse che l'applicazione compilasse il codice scritto dall'utente, ma solo che ne desse l'impressione. Era necessario che gli studenti scrivessero del codice reale e che potessero vederne gli effetti durante il runtime, ma questo non significa necessariamente che quel codice debba essere davvero compilato ed eseguito.

Ho deciso quindi di realizzare un **"finto compilatore"**, ovvero un componente che fosse in grado di leggere del testo scritto dall'utente, individuare le singole linee di codice ed interpretarle in modo tale da ricondurle a dei metodi precedentemente preparati nel codice di gioco.

Ad esempio scrivendo un for in codice C#, il finto compilatore analizzerà la stringa di testo ricevuta in input, capirà che si tratta di un for e comincerà a cercare i parametri dello stesso:

- La dichiarazione della variabile contatore (tipicamente `int i = 0`);
- La condizione del ciclo (es. `i < N`);
- Il passo o step del ciclo (es. `i++`);

Una volta individuati questi dati nella stringa verrà richiamato un metodo precedentemente preparato che riceverà i suddetti dati come parametri ed eseguirà un vero for, all'interno del quale verrà richiamato nuovamente il finto compilatore per analizzare la porzione di codice inserita all'interno delle parentesi graffe dall'utente, le quali a loro volta verranno gestite con un approccio simile.

Il giocatore potrà quindi usare questo finto compilatore, che chiameremo **ScriptConsole** o semplicemente **console**, per interagire con il mondo di gioco. Verranno infatti predisposti degli oggetti nella mappa capaci di essere controllati attraverso questa console. Ad esempio:

- Il giocatore muovendosi nella mappa di gioco si avvicina ad un oggetto interagibile;
- Quando si trova abbastanza vicino, l'oggetto viene selezionato e viene segnalata la cosa al giocatore applicando un contorno bianco sopra l'oggetto;
- A questo punto, con la pressione di un apposito pulsante sulla tastiera, è possibile aprire la console;

Arrivati a questo punto il giocatore può scrivere il proprio codice per interagire con l'oggetto selezionato. Gli oggetti interagibili avranno proprietà e metodi richiamabili attraverso la ScriptConsole, sarà quindi possibile richiamare uno dei metodi disponibili oppure eseguire operazioni di assegnazione sulle proprietà. Ad esempio la quasi totalità degli oggetti interagibili avrà il metodo **Action()**, ovvero l'azione eseguita interagendo normalmente con l'oggetto attraverso il pulsante di interazione della tastiera. Sarà quindi possibile attivare il metodo Action() premendo il pulsante oppure richiamando il metodo attraverso la ScriptConsole.

Dato che l'obiettivo è formare futuri programmatori, utilizzare queste dinamiche del videogioco per abituare gli studenti ai design pattern più comuni è qualcosa che può rilevarsi molto utile. Proprio per questo ho inserito alcuni oggetti che cercano di replicare il meccanismo dei **Gestori Evento**.

Un gestore evento è un componente che resta in attesa che si verifichi un evento specifico e quando accade invia un segnale a tutti gli eventi che erano stati sottoscritti ad esso, attivandoli. Ad esempio un pulsante (gestore evento) e l'azione che si attiva dopo la pressione dello stesso(evento).

Per ricreare questa dinamica utilizzo 2 proprietà differenti:

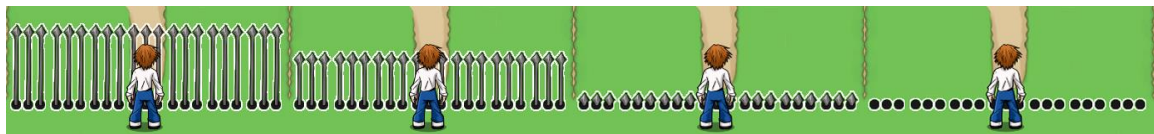
- **UniqueName**: un nome unico che identifica un oggetto specifico nella mappa e funziona come un riferimento ad esso;
- **RegisteredObj**: lo uniqueName di un altro oggetto sul quale viene perpretata l'azione dell'oggetto selezionato ( metodo Action() );

In questo modo il giocatore potrà ad esempio assegnare ad un cancello chiuso lo UniqueName "cancello" e successivamente assegnarlo al RegisteredObj di una leva. In questo modo tirando la leva(gestore evento), il cancello si aprirà(evento sottoscritto).

Ovviamente sarà possibile usare la console anche per altre operazioni, anche se non tutte saranno veramente utili. Ad esempio per gli oggetti che dispongono di una animazione, come il cancello o la leva, sarà presente anche la proprietà **Frame** che indica il frame dell'animazione (cancello aperto e cancello chiuso sono 2 frame diversi di un'unica animazione), sarà possibile quindi modificare questa proprietà ed assegnare frame diversi all'oggetto vedendo i vari stadi dell'animazione.

Sarà inoltre sempre presente il metodo `Wait()`, indipendentemente dall'oggetto selezionato, che permetterà di attendere una certa quantità di secondi prima di eseguire l'istruzione successiva.

Combinando quindi il `for`, con il `Wait()` e l'assegnazione della proprietà `Frame` sarà possibile variare il valore della proprietà da un minimo ad un massimo con un tempo di attesa da una assegnazione all'altra, andando di fatto a creare una animazione visibile.



*Frame = 0;*

*Frame = 1;*

*Frame = 2;*

*Frame = 3;*

Questa è una operazione possibile anche se inutile, infatti anche cambiando il frame del cancello da chiuso ad aperto questo non renderà possibile attraversarlo. Con questa modifica il giocatore ha solo cambiato forzatamente il frame di un oggetto, ma non la sua logica interna. In questo caso infatti, il **Collider**, ovvero il componente che gestisce le collisioni, non è stato modificato e il gioco si comporta quindi come se il cancello fosse ancora chiuso.

Anche questa situazione è voluta, proprio per far capire all'utente che c'è differenza dalla logica reale e quella di un videogioco o di una qualsiasi applicazione. Nella realtà se una porta è aperta allora è possibile attraversarla, ma nel videogioco quella è solo una immagine e il modo in cui si comporta dipende da altri parametri che vanno manipolati attraverso il codice.

Pertanto è chiaro che questa operazione sia inutile in pratica, tuttavia, oltre alle motivazione espresse sopra, credo sia importante permettere al giocatore di sperimentare qualcosa che di per sé è inutile, ma funziona. A volte è proprio con queste piccole cose, con delle intuizioni che non vengono suggerite dal software ma che funzionano, anche se solo graficamente in questo caso, che può nascere la soddisfazione di sperimentare, andare oltre e ragionare fuori dagli schemi, magari abbastanza dal desiderare di esercitarsi con un compilatore reale che possa offrire più libertà e possibilità.

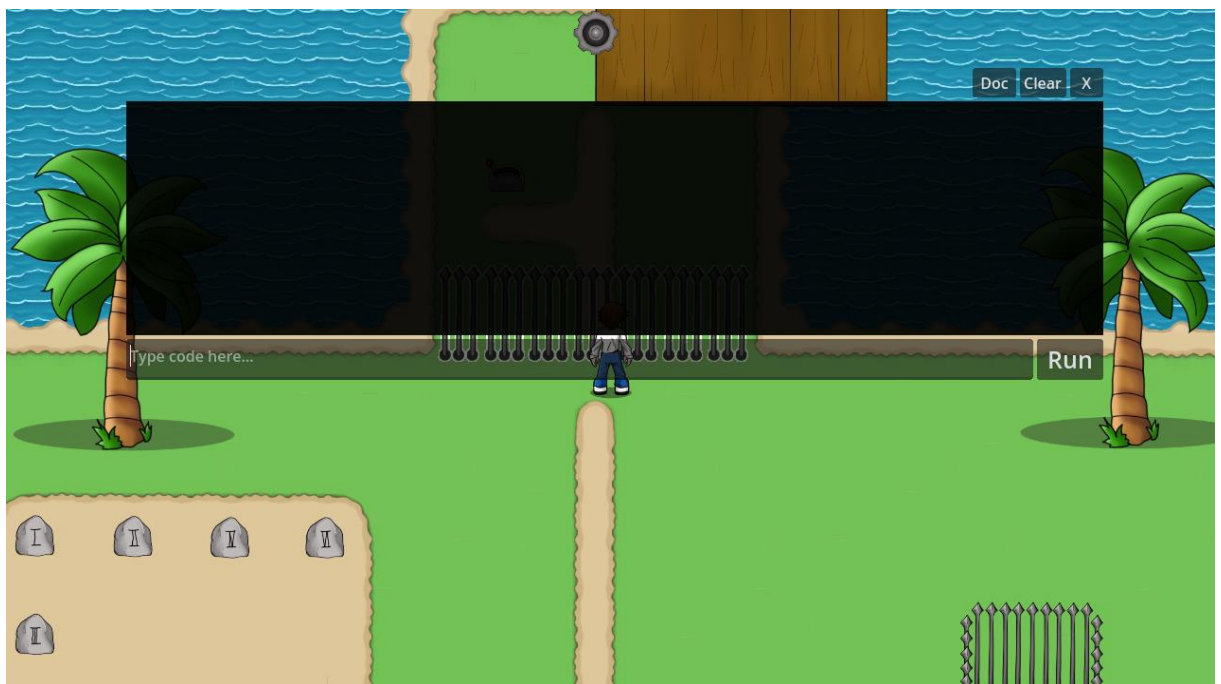
## 2.4 Delineare le Meccaniche di Gioco: Doc

Una volta delineate le modalità di gioco legate alla ScriptConsole si è subito fatto evidente un problema: *“Come fa il giocatore a conoscere metodi e proprietà di un oggetto? Come fa a sapere quali comandi sono stati implementati nella ScriptConsole?”*

Ovviamente questo è un problema importante e per la sua risoluzione ho deciso di provare a replicare all’interno del gioco un modo di fare che fosse quanto più vicino a quello reale.

Nella realtà infatti, quando si sviluppa in un linguaggio nuovo oppure utilizzando framework, plugin, librerie o engine sconosciuti, quello che si fa è consultare la documentazione. Io stesso approcciandomi allo sviluppo con Godot ho spesso dovuto consultare la documentazione per capire come gestire determinate situazioni.

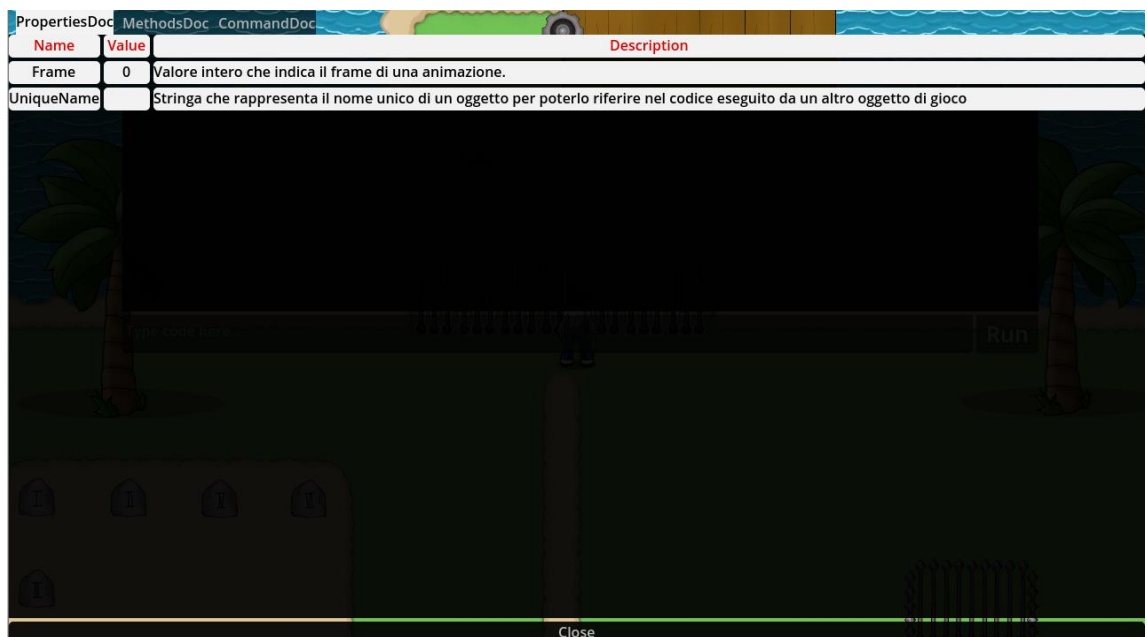
Nel gioco quindi, una volta aperta la ScriptConsole, apparirà un pulsante chiamato “Doc” grazie al quale verrà aperta una nuova vista contenente una tabella riempita con i dati relativi all’oggetto selezionato.



*ScreenShot di una schermata di gioco con la ScriptConsole a schermo*

La tabella avrà tre tab:

1. **PropertiesDoc**: contiene le informazioni riguardanti le proprietà dell'oggetto (nome, valore e descrizione);
2. **MethodsDoc**: contiene le informazioni riguardanti i metodi dell'oggetto (nome e descrizione sull'utilizzo);
3. **CommandDoc**: contiene informazioni riguardanti i comandi del linguaggio implementati nella ScriptConsole (sintassi e descrizione di if e for);



*ScreenShot della schermata Doc in gioco*

In questo modo il giocatore potrà subito capire come interagire con i vari oggetti e ragionare sulle scelte algoritmiche da adottare per risolvere i vari puzzle e proseguire nel gioco.

Inoltre abituandosi a ragionare in questi termini per approcciarsi ad un contesto nuovo, sarà più naturale, quando si troverà in una situazione reale, leggere la documentazione e informarsi per procedere con lo sviluppo in modo più efficace.

## 2.5 Delineare le Meccaniche di Gioco: Modalità Esercizio

A questo punto le meccaniche di gioco sono abbastanza delineate. Tuttavia al momento non sono presenti meccaniche che possano essere particolarmente

utili ai docenti, inoltre danno tutte per scontato che si abbia già un minimo di confidenza con la sintassi del C# e con le operazioni di assegnazione e chiamata dei metodi.

Per questo motivo ho deciso di inserire un'altra meccanica che potesse essere utile per introdurre i neofiti alle dinamiche sopra esplicitate.

Pertanto ho introdotto una modalità in cui il giocatore si trova di fronte uno script C# parzialmente scritto, da completare.

In questa vista sono presenti 3 schermate che dividono verticalmente lo schermo: una centrale e due laterali.

All'interno della schermata centrale sarà presente lo script da completare, a sinistra saranno presenti dei box, che chiameremo **Spawner**, contenenti parti di codice da usare per completare lo script e a destra una descrizione dello script che spieghi quale deve essere la logica da implementare. L'utente dovrà quindi trascinare gli Spawner contenenti il codice nelle zone vuote dello script e, una volta fatto, cliccare su di un pulsante di check posizionato nella parte inferiore dello schermo. A questo punto verrà visualizzato in un box di testo il risultato dell'operazione(corretto oppure errato).

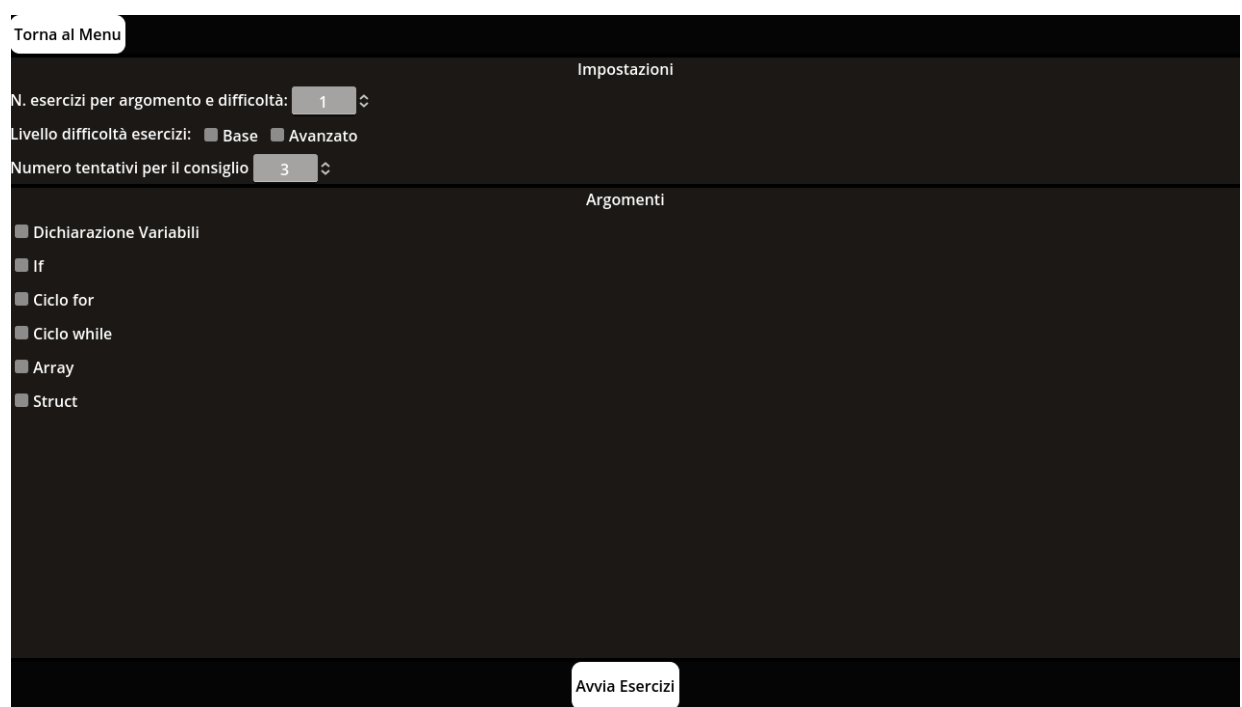
In questo modo gli studenti possono prendere confidenza con il codice senza doverlo scrivere o ricordare, ma semplicemente ragionando sulla logica algoritmica.

Ovviamente per proseguire nel gioco è necessario risolvere questi *“esercizi”*, per fornire un ulteriore aiuto agli studenti che potrebbero avere problemi: in caso di diversi errori verrà mostrato nel box di testo un suggerimento che sarà specifico sulle parti di codice completate in modo errato. In questo modo lo studente avrà modo di capire lo sbaglio, ragionare sul suggerimento ricevuto e arrivare in autonomia alla soluzione corretta. Questo servirà proprio per cercare di far comprendere meglio le dinamiche sulle quali si avevano dei dubbi portando così a migliorare le competenze di programmazione dello studente.

Una volta delineata questa meccanica, ci si è resi conto che poteva essere molto utile anche al di fuori del gioco, in modo da poterla utilizzare come un vero e proprio eserciziaro.

Così abbiamo deciso di realizzare una intera modalità di gioco, separata dalla campagna, chiamata **Modalità Esercizio** in cui l'utente può selezionare:

- **Argomenti:** Dichiarazione Variabili, if, for, while, Array, Struct;
- **Difficoltà:** Base, Avanzato;
- **Numero di esercizi:** quanti esercizi bisognerà risolvere per ogni livello di difficoltà di un argomento scelto;
- **Numero tentativi per suggerimento:** quanti errori bisogna fare prima di ricevere un suggerimento;



*Schermata di configurazione Modalità esercizio*

Una volta impostati i dati attraverso la Ui sarà possibile avviare gli esercizi. Veranno scelti gli esercizi in modo casuale e verranno riprodotti in ordine di argomento e difficoltà, ovviamente senza ripetere più volte lo stesso.

Durante la risoluzione degli esercizi verrà avviato un timer con il quale verrà tenuto conto del tempo impiegato per la risoluzione degli stessi.

Alla fine di tutti gli esercizi verrà mostrata una pagina di riepilogo all'interno della quale sarà possibile visionare tempo impiegato ed errori commessi per singola categoria e in totale.

In questo modo, dato che lo studente potrebbe semplicemente provare tutte le combinazioni possibili per andare avanti, il professore controllando il riepilogo della prestazione può farsi un'idea di quelle che sono le lacune e i punti di forza dei singoli studenti.

1 / 3
00:00:03
Torna al Menu


string
float
int

☐ direction  
☐ time

```

private IEnumerator MakeAnimation ()
{
    time = 0.12f;
    direction = GetStringDirection ();
    ☐ firstFrame = GetDirectionFirstFrame ();
    ☐ lastFrame = GetDirectionLastFrame ();
    for( ☐ i = 0; i < firstFrame; i < lastFrame; i++)
    {
        this.Frame = i;
        yield return Timing.WaitForSecond (time);
    }
    MakeAnimation (firstFrame, lastFrame);
}

```



**Traccia**

Questo script gestisce le animazioni 2D del tuo avatar. Dichiarare le variabili in modo consono alla logica implementativa.

Check
Next

### Esempio esercizio

Categoria	Errori	Tempo
Dichiarazione Variabili	0	00:00:41
If	1	00:00:32
Ciclo for	2	00:01:51
<b>Totale</b>	3	00:03:05

Torna al Menu

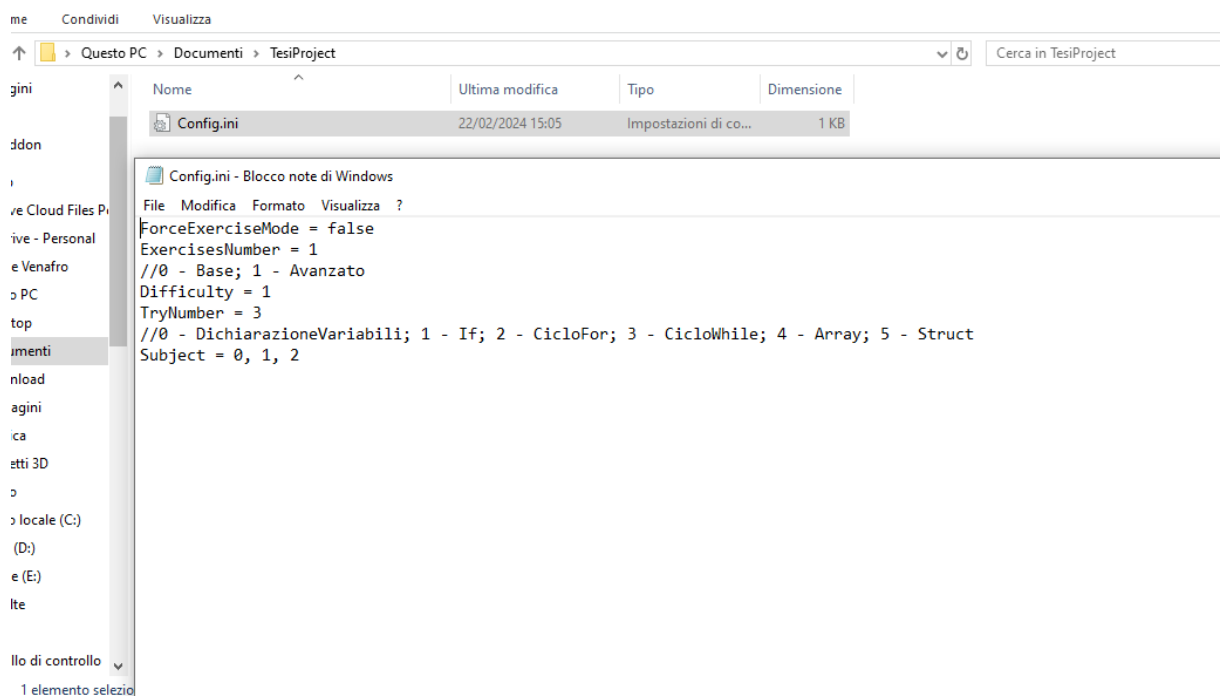
### Esempio riepilogo prestazioni dello studente



Ovviamente il tutto è stato realizzato in modo tale che attraverso l'editor di Godot sia possibile creare nuovi esercizi da aggiungere al gioco in modo comodo e soprattutto senza la necessità di scrivere codice ulteriore. Servirà solo prendere confidenza con l'editor dell'engine e utilizzare i componenti da me precedentemente preparati.

La modalità esercizio, inoltre, è possibile configurarla anche esternamente.

Gli studenti infatti, soprattutto delle scuole superiori o precedenti, non sempre sono collaborativi e potrebbero non entrare nella modalità esercizio o non configurarla correttamente. Per questo motivo, al primo avvio dell'applicazione verrà creato un file ".ini" nella cartella documenti del pc, all'interno di questo file sarà possibile configurare tutti i dati necessari all'avvio della modalità esercizio e sarà possibile anche abilitare l'apertura forzata della stessa. In questo modo, settando anticipatamente il file di configurazione nelle postazioni pc usate dagli studenti, avviando l'applicazione si verrà reindirizzati direttamente alla modalità esercizio avviata con i dati precedentemente configurati.



*File .ini per la configurazione esterna della Modalità Esercizio*

## 2.6 Delineare le Meccaniche di Gioco: Broker MQTT

Come già detto più volte ormai, uno degli obbiettivi del progetto di tesi è quello di avvicinare gli studenti alla programmazione facendo venir loro la voglia di scrivere codice e sperimentare.

Spesso uno dei primi approcci alla programmazione per gli studenti, soprattutto durante le scuole superiori, avviene grazie ad Arduino, caso di cui anche io sono un esempio.

Arduino è un microcontrollore programmabile con codice C++ e viene utilizzato in tantissime applicazioni diverse, una di queste è sicuramente la domotica. Ed è proprio in questi contesti che ad Arduino vengono spesso affiancati i Broker MQTT. Grazie ad essi è infatti possibile far comunicare Arduino con altri dispositivi capaci a loro volta di interagire con il Broker.

È possibile infatti definire un topic, ovvero un argomento, e un messaggio da inviare al Broker, in questo modo tutti i dispositivi collegati a quest'ultimo, e che hanno sottoscritto lo stesso topic, possono leggere quel messaggio e agire di conseguenza.

Il gioco quindi sarà in grado di inviare dei messaggi ad un broker precedentemente configurato e gli studenti potranno, ad esempio, creare degli script su Arduino capaci di leggere questi messaggi ed eseguire azioni in risposta.

Gli studenti potranno quindi divertirsi a creare ogni tipo di interazione tra il gioco ed il microcontrollore, sperimentando e migliorando sempre di più le loro abilità di coding.

Inoltre la programmazione con Arduino può essere un buona scuola anche per lo sviluppo di script per un Game Engine come Godot. Infatti, non solo C++ e C# hanno una sintassi simile, ma lo script di un videogioco è molto simile ad uno script di Arduino.

In uno script di Arduino abbiamo infatti principalmente 2 metodi:

- **Setup:** metodo che viene eseguito solo una volta all'accensione della scheda;
- **Loop:** codice che viene costantemente ripetuto durante tutto il periodo in cui la scheda resta operativa;

La stessa cosa avviene per gli script in Godot, l'unica differenza è che gli script vengono associati ad un determinato GameObject e vengono eseguiti in relazione al ciclo di vita di quel singolo oggetto e non quello dell'applicazione in generale. Imparare a ragionare con Arduino può essere quindi un grosso aiuto per trovarsi più preparati quando ci si ritroverà a sviluppare con un Game Engine, anche se ovviamente sono presenti molte altre differenze e complessità da affrontare.

## 2.7 Concept della Storia

Il gioco avrà quindi una progressione a livelli, all'interno della quale sarà presente anche una leggera narrazione che accompagnerà il giocatore durante l'avanzamento. Il concept della storia è il seguente:

*“Il protagonista è uno studente che si trova a vivere una esperienza virtuale Full Dive in un videogioco, sotto la guida di un suo professore (Con esperienza Full dive si intende una simulazione virtuale in cui l'utente può controllare il proprio avatar digitale con il pensiero e percepire quello che lo circonda attraverso i cinque sensi, come se fosse il suo vero corpo. È una tecnologia che attualmente non esiste ma che viene spesso usata come espediente narrativo in molti giochi e altre opere di intrattenimento).*

All'interno di questa esperienza virtuale il protagonista dovrà, sotto la guida del professore, risolvere tutti i problemi dell'applicazione. Il professore gli spiegherà che attraverso questo sistema è possibile effettuare operazioni di debug internamente al gioco, risolvendo i problemi nel mentre si verificano.

L'obiettivo sarà quindi proseguire nei livelli, risolvere tutti i problemi e scoprire cos'è che sta generando tutte queste situazioni inattese all'interno del software.”

## 2.8 Menu e Opzioni

Oltre a quanto detto il gioco necessita di un menu principale che permetta di muoversi tra le varie modalità di gioco e un menu di opzioni dove configurare le varie impostazioni.

In particolare il Menu dovrà contenere le voci:

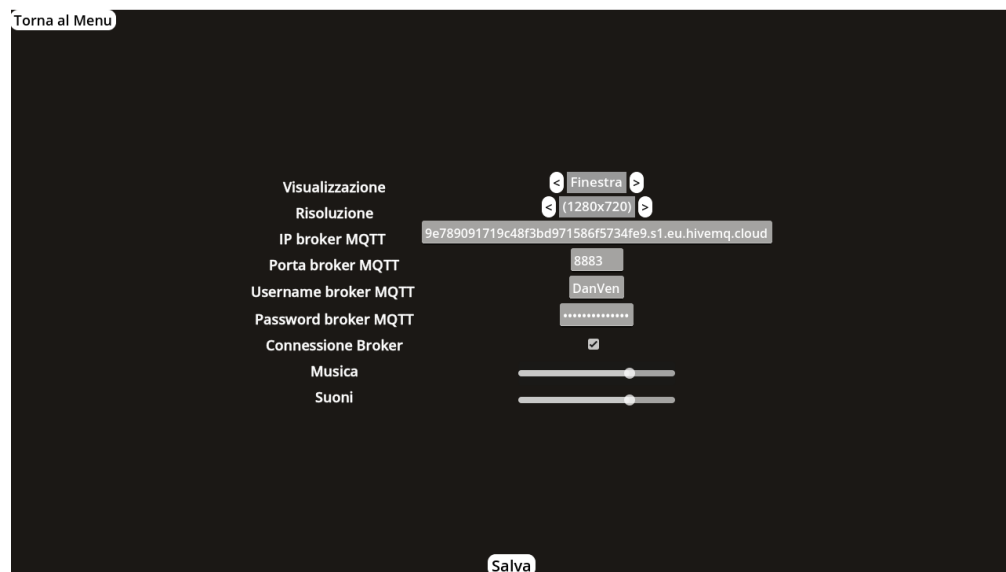
- **Nuovo gioco:** Cominciare una nuova partita dal primo livello;
- **Seleziona livello:** Giocare un livello precedentemente sbloccato;
- **Modalità esercizio:** Accedere alla pagina di configurazione della stessa;
- **Opzioni:** Accedere alla vista per modificare le impostazioni;
- **Esci dal gioco:** Termina l'applicazione;



*ScreenShot Menu iniziale del gioco*

Nella vista delle opzioni saranno presenti, invece, le voci:

- **Visualizzazione:** Finestra, finestra senza bordi, schermo intero;
- **Risoluzione:** Per adattare la visualizzazione del gioco a diversi schermi;
- **Broker:** Ip, password, credenziali e abilitazione dello stesso;
- **Audio:** Adattare il livello audio di musica ed effetti sonori;

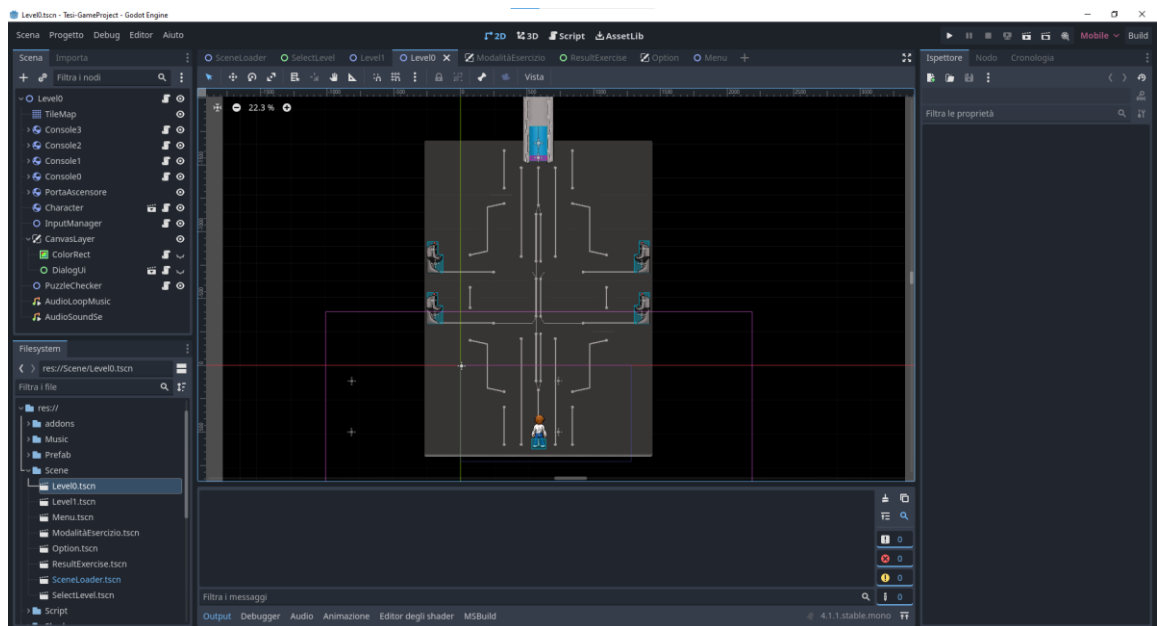


*ScreenShot Menu Opzioni del gioco*

## 3 Introduzione a Godot Game Engine

### 3.1 Interfaccia

Prima di dare uno sguardo al codice è meglio introdurre le basi del game engine utilizzato, ovvero Godot.



*Schermata dell'editor Godot una volta aperto il progetto*

Nella parte centrale troviamo l'ambiente virtuale che caratterizza la **Scena** corrente, ovvero tutta la parte grafica che sarà poi visibile in partita.

Nella parte sinistra troviamo 2 finestre:

1. **La finestra in alto** mostra tutti gli elementi presenti nella scena. La scena è la lista degli elementi che sono presenti nel progetto corrente e sono organizzati secondo una struttura ad albero. Ad esempio la scena "Level0" contiene la mappa di gioco, l'avatar del giocatore, gli elementi interagibili e gli elementi di interfaccia grafica. Tutti questi elementi sono chiamati **nodi**. Godot contiene diversi nodi preconfigurati utili per aggiungere funzionalità alla scena. Ogni nodo "specializzato" è figlio di un nodo generico da cui eredita i metodi e le proprietà di base.
2. **La finestra in basso** contiene un esplora risorse puntato alla directory del progetto, in modo da tenere sotto controllo tutti i file presenti ed eventualmente aggiungerne di nuovi oppure eseguire operazioni di eliminazione, modifica, duplicazione o spostamento degli stessi.

Nella parte a destra dello schermo è presente l'**inspector** che permette, una volta selezionato un nodo nella finestra a sinistra, di vederne i dettagli ed eventualmente modificare alcune delle sue proprietà.

Nella parte in basso è invece presente l'**output**, da cui è possibile leggere tutte le informazioni relative a problemi o errori di compilazione o configurazione dei nodi, oppure eventuali messaggi di logging da noi inseriti nel codice di gioco.

### 3.2 Nodi

Esistono tanti tipi di nodi: nodi 3d, nodi 2d, nodi control (UI) e tanti altri. Ogni nodo può essere configurato nell'**inspector**, ma è anche possibili "estenderli" scrivendo degli script che vanno ad aggiungere funzionalità alla classe del nodo stesso.

Ad esempio per creare l'avatar del personaggio giocante è stato usato:

1. un nodo **RigidBody2D** come nodo padre di tutti gli elementi. Questo nodo è realizzato per essere influenzato dalla fisica dell'ambiente virtuale.
2. Un nodo **Sprite2D** come primo figlio, un nodo a cui è possibile assegnare una spritesheet. Attraverso i suoi parametri è possibile ritagliare l'immagine e visualizzare uno solo dei frame in esso contenuti.
3. Un nodo **CollisionShape2D**, che gestisce le collisioni del RigidBody2D. Attraverso questo nodo è possibile assegnare una forma al RigidBody2D in modo tale da generare una collisione se quella forma dovesse toccare altri elementi influenzati dalla fisica.

In questo modo grazie al RigidBody2D e al CollisionShape2D è possibile muovere l'avatar del personaggio che verrà bloccato in automatico nel caso dovesse sbattere contro altri elementi fisici.

Tuttavia al momento non è possibile controllare il personaggio per muoverlo nella scena o cambiare il suo Frame per generare l'animazione, se non andando a modificare manualmente i suoi parametri all'interno dell'**inspector**.

Per poter eseguire questo tipo di operazioni è necessario scrivere uno script, in questo caso in C#, da associare al nodo che implementi la logica necessaria.

### 3.3 Script

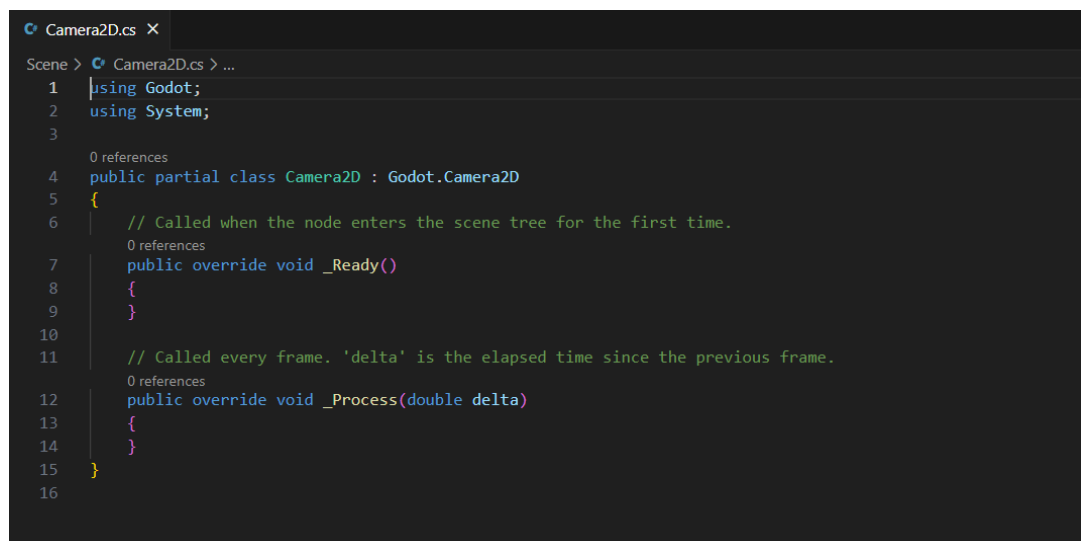
È possibile assegnare uno script a qualsiasi nodo, per farlo basta:

- cliccare sul nodo desiderato;
- scorrere in fondo l'inspector fino alla voce script;
- cliccare aggiungi nuovo script oppure inserirne uno già scritto in precedenza;

Gli script non sono altro che classi C# che utilizzano le librerie di Godot e possono estendere i tipi di base dell'engine, ovvero i nodi.

I nodi infatti non sono altro che classi, pertanto aggiungendo uno script al RigidBody2D verrà creato in automatico uno script che estenderà la classe RigidBody2D.

Ad esempio creando uno script per il nodo Camera2D verrà creato in automatico uno script con lo stesso nome che estende Camera2D e utilizza la libreria System di C# e quella di Godot.

A screenshot of a code editor showing a C# script named 'Camera2D.cs'. The script is a partial class that inherits from 'Godot.Camera2D'. It includes 'using' statements for 'Godot' and 'System'. It defines two methods: '\_Ready()' which is called when the node enters the scene tree, and '\_Process(double delta)' which is called every frame. The code is as follows:

```
1 using Godot;
2 using System;
3
4 public partial class Camera2D : Godot.Camera2D
5 {
6     // Called when the node enters the scene tree for the first time.
7     public override void _Ready()
8     {
9     }
10
11     // Called every frame. 'delta' is the elapsed time since the previous frame.
12     public override void _Process(double delta)
13     {
14     }
15 }
16
```

*Script Godot in C# appena creato*

Verranno inoltre inseriti 2 metodi di default:

1. **\_Ready()**: metodo di inizializzazione, viene eseguito una sola volta all'avvio della scena.
2. **\_Process(double delta)**: metodo che viene eseguito ad ogni frame in Loop. "delta" è una variabile numerica che indica il tempo trascorso tra un frame e l'altro.

È possibile notare, come detto in precedenza, che il funzionamento è molto simile a quello di uno script di Arduino.

In questo caso, con frame si intende il lasso di tempo necessario alla macchina per eseguire una volta tutti gli script dei nodi presenti nella scena.

Quindi quanto più le operazioni da svolgere sono complesse, più tempo sarà necessario per eseguire tutti gli script. Di conseguenza la durata di un frame aumenta e questo vuol dire minor numero di frame che possono essere contenuti in un secondo.

È da questo che viene quindi la parola **framerate** o **fps**(frame per seconds).

Non è quindi possibile sapere quante volte in un secondo verrà eseguito il codice presente nel `_Process()`, pertanto bisogna prestare attenzione al codice che si scrive.

Ad esempio se su una macchina il gioco gira a 30 fps, il codice verrà eseguito 30 volte nell'arco di un secondo, mentre su una macchina su cui gira a 60 fps verrebbe eseguito 60 volte.

Ipotizzando di voler scrivere uno script che sposta un oggetto nel mondo di gioco di un offset di 2 unità sull'asse X all'interno del `_Process()`: nel primo caso l'oggetto si sposterà di 60 unità nell'arco di un secondo, nel secondo si sposterà di ben 120 unità.

Ovviamente non può essere che la velocità di spostamento di un oggetto dipenda dal framerate a cui gira il gioco, per risolvere possono esserci varie metodologie:

1. **Moltiplicare l'offset per la variabile delta**, in questo modo l'offset sarà maggiore o minore in base al framerate dell'applicazione. L'oggetto si muoverà quindi sempre alla stessa velocità dato che, in caso di framerate elevato, delta sarà molto piccolo e quindi il valore di offset sarà minore. Se il framerate fosse basso, il valore di delta aumenterebbe andando così ad incrementare anche l'offset.
2. **Non eseguire l'operazione nel `_Process()`**, che ha un tasso di esecuzione dubbio, ma nel **`_PhysicsProcess()`**. Anche questo metodo viene richiamato in loop, ma viene usato per gestire l'aggiornamento della fisica del gioco. A differenza del `_Process()` la fisica viene aggiornata indipendentemente dal framerate e il valore di delta è perciò costante, quindi verrà eseguito sempre lo stesso numero di volte in un secondo. Tuttavia lo spostamento potrebbe non essere fluido, dato che questo metodo viene eseguito poche volte in un secondo.



3. **Eseguire operazioni di interpolazione**, in questo modo verrà calcolata la posizione avendo come dato la posizione iniziale, quella finale e il peso. In questo modo la posizione verrà aggiornata ad ogni frame, ma lo spostamento effettuato non dipenderà più dal frame, ma dal tempo trascorso.
4. **Usare i metodi predisposti in Godot** per gestire il movimento di un RigidBody. È possibile usare il metodo di RigidBody2d **MoveAndCollide(Vector2 motion)** per muovere il RigidBody nella direzione e verso del vettore motion di una quantità pari al suo modulo. In questo modo non solo verrà mosso, ma reagirà anche alle collisioni con altri oggetti. Anche qui resta il problema legato ai frame, per ovviarlo basterà moltiplicare il vettore per la variabile delta, in modo da ridimensionare il modulo in base al framerate. L'immagine sotto mostra un esempio:

```
1 reference
public override void _Process(double delta)
{
    Vector2 direction = GetDirectionVector();
    this.MoveAndCollide(direction * ((float) (550 * delta)));
}
```

*Esempio di Applicazione del metodo MoveAndCollide*

Queste sono solo alcuni esempi e linee guida su come scrivere degli script in C# per Godot. Per ulteriori approfondimenti è possibile visionare la documentazione online dell'engine al seguente indirizzo:

<https://readthedocs.org/projects/godot/>

## 4 Uno Sguardo alla Logica Implementativa

### 4.1 Addon Godot

All'interno del progetto ho utilizzato 2 addon specifici per la piattaforma di godot:

1. **GDMEC**: Libreria disponibile su GitHub. Permette di utilizzare le coroutine in modo simile a Unity, risolvendo i problemi di “*garbage generation*” e aggiungendo ulteriori metodi per avere maggior controllo sulle coroutine;
2. **PlayerPrefs C#**: Permette di adottare uno stile di programmazioni simile a Unity, permettendo di avere accesso ad un dizionario Chiave-Valore dove è possibile salvare dati in modo persistente. Utile ad esempio per salvare i settaggi del menu Opzioni. Si trova disponibile al download su Godot Asset Library;

### 4.2 Controllo Personaggio e Camera

Per gestire il movimento dell'avatar del personaggio ho usato tre diversi nodi:

- **RigidBody2D**: Nodo influenzabile dalla fisica. È possibile quindi applicarci delle forze per gestirne il comportamento e le collisioni. A questo nodo ho applicato uno script personalizzato che ho chiamato **CharMovementControl**;
- **Sprite2D**: Contiene la spritesheet del personaggio con i vari frame dell'animazione. A questo nodo ho associato un altro script chiamato **CharController**;
- **CollisionShape2D**: Nodo che definisce la forma di un oggetto sulla quale basare le sue collisioni ;

Il nodo principale è il RigidBody, gli altri sono suoi nodi figli.

Lo script **CharController** rileva gli input legati al movimento, ovvero i pulsanti “wasd” della tastiera. Premendo uno di questi pulsanti, viene salvata la direzione selezionata in una stringa **Direction**.

In base al valore salvato in questa variabile viene riprodotta l'animazione associata al movimento in quella direzione.

```
2 references
private void AnimationControl()
{
    CheckInput();
    if (IsDirectionChanged() && isAnimationEnabled)
    {
        if (Direction == Costanti.DirectionLeft || Direction == Costanti.DirectionLeftUp || Direction == Costanti.DirectionLeftDown)
        {
            isAnimating = true;
            myCoroutine = Timing.RunCoroutine(MakeAnimation(4, 7));
        }
        else if (Direction == Costanti.DirectionRight || Direction == Costanti.DirectionRightUp || Direction == Costanti.DirectionRightDown)
        {
            isAnimating = true;
            myCoroutine = Timing.RunCoroutine(MakeAnimation(8, 11));
        }
        else if (Direction == Costanti.DirectionUp)
        {
            isAnimating = true;
            myCoroutine = Timing.RunCoroutine(MakeAnimation(12, 15));
        }
        else if (Direction == Costanti.DirectionDown)
        {
            isAnimating = true;
            myCoroutine = Timing.RunCoroutine(MakeAnimation(0, 3));
        }
        currentDir = Direction;
    }
}
```

*Metodo AnimationControl della classe CharController*

L'animazione è realizzata attraverso le coroutine:

- In base alla direzione vengono individuati i coefficienti di inizio e fine della animazione;
- Utilizzo un for per incrementare un contatore dal coefficiente di inizio a quello di fine;
- A ogni step del for viene cambiato il valore del Frame dello Sprite e successivamente viene utilizzato il comando **yield return** per mettere in sospensione l'esecuzione del metodo per una certa durata in secondi;

```
5 references
private IEnumerator<double> MakeAnimation(int coefficienteStart, int coefficienteEnd)
{
    float time = 0.12f;
    for (int i = coefficienteStart; i < coefficienteEnd; i++)
    {
        this.Frame = i;
        yield return Timing.WaitForSeconds(time);
    }
    myCoroutine = Timing.RunCoroutine(this.MakeAnimation(coefficienteStart, coefficienteEnd), Costanti.PlayerAnimationTag);
}
```

*Metodo MakeAnimation per la riproduzione delle animazioni dell'avatar*

In questo modo il flusso di esecuzione del metodo viene separato dal `_Process()` e cambiando il frame con il giusto tempismo si ottiene l'animazione. Fatto questo il metodo richiama se stesso per permettere

all'animazione di essere eseguita in loop fintanto che il pulsante di input è premuto.

Lo script **CharMovementControl** ha un riferimento al CharController, in questo modo legge qual è la direzione attuale selezionata e applica al Rigidbody una forza nella direzione corrispondente, utilizzando il metodo MoveAndCollide visto nello scorso capitolo.

```
2 references
private Vector2 GetDirectionVector(){
    string direction = charController.Direction;
    if (direction.Equals(Costanti.DirectionLeft))
    {
        return Vector2.Left;
    }
    if (direction.Equals(Costanti.DirectionRight))
    {
        return Vector2.Right;
    }
    if (direction.Equals(Costanti.DirectionUp))
    {
        return Vector2.Up;
    }
    if (direction.Equals(Costanti.DirectionDown))
    {
        return Vector2.Down;
    }
    if (direction.Equals(Costanti.DirectionLeftUp))
    {
        return upLeft.Normalized();
    }
    if (direction.Equals(Costanti.DirectionLeftDown))
    {
        return downLeft.Normalized();
    }
    if (direction.Equals(Costanti.DirectionRightUp))
    {
        return upRight.Normalized();
    }
    if (direction.Equals(Costanti.DirectionRightDown))
    {
        return downRight.Normalized();
    }
    return Vector2.Zero;
}
```

*Metodo GetDirectionVector della classe CharMovementControl*

Per permettere la visualizzazione della scena ho inserito un nodo Camera2D puntato in modo da avere lo sprite del personaggio al centro dello schermo.

Il nodo è stato inserito come figlio del Rigidbody, in modo tale che muovendo il personaggio nella mappa lo stesso movimento venga applicato alla camera e gli altri nodi figli. In questo modo la camera seguirà sempre il giocatore.

### 4.3 Classi Statiche

Per semplificare la scrittura, ed evitare inutili ripetizioni del codice, ho realizzato delle classi statiche che implementassero del codice di utilizzo comune:

1. **Static\_NodeMethod**: classe contenente metodi utili alla ricerca di nodi nella scena. Godot utilizza una struttura ad albero, quindi è semplice passare da un nodo all'altro. Ad esempio, fornendo un nodo da cui cominciare la ricerca, è possibile cercare i nodi figli o padre con una classe o un nome specifico oppure duplicare uno o più nodi.
2. **Static\_AnimationMethod**: permette di creare una animazione, in modo simile a quanto visto sopra: dando un riferimento allo Sprite, coefficienti di inizio e fine ed eventualmente il tempo di attesa da un frame all'altro.
3. **Static\_DisplayControl**: contiene metodi utili per modificare le opzioni di visualizzazione dell'applicazione come la risoluzione;
4. **Static\_AudioControl**: permette di modificare il volume dei vari bus audio utilizzati dall'applicazione. È presente un bus **Music** che gestisce l'audio della musica di sottofondo riprodotta nei livelli, un bus **Se** che gestisce l'audio dei singoli effetti sonori e un bus **Master** che controlla tutti gli altri contemporaneamente.
5. **Static\_BrokerConnection**: utilizza la libreria HiveMQtt in C# per gestire la comunicazione con i Broker MQTT. È presente un metodo pubblico a cui è possibile specificare il topic e il messaggio da inviare, la classe statica provvederà ad estrapolare i dati per la connessione al broker dalle PlayerPrefs, creare il client, la connessione e successivamente ad inviare il messaggio in automatico. È necessario fornire anche un riferimento ad un oggetto che implementa l'interfaccia BrokerChatter, da me realizzata. Questo serve in caso di errore nel processo di invio del messaggio per richiamare un metodo di gestione dell'errore che dovrà essere implementato all'interno dell'oggetto BrokerChatter. A questo metodo verrà fornito il messaggio dell'eventuale eccezione o guardia sollecitata nel processo di connessione al broker.
6. **Static\_PopupWindowSpawner**: permette di creare finestre di popup. Utilizzando i vari metodi pubblici è possibile aggiungere delle label, impostare la finestra come persistente o a tempo ed aggiungere un

eventuale pulsante di chiusura della stessa. In questo modo è possibile creare una finestra per visualizzare messaggi con un pulsante di conferma oppure una notifica che mostra informazioni e sparisce autonomamente, il tutto senza dover creare manualmente la finestra attraverso l'editor grafico.

#### **4.4 Modalità Esercizio**

Come detto nel capitolo 2, la modalità esercizio è uno strumento rivolto più agli insegnanti che possono utilizzarla come eserciziaro per gli studenti. All'interno dell'applicazione sono stati inseriti come argomenti possibili i seguenti:

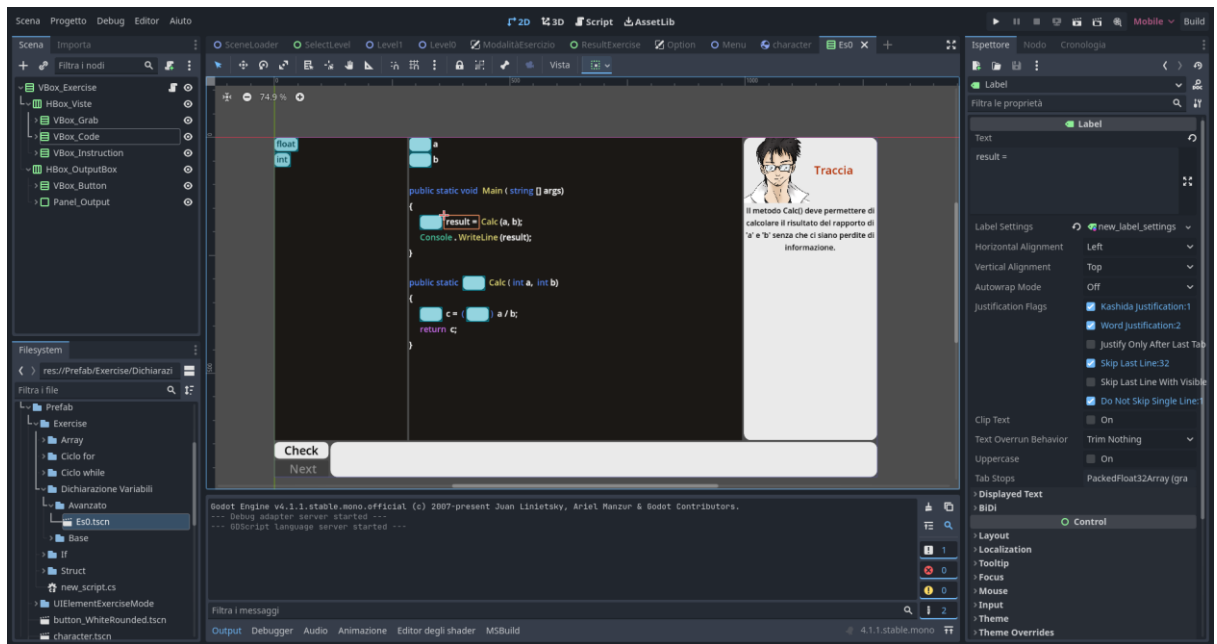
- Dichiarazione Variabili;
- If;
- Ciclo for;
- Ciclo while;
- Array;
- Struct;

Per ognuno di questi argomenti è stato creato un esercizio di livello Base e uno Avanzato.

Utilizzando l'editor dell'engine è possibile creare nuovi esercizi, per farlo basterà andare nel percorso "*<CartellaProgetto>\Prefab\Exercise*", qui sarà presente una cartella per ogni categoria al cui interno saranno presenti altre due cartelle chiamate Base ed Avanzato. A questo punto basterà scegliere una delle scene presenti in una delle due cartelle, duplicarla e successivamente aprirla per effettuare le modifiche.

Il codice di gioco infatti, quando deve caricare un esercizio, non fa altro che andare nel percorso corrispondente alla categoria e difficoltà scelte e caricare il numero specificato di esercizi scelti randomicamente.

Aperto un esercizio a caso ci si ritroverà di fronte ad una schermata simile.



*Scena di un esercizio aperta nell'editor di Godot*

A questo punto la prima cosa da fare sarà modificare il valore delle label presenti nella scena, in modo da costruire il corpo dello script da completare. Per farlo basterà selezionare una label nella scena con il mouse e successivamente modificarne il testo dall'inspector sulla destra.

In modo da rendere il testo più facilmente leggibile ho creato dei profili personalizzati utili per cambiare grandezza e colore del testo di una label, in modo replicare lo schema di colori utilizzati da Visual Studio Code per la visualizzazione di codice C#.

Per applicare un profilo alla label basterà cliccare su "Label Settings" dall'inspector e scegliere dalla lista dei profili quello adatto.

Le label sono organizzate all'interno di **Container**, componenti UI di Godot utili per visualizzare e posizionare altri elementi di interfaccia secondo una logica specifica. Ad esempio ogni riga del codice visualizzato nella finestra centrale è realizzata attraverso il nodo **HboxContainer**, che permette di visualizzare gli elementi al suo interno disposti orizzontalmente uno dopo l'altro. Ogni riga a sua volta è inserita all'interno di un **VboxContainer** che permette di disporre verticalmente tutte le righe al suo interno.

Volendo quindi inserire del testo in una riga, bisognerà inserire una label per ogni testo con colore diverso che si desidera visualizzare, assegnando ad ognuna il profilo corretto.

I box bianchi visibili all'interno dello script da completare sono chiamati **InteractiveBox** e rappresentano i punti da completare.

Gli InteractiveBox, così come gli esercizi, sono delle scene pre-configurate che è possibile istanziare all'interno dell'albero della scena. Come detto infatti una scena di Godot è organizzata secondo una struttura ad albero, pertanto è possibile selezionare una parte di questo albero e salvarlo come Scena. Allo stesso modo è possibile nidificare una scena all'interno di un'altra.

Ho inserito tutti questi elementi pre-configurati all'interno della cartella **Prefab** del progetto, in modo da rendere semplice la creazione di nuovi esercizi.

I prefab sono molto utili per gestire anche gli Spawner, gli elementi da utilizzare per completare lo script. Nella cartella Prefab sono infatti presenti tanti spawner già pronti per tutti gli elementi di utilizzo comune.

Gli spawner hanno un funzionamento semplice:

- Cliccando su uno Spawner viene istanziato un altro box identico, che chiamiamo **DraggableSpawner**, e sarà possibile trascinarlo lungo lo schermo tenendo premuto il pulsante sinistro del mouse;
- Sovrapponendo il DraggableSpawner con una InteractiveBox e rilasciando il pulsante sinistro del mouse, il testo scritto all'interno dello spawner verrà inserito all'interno dell'InteractiveBox;

Per creare un nuovo esercizio con degli Spawner diversi da quelli inclusi come Prefab, basterà istanziarne uno a caso e modificare la label al suo interno in modo da visualizzare il testo adatto.

Sono presenti anche degli Spawner più complessi, in particolare:

- If\_Spawner;
- Else\_Spawner;
- For\_Spawner;
- While\_Spawner;

Quando una interactiveBox rileva il testo di uno di questi spawner, invece che semplicemente ricopiarlo nella scena, quello che farà sarà istanziare al posto della InteractiveBox un altro Prefab contenente un insieme di stringhe e InteractiveBox per replicare il comando corrispondente e renderlo interattivo.





*Esempio del for applicato in un esercizio*

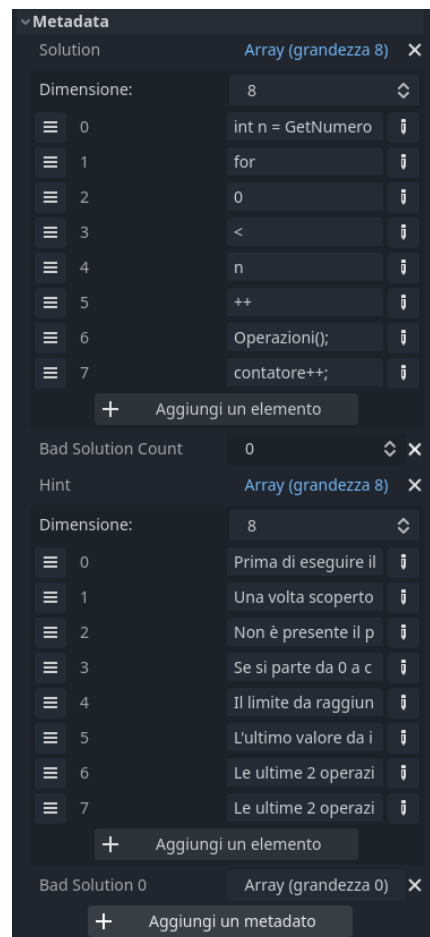
Come è possibile vedere dall'immagine sono presenti varie InteractiveBox per poter definire tutti i parametri del for, mentre all'interno delle parentesi graffe non solo è presente una InteractiveBox, ma una volta riempita è possibile cliccare sul pulsante “+” per aggiungere una nuova riga con una nuova InteractiveBox. È presente inoltre un pulsante “-” laterale con il quale è possibile eliminare il for istanziato e sostituirlo con una InteractiveBox vuota in modo da ripristinare la situazione di partenza in caso di errore.

Con questo meccanismo è possibile creare esercizi complessi e profondi andando a replicare gli stessi ragionamenti che sarebbero necessari durante la scrittura di codice reale.

Ovviamente è possibile anche non utilizzare lo spawner ed istanziare direttamente nello script il Prefab del for o dell'if, per questo ho creato dei Prefab con lo stesso nome ma preceduti dalla scritta “Static”. Queste versioni sono identiche ma con la differenza che le righe all'interno delle parentesi graffe sono fisse. In questo modo è possibile creare esercizi con un grado di complessità più basso e inoltre, come spiegherò successivamente, saranno utili per risolvere alcune problematiche legate all'algoritmo di verifica della risoluzione dell'esercizio. È possibile usare queste versioni anche per personalizzare il prefab, andando ad esempio a sostituire una InteractiveBox con una label per diminuire il numero di incognite che lo studente deve risolvere.

Per verificare la correttezza dell'esercizio, al primo nodo di ogni scena è stato associato lo script **Exercise\_Control**. Tutto quello che farà questo script sarà controllare il valore dei **metadati** del nodo a cui è associato.

I metadati sono dei dati che è possibile aggiungere a qualsiasi nodo attraverso l'inspector. In particolare i dati che controllerà sono:



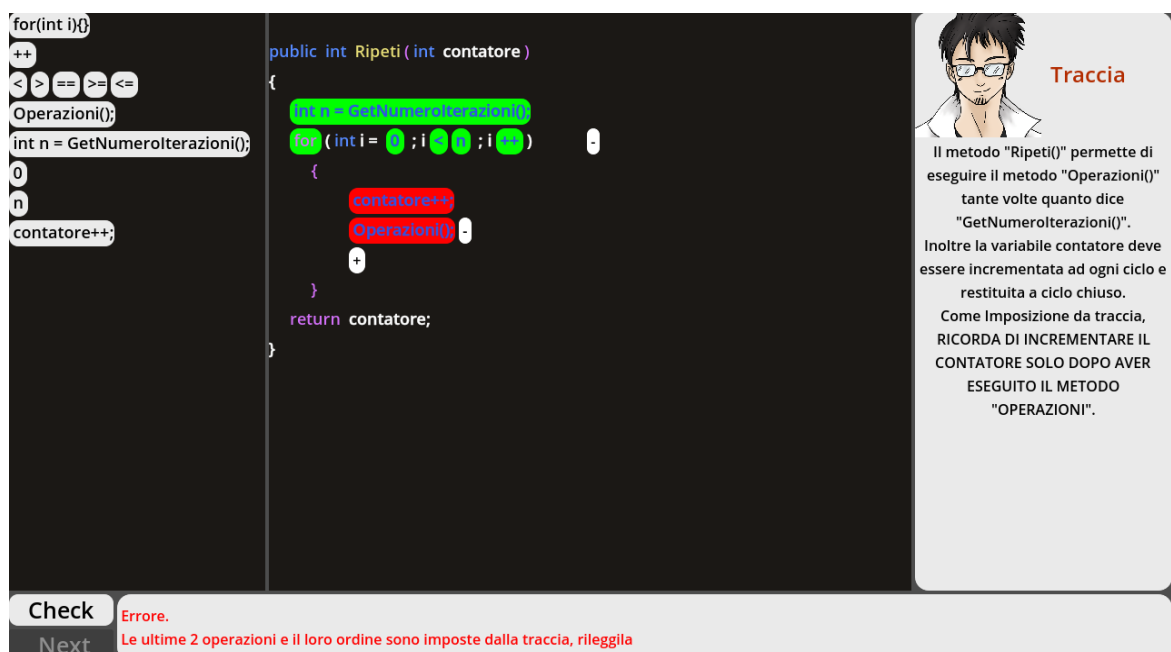
*Metadati del Nodo associato allo script Exercise\_Control*

- **Solution:** un array di tipo stringa al cui interno sono scritti in ordine le soluzioni di tutte le InteractiveBox;
- **Bad Solution Count:** Numero di soluzioni non ottimali possibili, ad esempio ci sono casi in cui una variabile può essere definita come int oppure float senza pregiudicare il funzionamento dello script. In questi casi però la soluzione ottimale sarà solo una;
- **Hint:** un array di tipo stringa al cui interno sono scritti i suggerimenti relativi alle InteractiveBox corrispondenti. Ovviamente avrà la stessa dimensione dell'array solution;
- **Bad Solution N:** un array di tipo stringa che contiene le possibili soluzioni non ottimali della relativa InteractiveBox (N è un numero intero da 0 a Bad Solution Count);

L'algoritmo di verifica sarà quindi molto semplice:

1. Utilizza la classe statica Static\_NodeMethod per individuare tutti i nodi InteractiveBox e li inserisce in una lista;

2. Prende il contenuto della label nella prima posizione della lista ottenuta e lo confronta con il contenuto del primo elemento dell'array Solution.
3. Se le due stringhe sono uguali ripete la stessa operazione per la posizione successiva, altrimenti controlla se è presente quella soluzione negli eventuali array Bad Solution;
4. Se è presente la soluzione in Bad Solution, la interactiveBox interessata dall'errore viene colorata di giallo e viene visualizzato un messaggio che indica la soluzione poco efficiente; se la soluzione non è presente neanche in Bad Solution, allora la InteractiveBox viene colorata di rosso e viene visualizzato un messaggio di errore.
5. Una volta superato il numero di tentativi indicato in fase di setup della modalità esercizio, oltre al messaggio di errore viene visualizzato anche il suggerimento associato a quella InteractiveBox. Ad esempio sbagliando a completare la settima InteractiveBox verrà visualizzato il suggerimento in posizione 6 (dato che si inizia a contare da 0).



The screenshot shows a programming environment with three main sections:

- Left Panel (Code Snippet):** Contains a C# code snippet:
 

```
for(int i){
++
< > == >= <=
Operazioni();
int n = GetNumeroOperazioni();
0
n
contatore++;
```
- Middle Panel (Code Snippet):** Contains a C# code snippet:
 

```
public int Ripeti (int contatore)
{
    int i = GetNumeroOperazioni();
    for (int i = 0 ; i <= i ; i++)
    {
        contatore++;
        Operazioni();
    }
    return contatore;
}
```
- Right Panel (Hint Box):** Titled "Traccia" (Trace), it contains the following text:
 

Il metodo "Ripeti()" permette di eseguire il metodo "Operazioni()" tante volte quanto dice "GetNumeroOperazioni()". Inoltre la variabile contatore deve essere incrementata ad ogni ciclo e restituita a ciclo chiuso. Come Imposizione da traccia, RICORDA DI INCREMENTARE IL CONTATORE SOLO DOPO AVER ESEGUITO IL METODO "OPERAZIONI".
- Bottom Panel (Error Message):** A red bar with the text:
 

Errore.  
Le ultime 2 operazioni e il loro ordine sono imposte dalla traccia, rileggila

*Esempio Suggerimento in seguito a molteplici errori*

In questo modo è possibile creare nuovi esercizi da aggiungere all'applicazione senza dover scrivere alcun tipo di codice, andando semplicemente a sfruttare l'interfaccia grafica dell'editor.

Invece, nel caso in cui si volessero aggiungere nuovi argomenti alla modalità esercizio, sarà necessario modificare la scena di setup della modalità aggiungendo gli elementi UI necessari e scrivere nello script associato il codice

inerente il controllo di questi elementi. Per farlo basterà usare come esempio il codice già presente.

L'ultima cosa da dire riguardo la modalità esercizio è che la classe `Exercise_Control` implementa anche l'interfaccia `BrokerChatter`.

Infatti ad ogni "Check" dell'esercizio, il risultato verrà inviato al Broker. Ma come detto precedentemente è necessario implementare un metodo per gestire gli errori di comunicazione con il Broker.

```
public override void _Process(double delta)
{
    if(!isConnectionOk)
    {
        PopupPanel popupPanel = Static_PopupWindowSpawner.MakePopup(false);
        Static_PopupWindowSpawner.AddLabel(popupPanel, "Errore durante la connessione al Broker\n");
        Static_PopupWindowSpawner.AddLabel(popupPanel, connectionError);
        popupPanel.PopupExclusive(this);
        isConnectionOk = true;
    }
}
```

#### *Implementazione gestione Errori di comunicazione con il Broker*

Per farlo ho scritto questo codice nel `_Process()`, infatti in caso di errore la classe statica andrà a rendere false il booleano `isConnectionOk` e scriverà all'interna della stringa `connectionError` il messaggio di errore riscontrato.

A questo punto, utilizzando la classe statica `Static_PopupWindowSpawner` viene creata una notifica nell'angolo in alto a destra dello schermo contenente il messaggio di errore.

The screenshot shows a game interface with a dark background. On the left, there's a panel with the text "Errore durante la connessione al Broker" and "Host sconosciuto." Below this, there's a text input field containing "Operazioni()", followed by "int n = GetNumerolterazioni();", a numeric input field with "0", and a button labeled "contatore++;". In the center, there's a code editor showing a C# method `public int Ripeti (int contatore)` with a `for` loop and a `return` statement. On the right, there's a character icon and the word "Traccia". Below the character, there's a text box explaining the "Ripeti()" method and the "contatore" variable. At the bottom, there's a "Check" button and a "Next" button. A red error message is displayed at the bottom: "Errore. Le ultime 2 operazioni e il loro ordine sono imposte dalla traccia, rileggila".

Errore durante la connessione al Broker  
Host sconosciuto.  
Operazioni()  
int n = GetNumerolterazioni();  
0  
n  
contatore++;

```
public int Ripeti (int contatore)
{
    int n = GetNumerolterazioni();
    for (int i = 0; i < n; i++)
    {
        contatore++;
        Operazioni();
    }
    return contatore;
}
```

**Traccia**

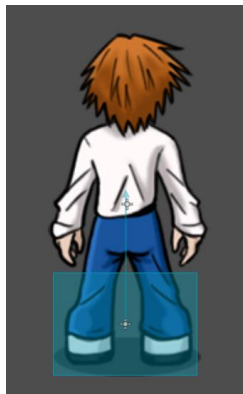
Il metodo "Ripeti()" permette di eseguire il metodo "Operazioni()" tante volte quanto dice "GetNumerolterazioni()". Inoltre la variabile contatore deve essere incrementata ad ogni ciclo e restituita a ciclo chiuso. Come Imposizione da traccia, RICORDA DI INCREMENTARE IL CONTATORE SOLO DOPO AVER ESEGUITO IL METODO "OPERAZIONI".

Check Errore. Le ultime 2 operazioni e il loro ordine sono imposte dalla traccia, rileggila  
Next

#### *Esempio Notifica errore di comunicazione con il Broker*

## 4.5 ScriptConsole

Per realizzare il meccanismo della ScriptConsole e della Doc è stato necessario aggiungere un ulteriore nodo figlio al RigidBody2D dell'avatar del giocatore, ovvero il nodo **RayCast2D**. Questo nodo viene visualizzato come una freccia o un vettore di colore azzurro che spunta fuori del CollisionShape del personaggio (il rettangolo azzurro). Se questa freccia entra all'interno di un altro CollisionShape riesce a rilevarlo ottenendo così un riferimento verso di esso.



*Sprite con CollisionShape2D e RayCast2D*

Utilizzando questo nodo è stato quindi possibile rilevare gli oggetti selezionati sui quali eseguire le operazioni della console. Al raycast ho quindi associato uno script che ho chiamato **Focus\_InteractiveObj** e gli ho assegnato un riferimento a CharController in modo da ruotare il vettore in base alla direzione verso cui il personaggio è rivolto, in modo che selezionasse sempre gli oggetti di fronte allo sprite.

Una volta selezionato un oggetto a quest'ultimo viene anche associato uno **Shader**. Gli Shader sono dei componenti capaci di disegnare elementi grafici al di sopra degli oggetti a cui sono associati, ogni engine di questo tipo dispone di un proprio linguaggio di programmazione con cui è possibile realizzarli. In questo caso lo shader non fa altro che andare a disegnare un contorno bianco lungo lo sprite dell'oggetto selezionato. Questo shader è stato scaricato gratuitamente dal sito GodotShaders al seguente indirizzo:

<https://godotshaders.com/shader/2d-outline-inline/>

Una volta selezionato un oggetto, premendo il pulsante *“backslash”* (\), è possibile aprire la ScriptConsole. Questo è possibile grazie alla classe

**InputManager** da me realizzata, che ha il compito di assegnare determinate azioni a determinati input della tastiera.

```
2 references
private void OpenConsole()
{
    if(scriptConsole == null)
        return;
    if (scriptConsole.Visible)
    {
        if (dockControl.Visible)
            return;
        scriptConsole.Visible = false;
        charController.IsMovementEnabled = true;
        return;
    }

    Focus_InteractiveObj focus = (Focus_InteractiveObj)Static_NodeMethod.GetChildType(characterNode, "RayCast2D");
    if (focus.Body == null)
    {
        GD.Print("raycast null");
        return;
    }
    if (focus.Body != null)
    {
        Interactable switchControl = (Interactable)focus.Body;
        selectedObject = switchControl.GetConsoleInteraction();
        scriptConsole.SelectedObj = selectedObject;
        GD.Print("ObjectConsole selected " + scriptConsole.SelectedObj.ToString());
        scriptConsole.Visible = true;
        scriptConsole.ProcessMode = Node.ProcessModeEnum.Inherit;
        charController.IsMovementEnabled = false;
        charController.StopAnimation();
    }
}
```

#### *Metodo OpenConsole di InputManager*

Con questo metodo la classe InputManager ottiene il riferimento al Rigidbody dell'oggetto selezionato e lo converte con un cast in un oggetto di tipo **Interactable**. Interactable è una interfaccia che ho definito per indicare tutti gli oggetti interagibili e quindi controllabili attraverso la console.

A questo punto con il metodo GetConsoleInteraction di Interactable è possibile ottenere il riferimento all'oggetto **ConsoleInteraction**, una classe astratta contenente tutta la logica per interagire con la ScriptConsole e la Doc.

Infine il riferimento a ConsoleInteraction viene passato alla scriptConsole che lo memorizza all'interno di una variabile nominata SelectedObj.

ConsoleInteraction è stata realizzata come classe astratta per due motivi:

- Definire una interfaccia grazie alla quale poter scrivere uno script unico che potesse adattarsi a tanti oggetti diversi;
- Implementare tutti i metodi di base necessari al funzionamento della ScriptConsole senza dover riscrivere più volte lo stesso codice e andando a semplificare la vita ai futuri studenti che vorranno implementare nuove funzionalità, permettendo loro di concentrarsi unicamente sul nuovo codice da scrivere;

All'interno della classe astratta `ConsoleInteraction` sono presenti varie proprietà e metodi. In particolare:

- **propertiesList**: una lista contenente tutte le proprietà dell'oggetto con cui è possibile interagire attraverso la `ScriptConsole` e la `Doc`;
- **methodList**: una lista contenente tutti i metodi dell'oggetto con cui è possibile interagire attraverso la `ScriptConsole` e la `Doc`;
- **RegisteredObject**: oggetto terzo attivabile interagendo con quello selezionato;
- **UniqueName**: riferimento utilizzabile per associare questo oggetto ad un altro;
- **\_Action(StringBuilder sb = null)**: metodo astratto da implementare per definire il comportamento dell'oggetto quando viene interagito;
- **SetPropertiesValues()**: metodo astratto da implementare per specificare come interagire con le singole proprietà definite all'interno di `propertiesList`;
- **Initialize()**: metodo astratto da implementare in cui inizializzare i valori dell'oggetto come `propertiesList` e `methodList`;
- **IsParameterExist(string nameVar)**: metodo che verifica l'esistenza di una proprietà in `propertiesList`;
- **GetParameterValue(string nameVar)**: metodo che restituisce il valore di una proprietà;
- **SetInt/SetBool/SetString()/ecc..** : metodi utili per assegnare un valore ad una proprietà con un tipo specifico. Da usare durante l'implementazione di `SetPropertiesValue()`;
- **InvokeMethod(string methodName, List<object> args)**: metodo che permette di invocare uno dei metodi presenti in `methodList`;

Nel codice di gioco sono presenti varie classi che implementano `ConsoleInteraction`, ad esempio **`ConsoleInt_Door`**.

Questa classe implementa il funzionamento di una porta all'interno del gioco. All'interno di `propertiesList` ha 2 proprietà: `Frame`(dello sprite) e `UniqueName`.

Il metodo `SetPropertiesValue` sarà quindi:

```

2 references
public override void SetPropertyValue(string name, string value, StringBuilder sb)
{
    if (IsParameterExist(name))
    {
        switch (name)
        {
            case "Frame": try{
                                SetInt(ref frame, value, "Frame", sb, (spriteList[0].Hframes * spriteList[0].Vframes));
                            } catch{
                                throw;
                            }

                                //sprite.Frame = frame;
                                SetFrame(frame);
                                propertiesList["Frame"] = frame.ToString();
                                break;

            case "UniqueName": if(!IsUniqueNameUnique(value, spriteList[0].GetTree(), sb))
                                break;
                                SetString(ref uniqueName, ref value, "UniqueName", sb);
                                propertiesList["UniqueName"] = uniqueName;
                                GD.Print("Unique name = " + propertiesList["UniqueName"].ToString());
                                break;

        }
        return;
    }
    sb.AppendLine(Costanti.WarningParameterNotFound);
}

```

### *Esempio implementazione del metodo astratto SetPropertyValue*

MethodsList è invece vuota, in quanto questo oggetto specifico non avrà metodi richiamabili attraverso la console. Infatti ConsoleInt\_Door implementa anche un'altra interfaccia, ovvero **SwitchControlled**. Questo perché le porte devono essere aperte interagendo con altri oggetti come leve e pulsanti. È quindi presente una classe **ConsoleInt\_Switch** che avrà all'interno di methodsList il metodo string Action(), il quale permette di controllare gli oggetti che implementano l'interfaccia SwitchControlled andando ad eseguire il metodo **OpenAction()** oppure **CloseAction()** in base al suo stato.

All'interno di ConsoleInt\_Switch sono quindi presenti 2 metodi dal nome simile:

1. **string Action():** metodo presente all'interno di methodsList ed eseguito attraverso la ScriptConsole;
2. **void \_Action(StringBuilder sb = null):** metodo astratto ereditato da ConsoleInteraction da implementare e che viene eseguito interagendo con l'oggetto attraverso la tastiera;

Questa è una differenza necessaria, come spiegherò successivamente, la scriptConsole utilizza uno StringBuilder durante le sue operazioni per salvare al suo interno tutti i messaggi, di errore e non, che dovrà far visualizzare al giocatore per informarlo sullo stato delle operazioni eseguite. Tuttavia non esiste alcuno StringBuilder se si interagisce con l'oggetto al di fuori della ScriptConsole. Per questo motivo sarà necessario scrivere all'interno di void



`_Action(StringBuilder sb = null)` il codice relativo a ciò che deve avvenire in seguito all'interazione; mentre all'interno di `Action()` solitamente sarà necessario definire un oggetto `StringBuilder` da inserire come parametro di `_Action(...)`, in modo da sovrascrivere il parametro nullo di default, e infine restituire le stringhe contenute nello `StringBuilder`. In questo modo il contenuto verrà ricevuto dalla `ScriptConsole` che provvederà a riportarlo a schermo.

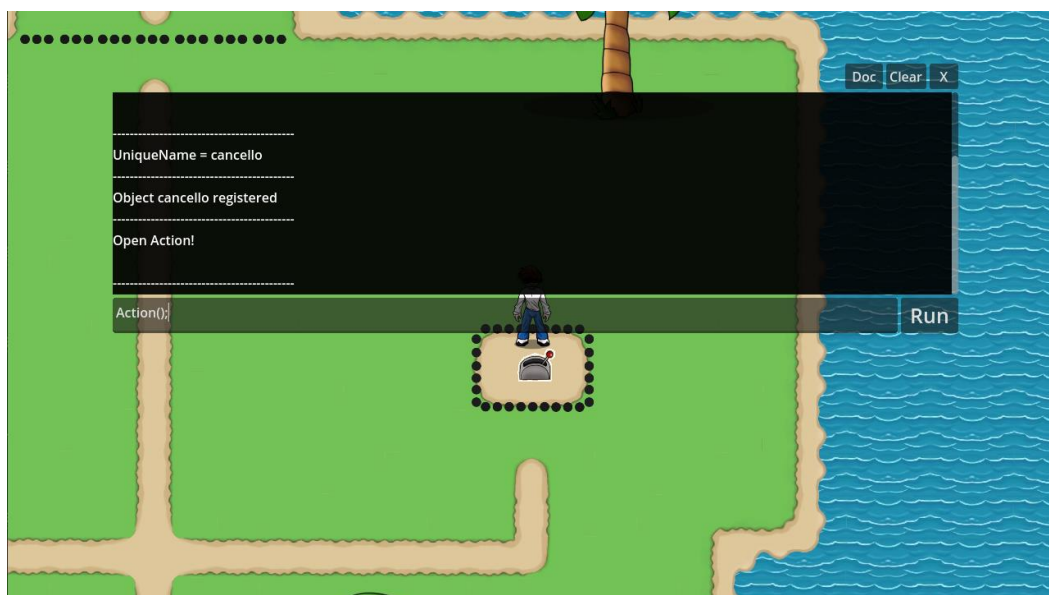
```

7 references
public override void _Action(StringBuilder sb = null){
    if(registeredObj == null){
        if(sb != null)
            sb.AppendLine(Costanti.Error + "No object registered in Event manager");
        return;
    }
    if(isPlaying)
        return;
    SwitchControlled switchControlled = ((Interactable)registeredObj).GetConsoleInteraction().GetSwitchControlled();
    if(switchControlled == null)
    {
        if(sb == null)
            return;
        sb.AppendLine(Costanti.Error + "The selected object is not valid");
        return;
    }
    if(sprite.Frame == 0){
        if(sb != null)
            sb.AppendLine("Open Action!");
        switchControlled.OpenAction(switchControl);
        OpenAction(sprite);
        return;
    }
    switchControlled.CloseAction(switchControl);
    CloseAction(sprite);
    if(sb != null)
        sb.AppendLine("Close Action!");
    }

1 reference
public string Action(){
    StringBuilder sb = new StringBuilder();
    this._Action(sb);
    return sb.ToString();
}

```

*Esempio implementazione del metodo astratto `_Action(StringBuilder sb = null)`*



*Esempio Output nella Console in gioco*

Per quanto riguarda il funzionamento della ScriptConsole in sé per sé, invece, è presente una classe **CodeAnalyzer** che si occupa di leggere il testo scritto dall'utente ed individuare le singole istruzioni eseguendole in ordine.

CodeAnalyzer è in grado di riconoscere:

- **Comandi:** ovvero le funzionalità interne del codice(for, if, ecc...). Questi comandi sono caratterizzati da una stringa iniziale, delle parentesi tonde che racchiudono le condizioni del comando e infine delle parentesi graffe contenenti il codice da eseguire quando le condizioni sono soddisfatte. In particolare CodeAnalyzer individua il nome del comando, le condizioni e il codice contenuto nelle parentesi graffe e invia questi dati ad un'altra classe chiamata **CommandControl**, al cui interno sono presenti dei riferimenti a tutti gli oggetti specializzati nell'esecuzione dei singoli comandi. Se CommandControl riceve un nome valido, allora eseguirà il metodo associato dell'oggetto adeguato;

```
1 reference
public CommandControl(CodeAnalyzer analyzer, ParameterControl control){
    codeAnalyzer = analyzer;
    parameterControl = control;

    forCommand = new ForCommand(analyzer);
    ifCommand = new IfCommand(analyzer, parameterControl);
}

1 reference
public void CheckCommand(string command, ref string body, StringBuilder sb, float? i = null, string nameForCounter = "")
{
    if (command.StartsWith("for("))
    {
        //sb.AppendLine("for command Action!");
        forCommand.For(command, ref body, sb);
    }
    else
    {
        if (command.StartsWith("if("))
        {
            sb.AppendLine("if command Action!");
            ifCommand.If(command, ref body, sb, i, nameForCounter);
        }
        else
        {
            sb.AppendLine(Costanti.ErrorNotRecognizedCode);
        }
    }
}
```

*Metodo CheckCommand di CommandControl*

- **Operazioni di assegnazione:** riconosce la presenza di una stringa indicante il nome di una proprietà, il simbolo "=", il valore da assegnare e la presenza del ";" per chiudere la linea. Quando si rende conto di trovarsi in questa situazione si comporterà allo stesso modo di quanto detto con CommandControl, invierà i dati alla classe **ParameterControl** che controllerà se all'interno dell'oggetto selezionato è presente la proprietà richiesta e nel caso ne aggiornerà il valore;
- **Metodi:** controlla la presenza di una stringa indicante il nome del metodo, la presenza delle parentesi tonde contenenti gli eventuali

argomenti del metodo e infine il “;” per indicare la chiusura della linea. Anche questo caso i dati estrapolati verranno inviati ad una classe **MethodControl** che verificherà la presenza del metodo richiesto e lo eseguirà inserendo gli eventuali argomenti;

Avendo organizzato il codice in questo modo risulta abbastanza semplice l’aggiunta di nuovi metodi o comandi. Basterà infatti implementare le classi contenenti il codice necessario al funzionamento della nuova funzionalità e successivamente aggiungere alla relativa classe di controllo (ParameterControl, MethodControl o CommandControl) un riferimento e una chiamata alla classe appena realizzata.

Nel caso dell’aggiunta di un nuovo metodo potrebbe essere necessario un passaggio in più. Infatti gli argomenti individuati da CodeAnalyzer vengono salvati all’interno di una lista di stringhe per poterli poi riconvertire negli oggetti e tipi originali. Tuttavia per eseguire questa operazione è necessario l’utilizzo di un cast, in quanto il codice non può capire in autonomia qual è il tipo corretto in cui convertire un dato parametro.

Perciò all’interno di MethodControl è presente il metodo **CalcArgumentList**, il quale, dando come argomento la lista di argomenti e il nome del metodo, andrà a creare una nuova lista di Object al cui interno inserirà gli argomenti con il tipo corretto. In questo modo il metodo potrà essere eseguito.

```
1 reference
private List<Object> CalcArgumentList(List<string> list, string methodName){
    List<Object> argumentList = new List<Object>();
    if( methodName.Equals(Costanti.MethodMoveUp.Substring(0, Costanti.MethodMoveUp.IndexOf("("))) ||
        methodName.Equals(Costanti.MethodMoveDown.Substring(0, Costanti.MethodMoveDown.IndexOf("("))) ||
        methodName.Equals(Costanti.MethodMoveLeft.Substring(0, Costanti.MethodMoveLeft.IndexOf("("))) ||
        methodName.Equals(Costanti.MethodMoveRight.Substring(0, Costanti.MethodMoveRight.IndexOf("(")))
    {
        argumentList.Add(list[0].ToInt());
    }
    return argumentList;
}

1 reference
private void CheckMethodInSelectedObj(string methodName, List<object> argumentList, StringBuilder sb){
    GD.Print("Cerco metodi nel selectedObj");
    ConsoleInteraction selectedObj = codeAnalyzer.GetSelectedObject();
    List<string> methodsList = selectedObj.GetAllMethodsList();
    foreach(string s in methodsList){
        //methodName += "(";
        string str = s.Substring(0, s.IndexOf("("));
        GD.Print("method searched: " + methodName + " | method in class: " + str);
        if(methodName.Equals(str)){
            GD.Print("Find method " + methodName);
            GD.Print("Method to execute " + s);
            sb.AppendLine(selectedObj.InvokeMethod(s, argumentList));
        }
    }
}
```

*Metodi CalcArgumentList e CheckMethodInSelectedObj della classe MethodControl*

## 4.6 Doc

Per quanto riguarda la Doc invece il codice è realizzato per funzionare autonomamente. È presente uno script **DocControl** che analizza l'oggetto selezionato e crea le voci della tabella sulla base degli elementi presenti all'interno di `parametersList` e `methodsList` di `ConsoleInteraction`.

Nel caso si volessero aggiungere nuovi parametri o metodi sarà sufficiente modificare la classe **DocDescReader** contenente 2 metodi:

1. **GetDescProperty(string name):** in base al nome della proprietà cercata restituisce la sua descrizione da visualizzare nel gioco;
2. **GetDescMethod(string name):** in base al nome del metodo cercato restituisce la sua descrizione da visualizzare nel gioco;

Tutte le descrizioni sono salvate all'interno di una classe **Costanti** come stringhe costanti.

```
2 references
public class DocDescReader
{
    1 reference
    public string GetDescProperty(string name)
    {
        string desc = "";
        if (name.Equals(Costanti.PropertyFrame))
        {
            desc = Costanti.PropertyFrameDesc;
        } else
        if (name.Equals(Costanti.PropertyRegisteredObj))
        {
            desc = Costanti.PropertyRegisteredObjDesc;
        } else
        if (name.Equals(Costanti.PropertyTestBool))
        {
            desc = Costanti.PropertyTestBoolDesc;
        } else
        if (name.Equals(Costanti.PropertyUniqueName))
        {
            desc = Costanti.PropertyUniqueNameDesc;
        }

        return desc;
    }

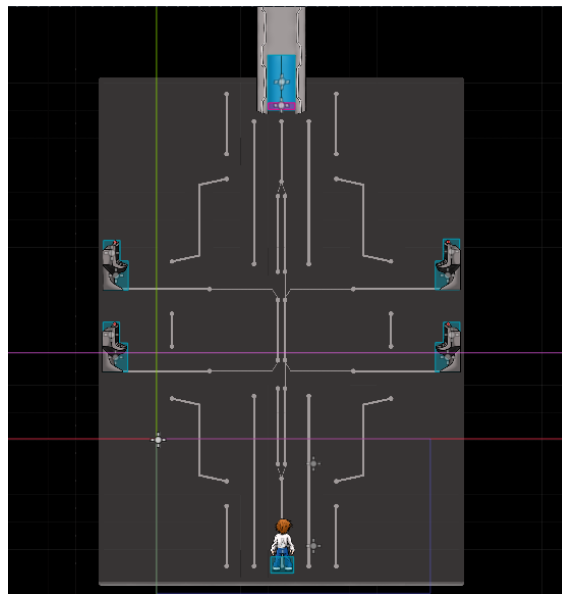
    1 reference
    public string GetDescMethod(string name){
        string desc = "";
        if (name.Equals(Costanti.MethodAction))
        {
            desc = Costanti.MethodActionDesc;
        } else
        if (name.Equals(Costanti.MethodMoveUp))
        {
            desc = Costanti.MethodMoveUpDesc;
        } else
        if (name.Equals(Costanti.MethodMoveDown))
        {
            desc = Costanti.MethodMoveDownDesc;
        }
    }
}
```

*Metodi `GetDescProperty` e `GetDescMethod` della classe `DocDescReader`*

## 5 I Livelli

### 5.1 Livello 1

Durante il primo livello il giocatore si ritrova in uno spazio scuro, l'ambientazione esplorabile è composta da una piattaforma che ricorda un circuito stampato sopra il quale sono presenti delle console interagibili. In questo livello la ScriptConsole non è tuttavia ancora utilizzabile.



*Scena livello 1*

Iniziata la partita verrà riprodotto un dialogo in cui il professore avvertirà il giocatore che prima di poter entrare nel gioco è necessario risolvere alcuni errori di compilazione che si sono verificati.

Infatti tutta la scena vuole essere una sorta di area di Debug esterna al gioco all'interno della quale risolvere gli errori del compilatore.

Interagendo con le console verranno riprodotti alcuni esercizi non dissimili dalla modalità esercizio da risolvere, risolvendo tutti gli esercizi verrà sbloccato un ascensore nella parte in alto della mappa, permettendo al giocatore di raggiungere il secondo livello.

Inoltre, per aumentare l'immersività, gli script da risolvere dalle console causeranno un effetto reale all'interno del gioco. Ad esempio uno degli errori riguarderà il codice relativo alla **riproduzione delle animazioni**, infatti finché

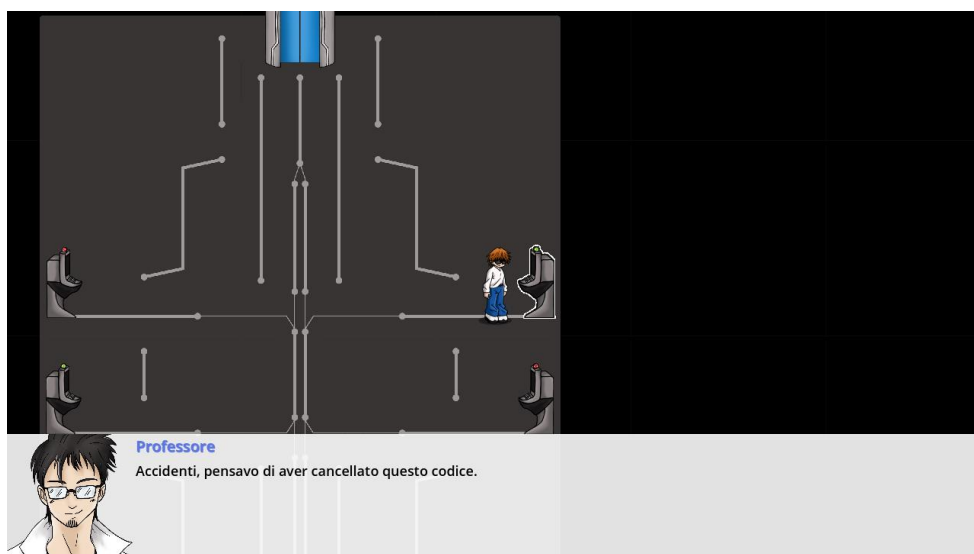
il giocatore non risolverà quell'errore, il suo sprite si muoverà nella mappa senza alcun tipo di animazione.



*Dialogo che appare quando l'utente prova a muoversi nel livello 1*

Gli altri errori riguarderanno:

- **La riproduzione della musica:** risolvendolo verrà riprodotta della musica in loop come sottofondo;
- **Apertura e chiusura del menu di gioco:** il menu sarà inaccessibile prima di aver risolto lo script;
- **Script per gestire una partita a battaglia navale:** Non inerente alle meccaniche descritte prima, una volta risolto il professore dirà che si tratta di codice non più utilizzato all'interno dell'applicazione che ha dimenticato di eliminare;



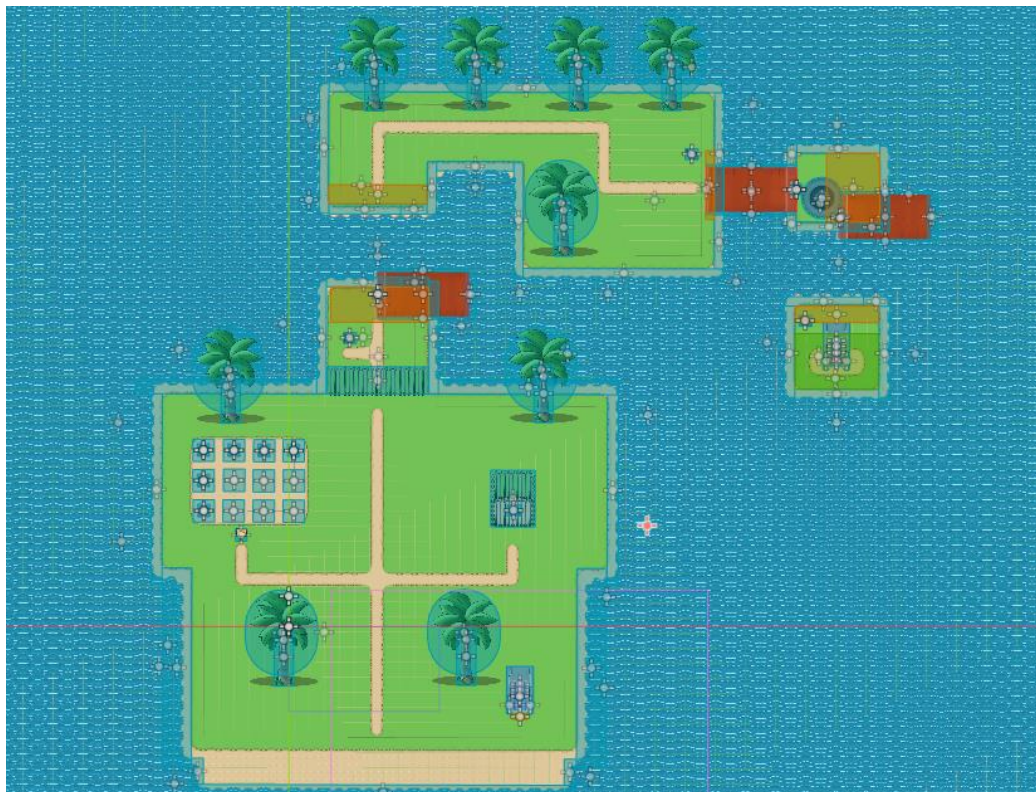
*Dialogo che appare quando l'utente risolve l'esercizio di battaglia navale nel livello 1*



## 5.2 Livello 2

Nel secondo livello il giocatore si trova su un'isola tropicale. Una volta usciti dall'ascensore, il professore resterà sorpreso nel vedere come il livello sia molto diverso da quello che sarebbe dovuto essere, in particolare noterà la presenza di un altro ascensore in lontananza e pertanto chiederà al giocatore di raggiungerlo per scoprire cosa sta succedendo.

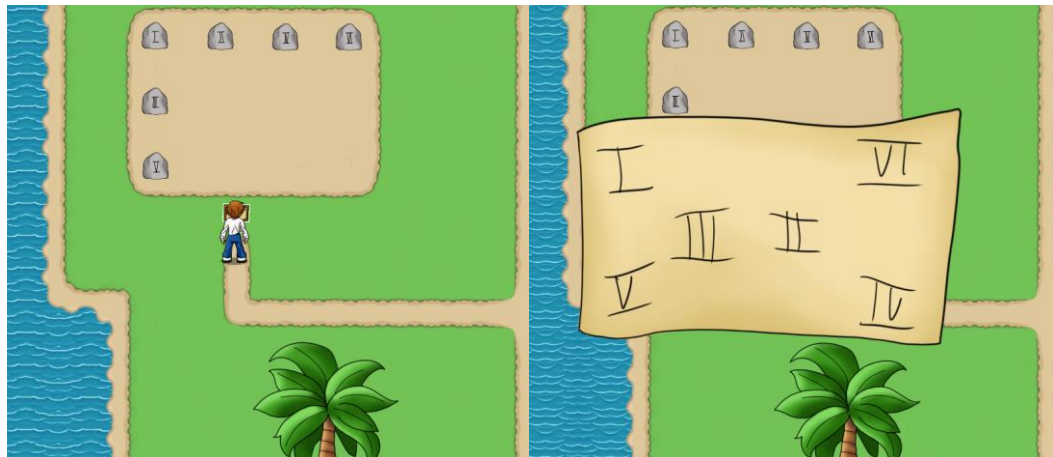
Il giocatore dovrà quindi usare la ScriptConsole per risolvere i vari puzzle ed avanzare nel gioco fino a raggiungere il nuovo ascensore.



*Scena livello 2*

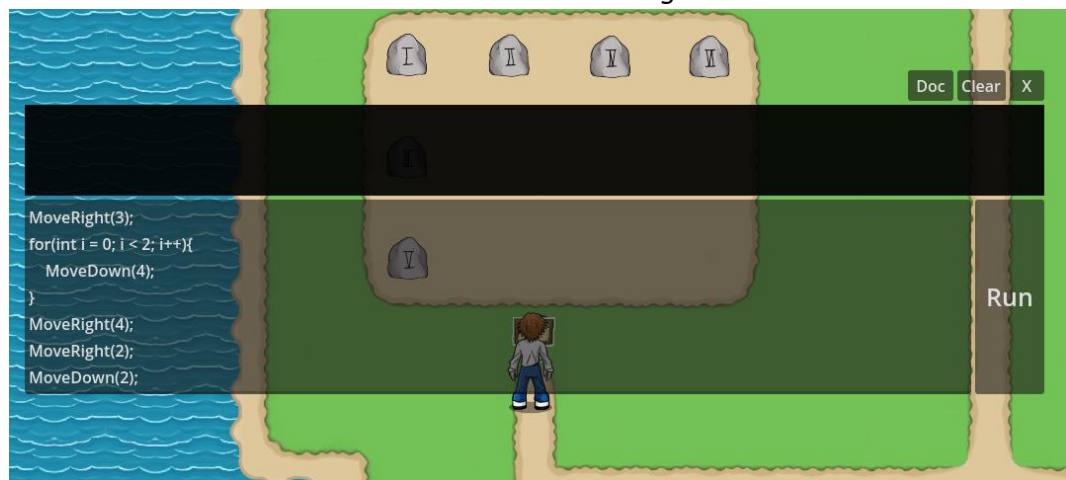
Per avanzare il giocatore dovrà:

1. Risolvere il puzzle a sinistra del cancello sull'isola più in basso. Interagendo con il cartello verrà visualizzata una immagine che spiegherà come disporre le rocce. Usando la Doc il giocatore scoprirà la presenza di vari metodi per spostare le suddette, a questo punto avrà vari modi per scrivere il codice per ottenere la formazione corretta. Una volta risolto il puzzle si apriranno le sbarre che bloccano la leva a destra;



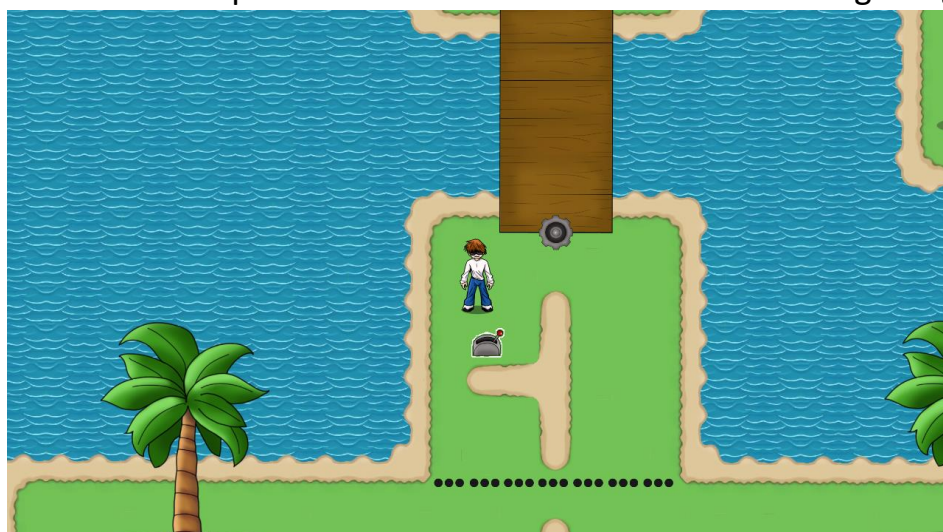
*L'utente si avvicina al cartello*

*Interagendoci è visibile l'indizio*



*Aprendo la console l'utente scrive il codice adatto alla risoluzione*

2. A questo punto il giocatore dovrà assegnare al cancello un UniqueName, in modo da poterlo registrare al RegisteredObj della leva. In questo modo interagendo con la leva sarà possibile aprire il cancello;
3. Adesso il giocatore si ritroverà di fronte un ponte con un ingranaggio ed una leva. Anche in questo caso basterà associare la leva all'ingranaggio.



*Una volta risolto il puzzle il ponte ruota permettendo di raggiungere l'altra sponda*

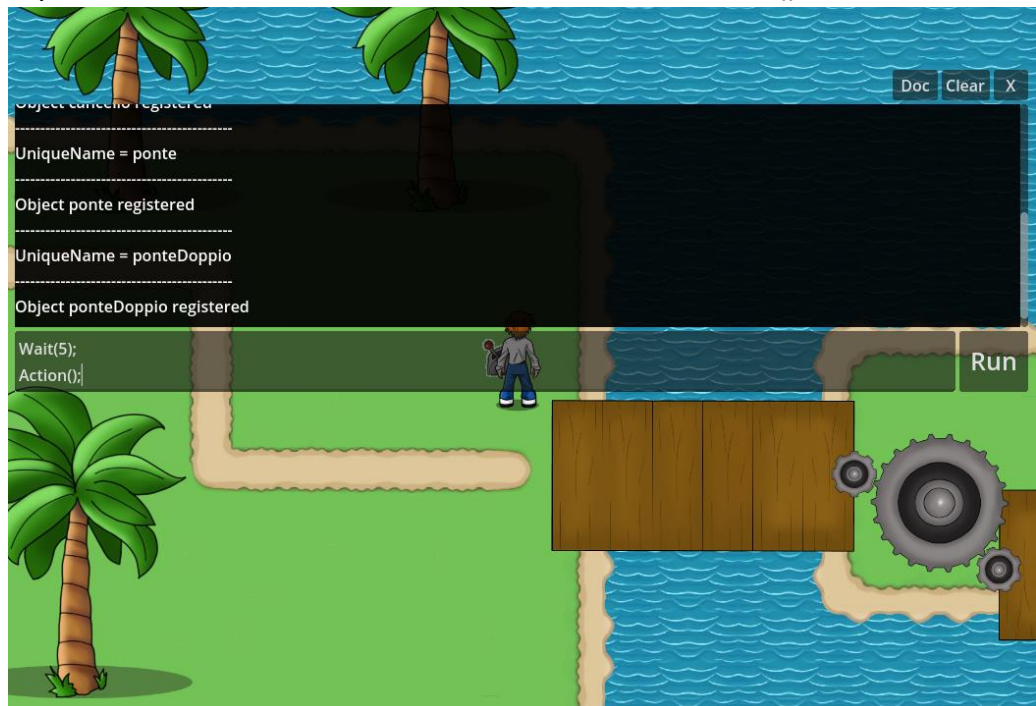


4. Andando avanti il giocatore si troverà di fronte un altro ponte. Questa volta però sarà un ponte doppio, quindi azionandolo con una leva sarà possibile chiudere una parte del percorso, ma così facendo ne verrà aperta un'altra impedendo il passaggio. Inoltre, una volta azionato, partirà un conto alla rovescia al cui termine verrà ripristinata la posizione iniziale del ponte.



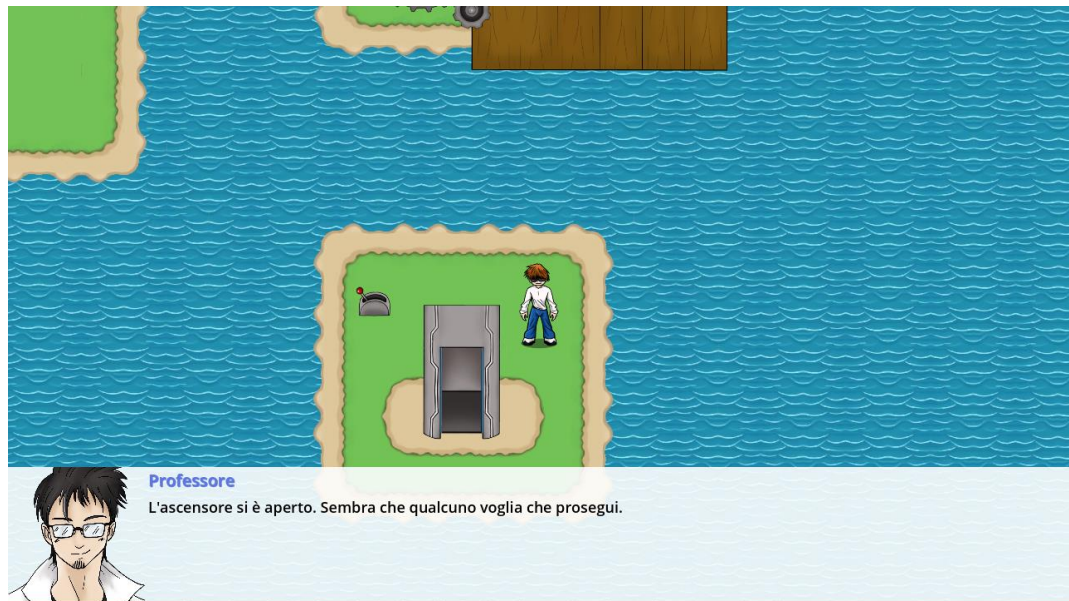
*Puzzle Ponte Doppio temporizzato*

Il giocatore dovrà quindi usare il metodo `Wait()` prima di eseguire l'interazione, in questo modo il ponte verrà azionato in ritardo dando il tempo di spostare il personaggio sull'isola centrale e successivamente su quella finale una volta terminata l'attesa del `Wait()`.



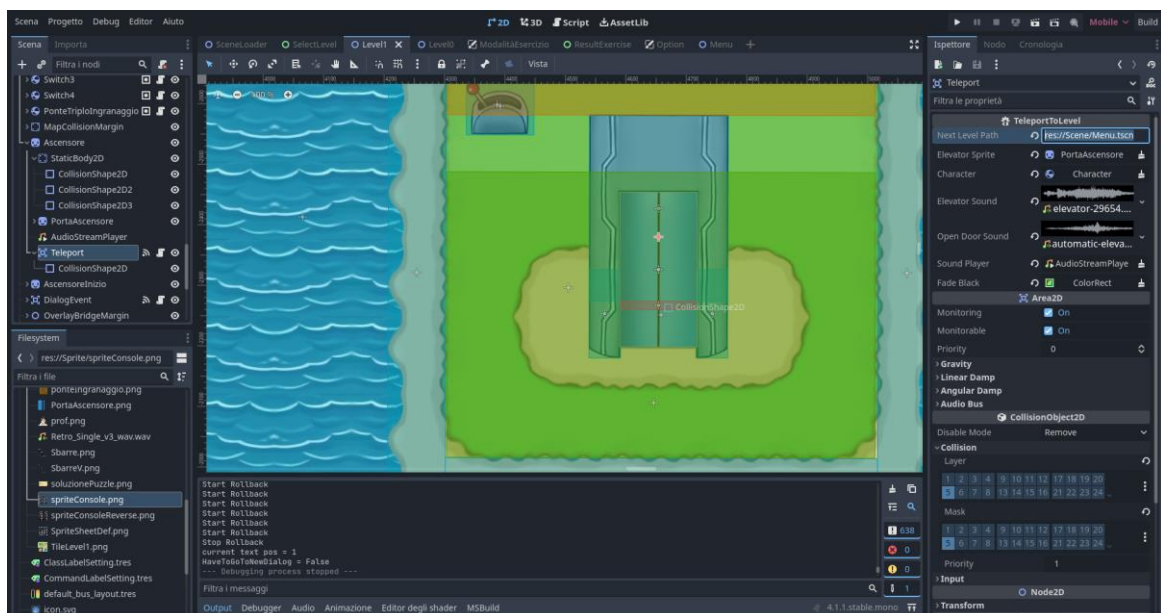
*Esempio risoluzione Puzzle Ponte Doppio temporizzato*

5. Una volta raggiunta l'ultima isola l'ascensore si aprirà in automatico e una volta entrati il gioco terminerà.



*Dialogo finale livello 2*

Quando verranno creati nuovi livelli basterà selezionare l'ascensore e inserire dall'inspector il filepath, o percorso, della scena contenente il nuovo livello.



*Inspector Nodo Teleport con script TeleportToLevel*

Per rendere l'esperienza ancora più immersiva ho aggiunto un ulteriore nodo figlio del RigidBody del giocatore, ovvero **AudioListener2D**. Questo nodo è in grado di sentire i suoni riprodotti dai nodi **AudioStreamPlayer2D** in base alla distanza che li separa nella scena. In questo modo il giocatore potrà ascoltare il suono del mare mentre si trova lungo le estremità delle isole e più si sposterà verso l'interno, più quel suono diventerà flebile, fino a scomparire del tutto.

## 6 Conclusioni

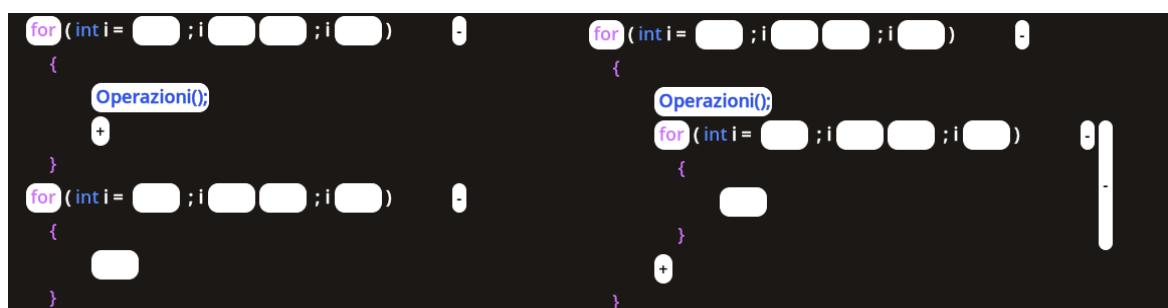
### 6.1 Considerazioni Finali

Come detto più volte, l'obiettivo di questo progetto era creare una base che fosse ampliabile nel tempo da più sviluppatori. Proprio per questo ho investito maggiore impegno nel dare alla applicazione una struttura adatta allo scopo piuttosto che alla creazione di vero e proprio contenuto di gioco.

Infatti i livelli della campagna sono molto semplici e, sapendo già cosa bisogna fare, è sufficiente poco tempo per completarli; è possibile dire lo stesso anche per la modalità esercizio, che alla fine non contiene un numero di esercizi molto elevato al momento, ma contiene tutto il necessario per essere facilmente espansa.

Proprio riguardo a questo è necessario fare alcune considerazioni: la modalità esercizio presenta un algoritmo di verifica della soluzione molto semplice, questo proprio per permettere la creazione di nuovi esercizi senza la necessità di dover scrivere uno script specifico per ognuno di essi. Questo però ha portato ad alcune semplificazioni che in alcuni casi specifici possono portare a situazioni inattese.

Infatti nel caso fosse necessario l'utilizzo di blocchi dinamici nidificati (ad esempio "for", "if", "else"), l'algoritmo non sarebbe in grado di percepire la differenza tra le due situazioni mostrate nell'immagini sotto.



*Scena con for dinamico*

*Scena con for statico*

Questo perché l'algoritmo semplicemente controlla che nella InteractiveBox successivamente ci sia il valore atteso, ma, dato che questi blocchi possono contenere un numero indefinito di righe di codice, non riesce a capire se quel valore atteso si trova dentro o fuori il blocco.

È anche per risolvere questi errori che sono stati introdotti i blocchi statici.

In questo caso, infatti, inserendo il primo for come blocco statico il numero di righe contenuto al suo interno sarà noto e quindi sarà impossibile che si verifichi una situazione simile a quella descritta.

Per la creazione di un nuovo esercizio è necessario quindi che lo sviluppatore incaricato presti attenzione alla logica della correzione dell'esercizio e scriva quindi qualcosa in modo tale da non far verificare situazioni simili o dove le soluzioni ottimali accettabili sono due, in quanto l'array soluzione è unico.

## 6.2 Sviluppi Futuri

Per quanto riguarda possibili sviluppi futuri sono tante le funzionalità che potrebbero essere introdotte.

Per la modalità esercizio si potrebbero aggiungere nuovi argomenti ed esercizi, mentre per la campagna si potrebbero fare varie aggiunte:

- Nuovi comandi, metodi e proprietà per implementare nuove soluzioni di gameplay per la ScriptConsole;
- Creazione di nuovi livelli con le grafiche messe a disposizione o delle nuove;

In particolare nelle idee iniziali sarebbe dovuto essere presente un livello ambientato all'interno di una scheda madre, dove il giocatore avrebbe dovuto spostarsi tra i vari settori della scheda(CPU, GPU, RAM, Unità Periferiche, ecc...) per risolvere gli errori di comunicazione tra i suddetti, andando così ad istruire il giocatore sul funzionamento dell'architettura di un calcolatore.

Per la ScriptConsole, invece, sarebbe dovuto essere presente una proprietà di tipo stringa di ConsoleInteraction chiamata **OverrideAction**. Andando a salvare delle linee di codice al suo interno, una volta eseguita l'azione Action(), la console avrebbe invece inviato a CodeAnalyzer il contenuto della stringa. In questo modo il giocatore avrebbe potuto creare delle azioni personalizzate da far eseguire agli oggetti al posto di quelle standard.

Ovviamente questi sono solo alcuni esempi, ma, data la natura OpenSource del progetto, chiunque deciderà di sviluppare su questa applicazione potrà implementare ciò che riterrà più opportuno.

