**Documentazione ScriptAdventure**

**INDICE**

Setup pag.1

Introduzione a Godot pag.6

Documentazione progetto pag.12

Addon Godotpag.12

Controllo Personaggio e Camerapag.12

Classi Statiche pag.14

Modalità Esercizio pag.16

ScriptConsole pag.25

Doc pag.32

**Setup**

1. Per prima cosa è necessario scaricare Godot, ovvero l’engine usato per lo sviluppo del gioco. Per farlo è necessario andare al link:

<https://godotengine.org/download/windows/>

Una volta aperta la pagina, scaricare la versione “***Godot Engine -.NET****”*, ovvero la versione compatibile con C#.

Per il progetto è stata utilizzata la versione 4.1.1 dell’engine, ma anche versioni successive dovrebbero essere compatibili. Nel caso è possibile scaricare la medesima versione dall’archivio presente nella stessa pagina.

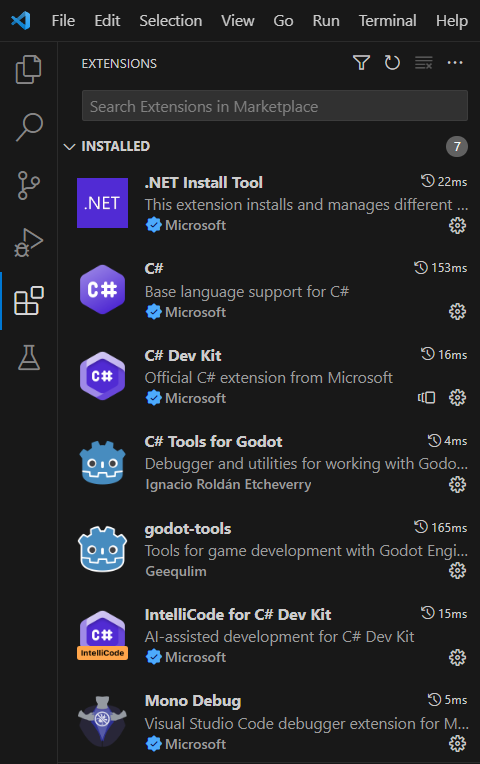
1. Una volta scaricato ed installato l’engine è necessario scaricare l’sdk .NET. Per farlo basta andare all’indirizzo:

<https://dotnet.microsoft.com/en-us/download>

A questo punto bisogna fare attenzione e non scaricare .NET SDK normale, ma bisogna scrollare la pagine e scaricare, nella sezione “***Other .NET download***” la versione “***.NET Coding Pack***”, in questo modo andremo ad installare sia l’sdk che Visual Studio Code (VS Code), ovvero l’IDE che andremo ad utilizzare per scrivere script con Godot.

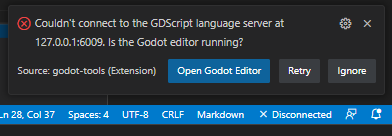
1. Completata l’installazione andiamo ad importare il progetto all’interno dell’engine. Farlo è molto semplice, una volta scaricata la cartella del progetto sarà sufficiente posizionarla in un percorso a noi congeniale e successivamente aprire Godot e cliccare sul pulsante “***Importa***” posizionato sulla destra dell’interfaccia. Clicchiamo quindi su sfoglia e andiamo a cercare la cartella contenente il progetto e troviamo al suo interno il file “***project.godot***” e lo selezionamo. A questo punto il progetto verrà caricato e potremo aprirlo senza problemi.
2. Adesso dobbiamo installare delle estensioni in VS Code per facilitarci l’utilizzo con Godot.

Per fare ciò, ci basterà aprire VS Code e cliccare nel menu laterale a sinistra, l’icona la quinta icona a partire dall’alto, ovvero la voce estensioni. A questo punto basterà scrivere il nome dell’estensione da installare nella barra di ricerca. Le estensioni che bisognerà avere installate sono quelle mostrate nella immagine sottostante.

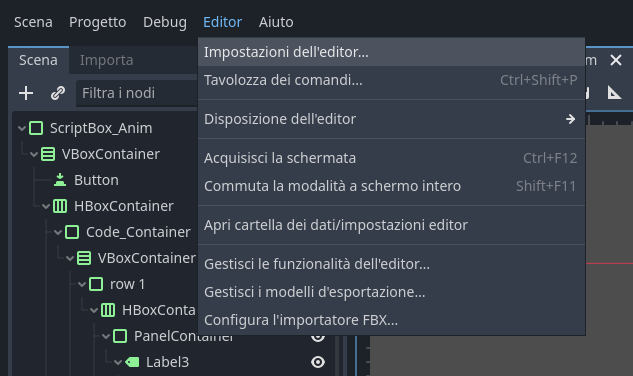


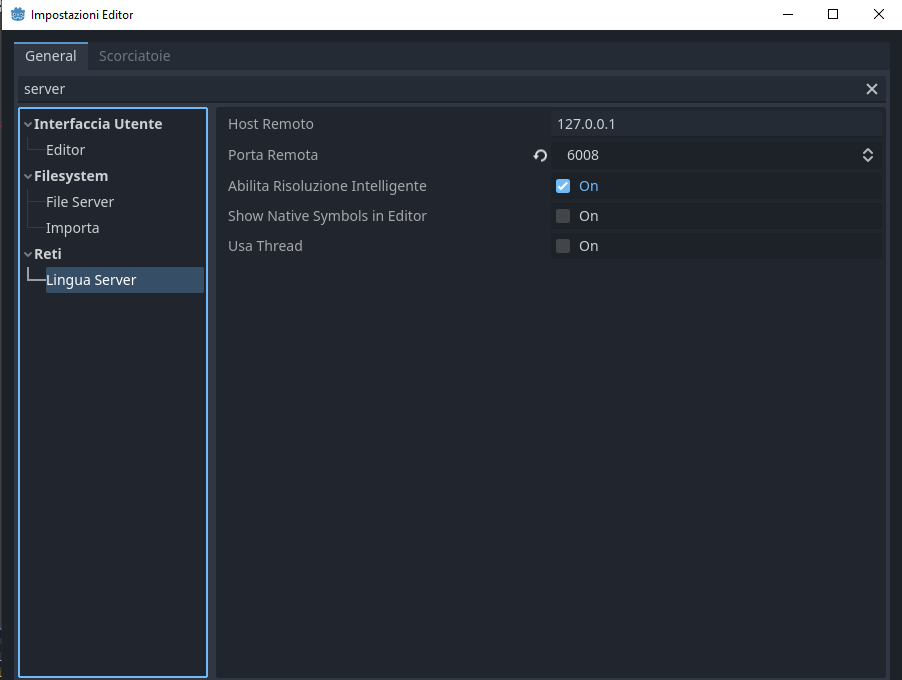
Godot-tools serve per utilizzare il linguaggio proprietario GDScript, cosa che nel progetto non viene fatta data che useremo C#, pertanto potrebbe non essere necessaria l’installazione di questa estensione.

Nel caso voleste installarla comunque potrebbe uscire un messaggio di errore relativo al server GDScript non raggiungibile. Questo accade perché, per gestire funzionalità come il completamento automatico del codice, Godot si collega ad un server. Per prima cosa è necessario che l’editor di Godot sia aperto, ovvero bisogna aprire l’engine e caricare un progetto. Se comunque dovesse dare errore allora sarà necessario riconfigurare il server di Godot in modo che corrisponda ai dati di accesso caricati in VS Code.



Dovremo quindi aprire l’editor e impostare nelle impostazioni dell’editor l’indirizzo ip e porta corretti (quelli presenti nel messaggio di errore su VS Code).

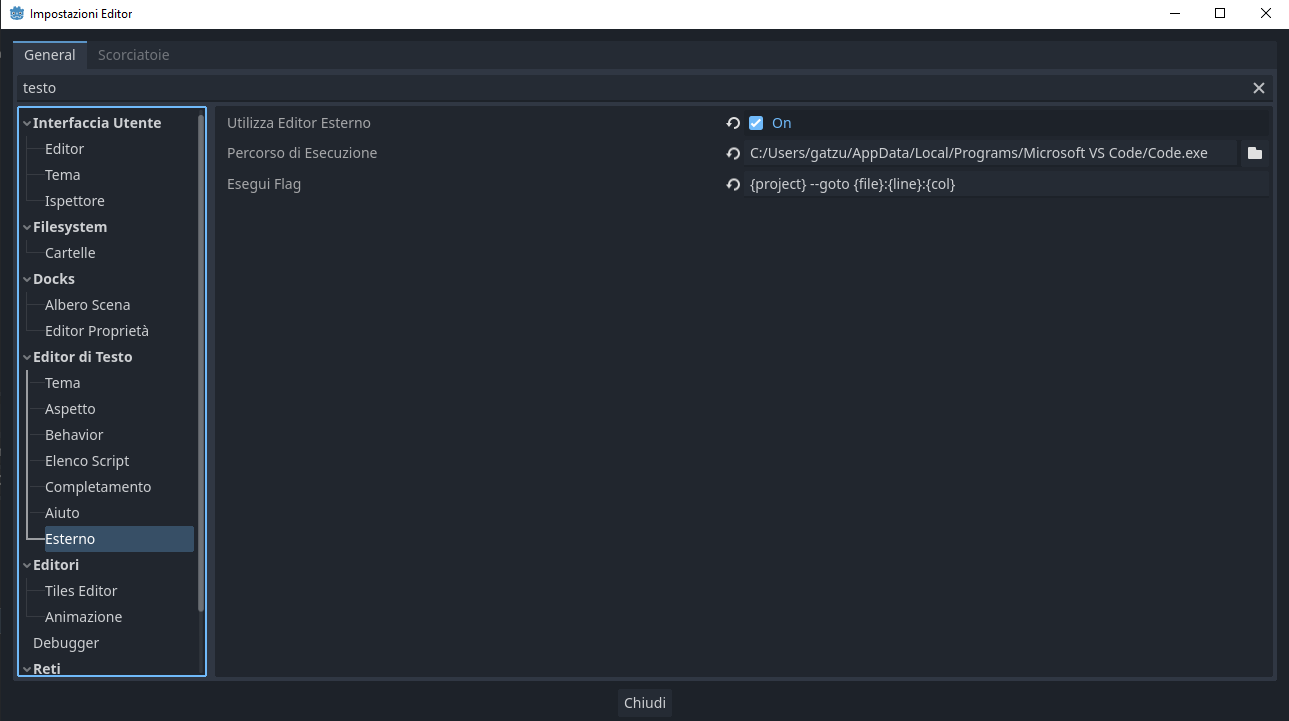




Inseriamo Ip e porta e salviamo. Avremo così configurato Godot-tools.

1. Adesso dobbiamo configurare Godot in modo tale che usi come Ide di default VS Code. Godot infatti dispone di un IDE integrato, tuttavia non fornisce alcuna funzionalità per chi scrive in C# e questo lo rende molto scomodo da usare nel nostro caso.

Per fare questo dovremo aprire nuovamente le impostazioni dell’editor, come fatto nel punto precedente, e cercare “***Esterno***” nelle voci dell’Editor di testo .



A questo punto abilitiamo l’utilizzo dell’editor esterno e inseriamo il *percorso dell’editor*, ovvero il percorso del file .exe di VS Code, solitamente:

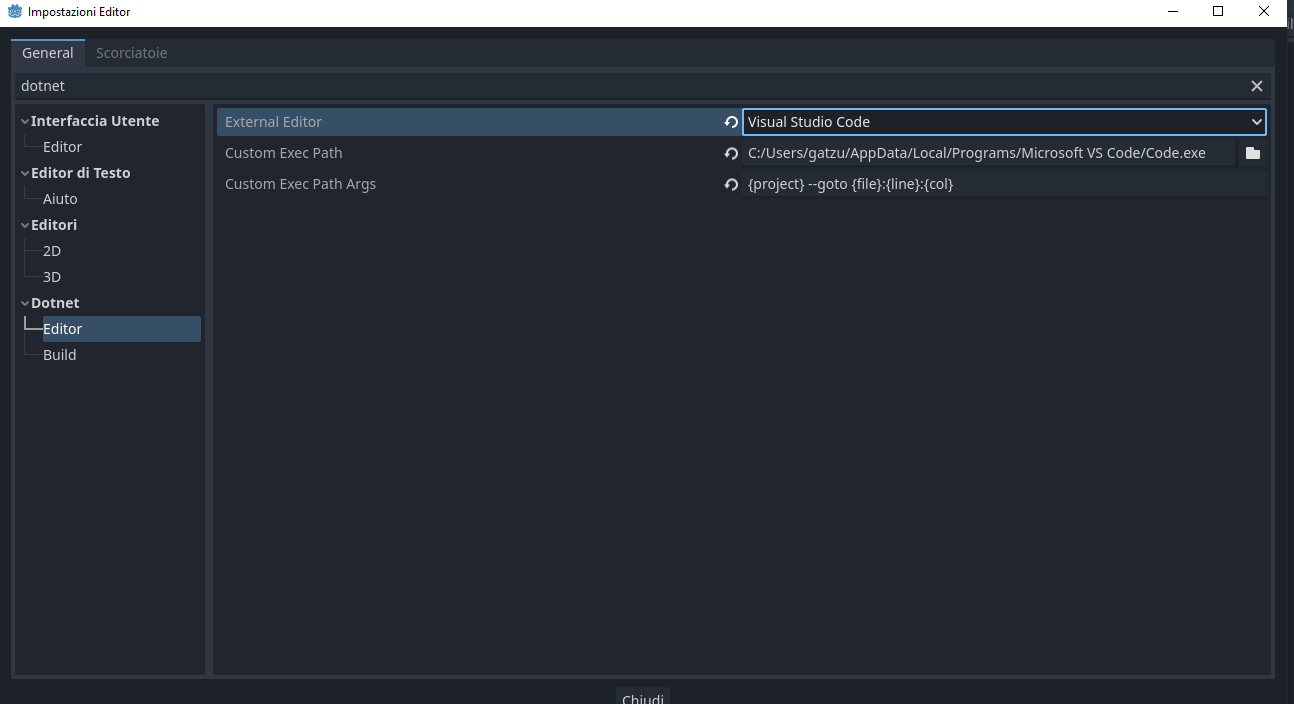
**“C:/Users/<*NomeUtentePC*> /AppData/Local/Programs/Microsoft VS Code/Code.exe”**

Dove al posto di <*NomeUtentePC*> va messo il nome effettivo del vostro sistema.

Alla voce *Esegui Flag* andremo invece ad inserire il comando:

**{project} --goto {file}:{line}:{col}**

Sempre nelle impostazioni dell’editor cerchiamo ora la voce Editor in Dotnet

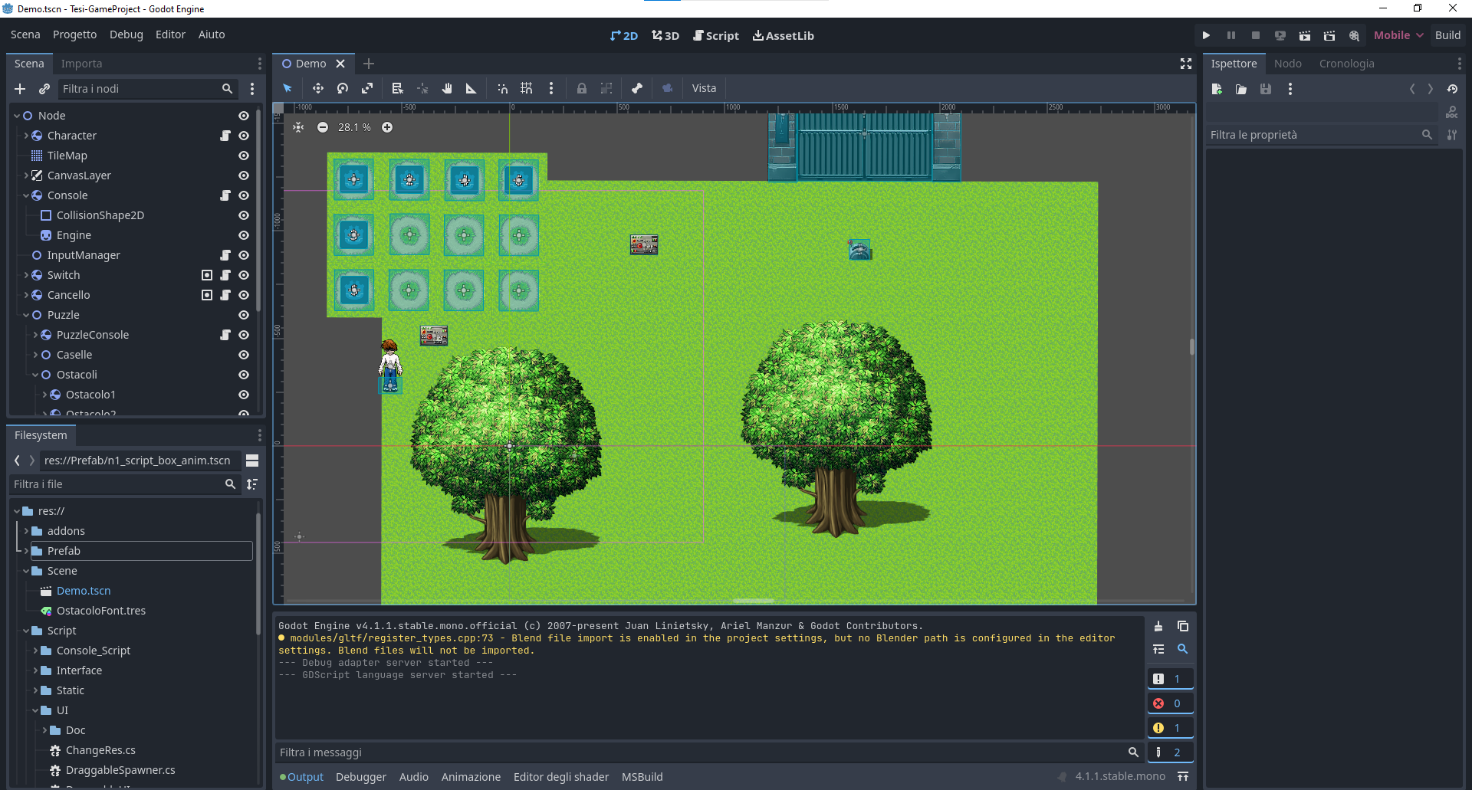


Selezioniamo Visual Studio Code nel menu a tendina, nel *Custom Exec Path* andiamo ad inserire di nuovo il percorso di Code.exe e anche in *Custom Exec Path Args* inseriamo il comando immesso in precedenza.

Perfetto, una volta fatto tutto questo il sistema sarà configurato e sarà possibile lavorare con il progetto senza problemi.

**Introduzione a Godot**

Diamo uno sguardo all’engine.



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**, invece, 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.

**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.

**Script**

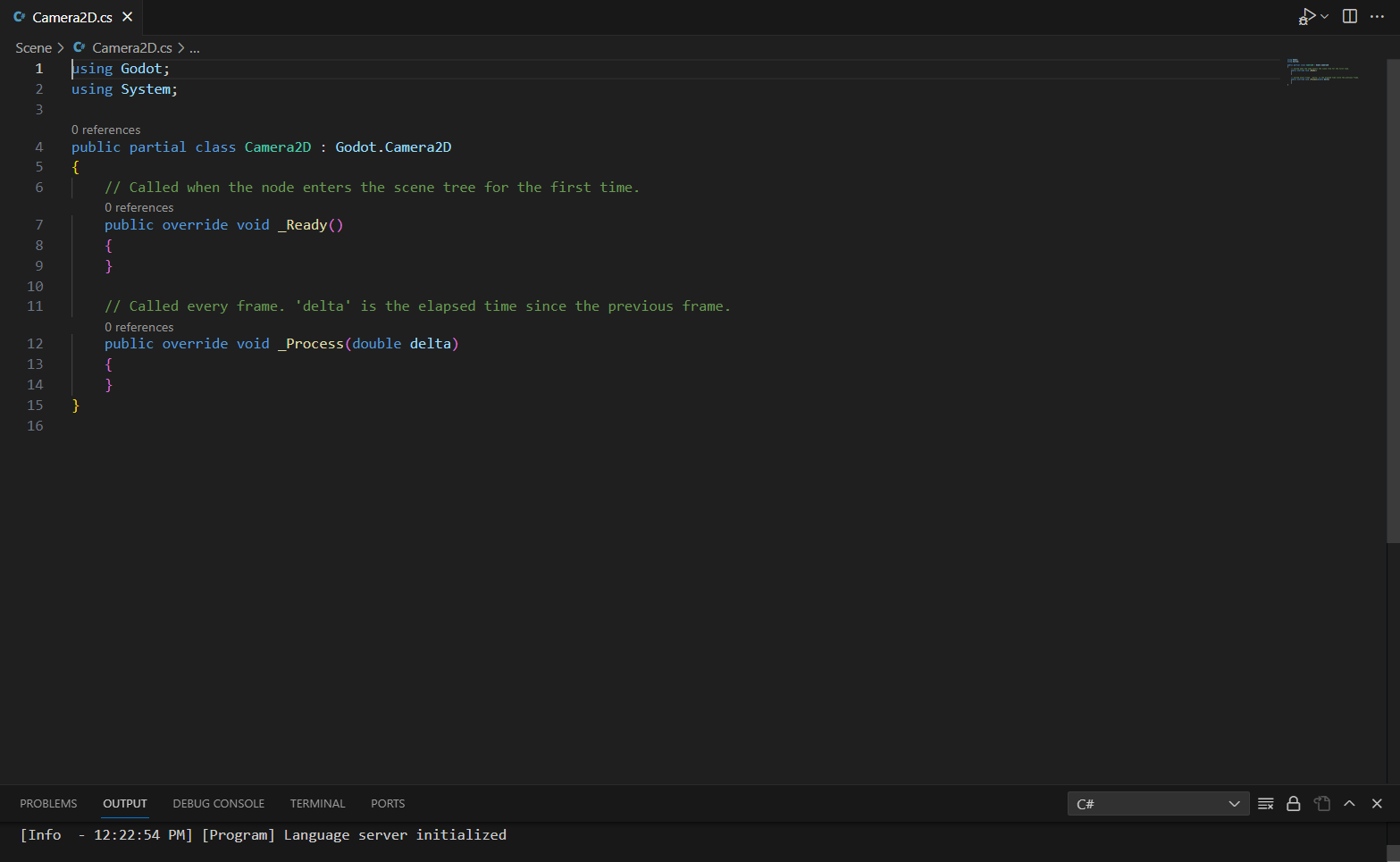
È possibile assegnare uno script a qualsiasi nodo, per farlo basta:

* 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.



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.

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).

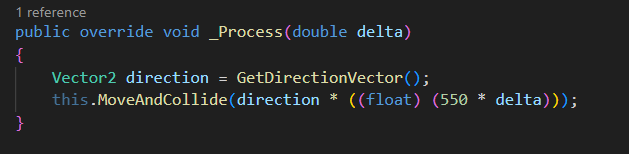
Non è quindi possibile sapere quante volte in un secondo verrà eseguito il codice presente nel \_Process(), pertanto bisogna prestare attenzione al codice che si scrive.

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:



Quindi per muovere il nostro personaggio possiamo assegnare uno script al nostro RigidBody2D che lo estende e applica un vettore al metodo MoveAndCollide(), vettore che sarà definito in base agli input che daremo da tastiera, in modo da farlo muovere in tutte le direzioni possibili.

Per far cambiare i frame dello sprite possiamo scrivere un altro script collegato allo Sprite2D che in base al vettore direzione selezionato applica una animazione adatta. Per fare questo, in questo caso, abbiamo usato le **coroutine**, grazie alle quali possiamo eseguire delle operazioni temporizzate e quindi possiamo cambiare il numero di frame, aspettare un dato tempo t e poi passare al frame successivo. In questo modo creiamo l’animazione.

Proprio per le coroutine è stata usata nel progetto una **libreria di terze parti** per utilizzare delle coroutine più sofisticate, la libreria si chiama **More Effective Godot Coroutines** (MEC) ed è disponibile al seguente link:

<https://github.com/WeaverDev/More-Effective-Godot-Coroutines>

Ultima cosa è possibile referenziare dei nodi o dei tipi specifici, dichiarandoli come proprietà nello script facendoli precedere dal tag [Export], fatto questo basterà aprire l’editor di Godot, cliccare su MSBuild nella parte in basso dell’interfaccia vicino alla scritta Output vista prima e poi cliccare su Build e dopo Rebuild project.

Facendo questo e andando ad eseminare nell’inspector il nodo a cui è assegnato lo script che abbiamo modificato, noteremo che sarà comparsa la voce della proprietà da noi definita. Potremo quindi trascinare il nodo specifico, contente il tipo da refenziare, direttamente sulla nuova voce ottenuta, oppure cliccare su di essa e selezionare la risorsa cercata dall’elenco. In questo modo, quando lo script verrà eseguito, andrà ad effettuare le operazioni inerenti a quella proprietà sul nodo che abbiamo referenziato. Possiamo così facilmente mettere in comunicazione nodi diversi della scena.

Queste sono le basi per poter utilizzare l’editor di Godot. Per capire meglio come funziona, vedere esempi e leggere più nel dettaglio cosa è possibile fare con i nodi e le librerie dell’engine, è possibile visionare la documentazione online, completa sia di esempi scritti in GDScript che C#.

È possibile visionare la documentazione ufficiale scegliendo la versione di interesse al seguente indirizzo:

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

**Documentazione Progetto**

Andiamo a vedere meglio come funzionano alcune parti del progetto in modo da facilitare la scrittura di nuove funzionalità e contenuti, sfruttando i mezzi che sono stati predisposti per rendere il lavoro più semplice.

**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 similare 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;

**Controllo Personaggio e Camera**

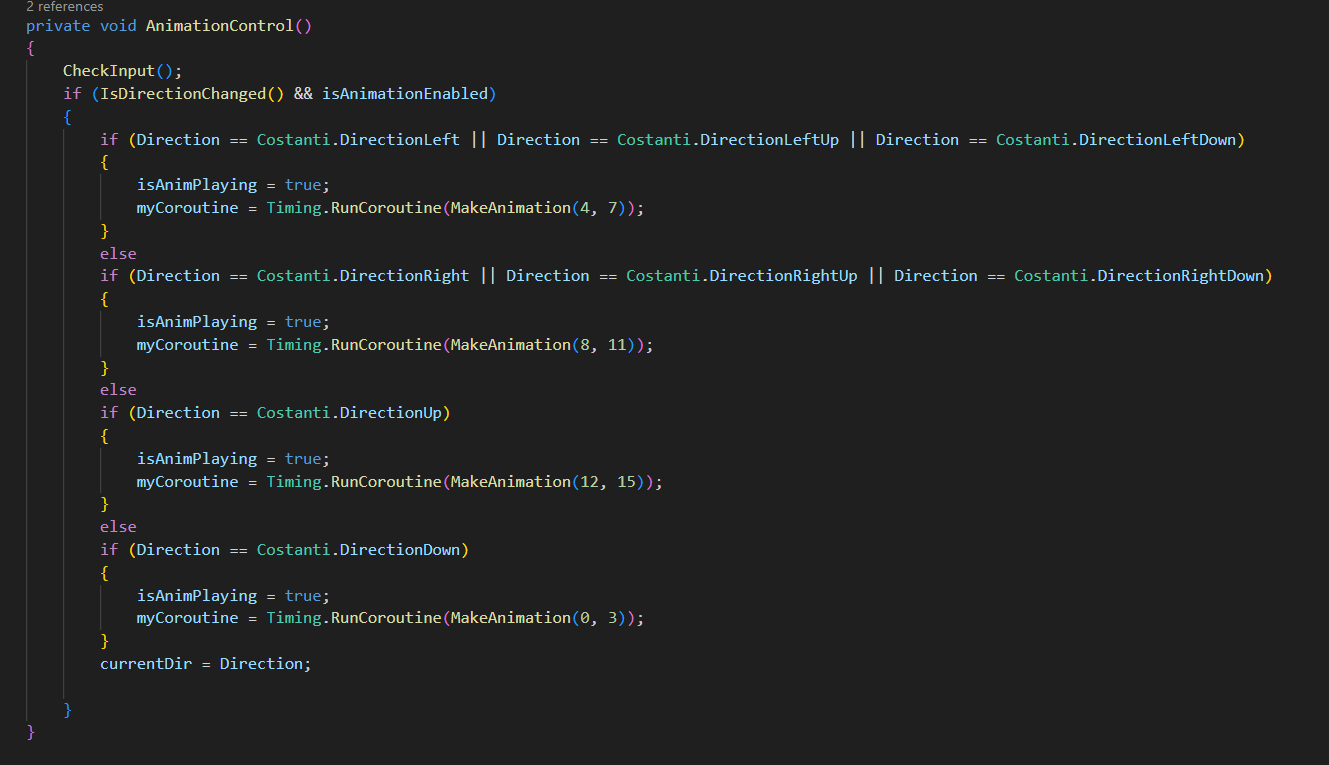
Per gestire il movimento dell’avatar del personaggio ho usato tre diversi nodi:

* **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.

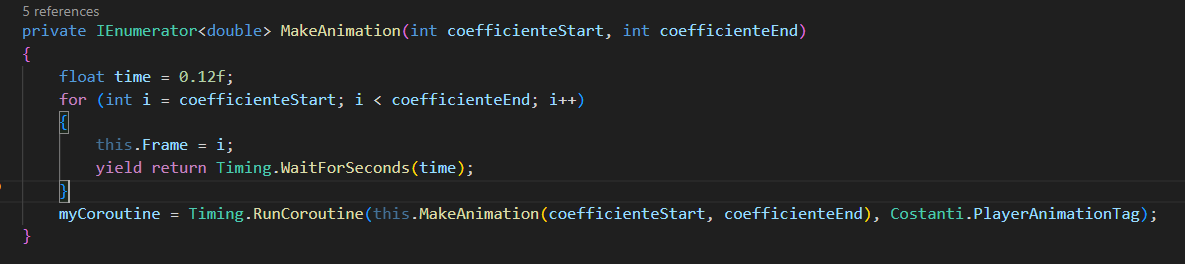
Lo script **CharController** rileva gli input legati al movimento, ovvero i pulsanti “wasd” della tastiera. Premendo uno di questi pulsanti, viene salvata la direzione selezionata in una stringa **Direction**.

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



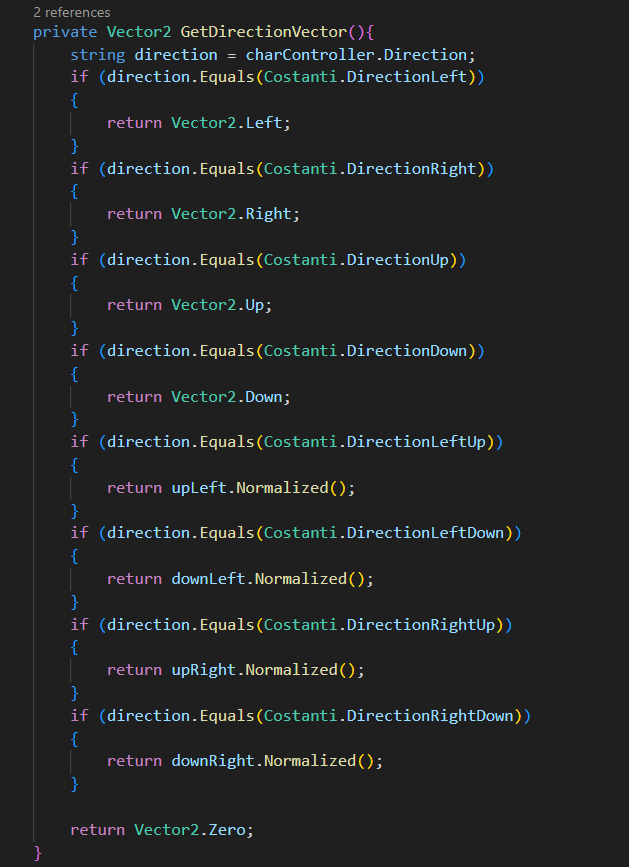
L’animazione è realizzata attraverso le coroutine:

* 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;



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.



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.

**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 contente 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 similare 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 connesione 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.

**Modalità Esercizio**

La modalità esercizio è uno strumento rivolto più agli insegnanti che possono utilizzarla come eserciziario per gli studenti. All’interno dell’applicazione sono stati inseriti come argomenti possibili i seguenti:

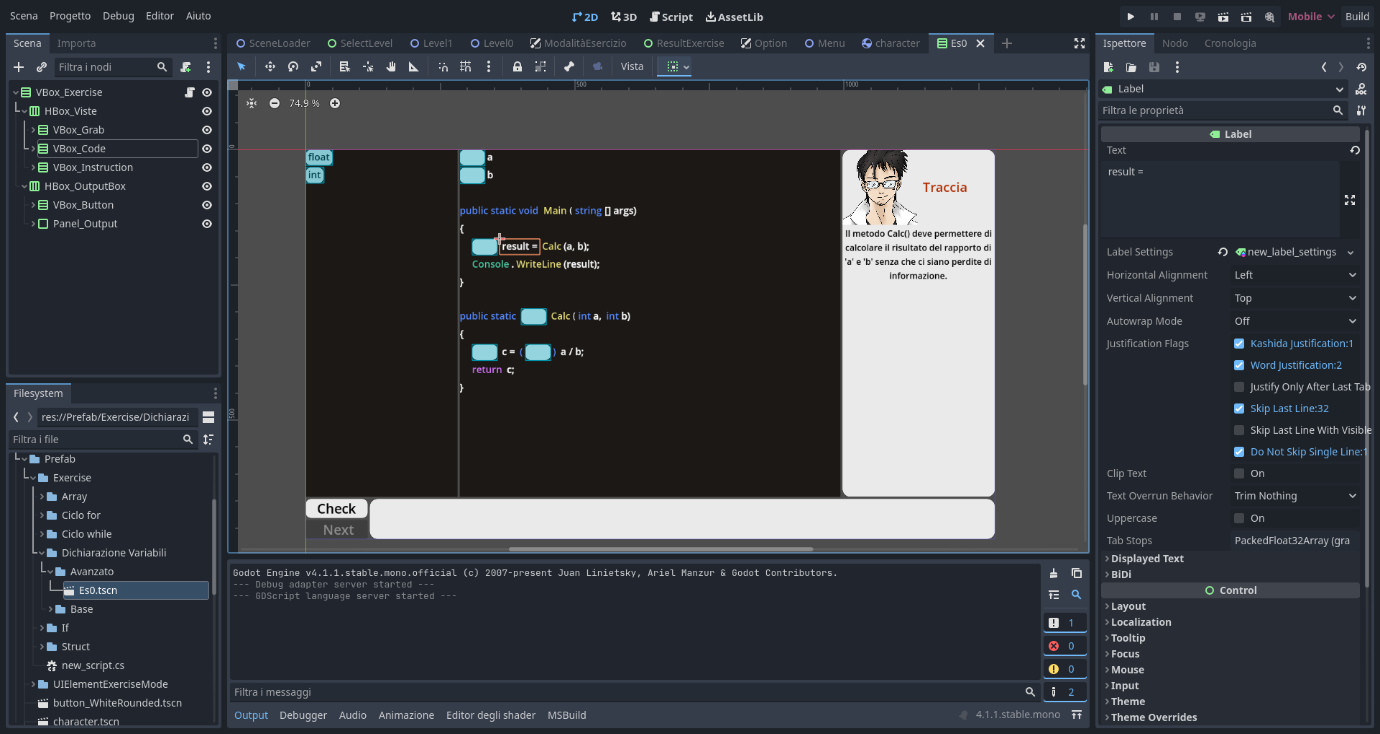
* Dichiarazione Variabili;
* If;
* Ciclo for;
* Ciclo while;
* Array;
* Struct;

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.

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



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, pertento 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.

Quando si instazia un prefab, **se si vuole modificarlo senza che le modifiche si ripercuotano su tutte le altre sue istanze**, basterà cliccare con il tasto destro del mouse sul nodo del prefab nella scena e cliccare su **Rendi Locale**.

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.

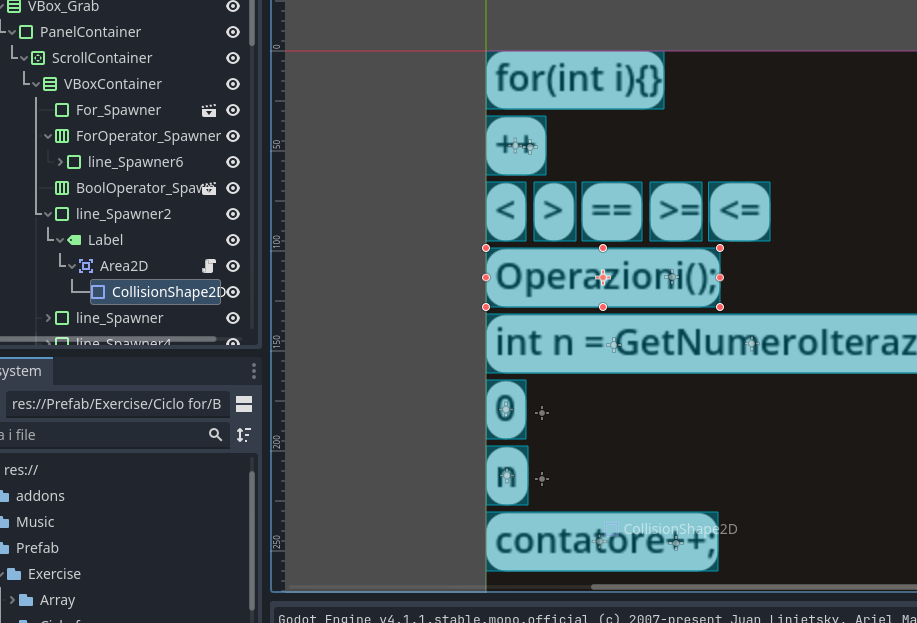
Gli spawner hanno un funzionamento semplice:

* 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.

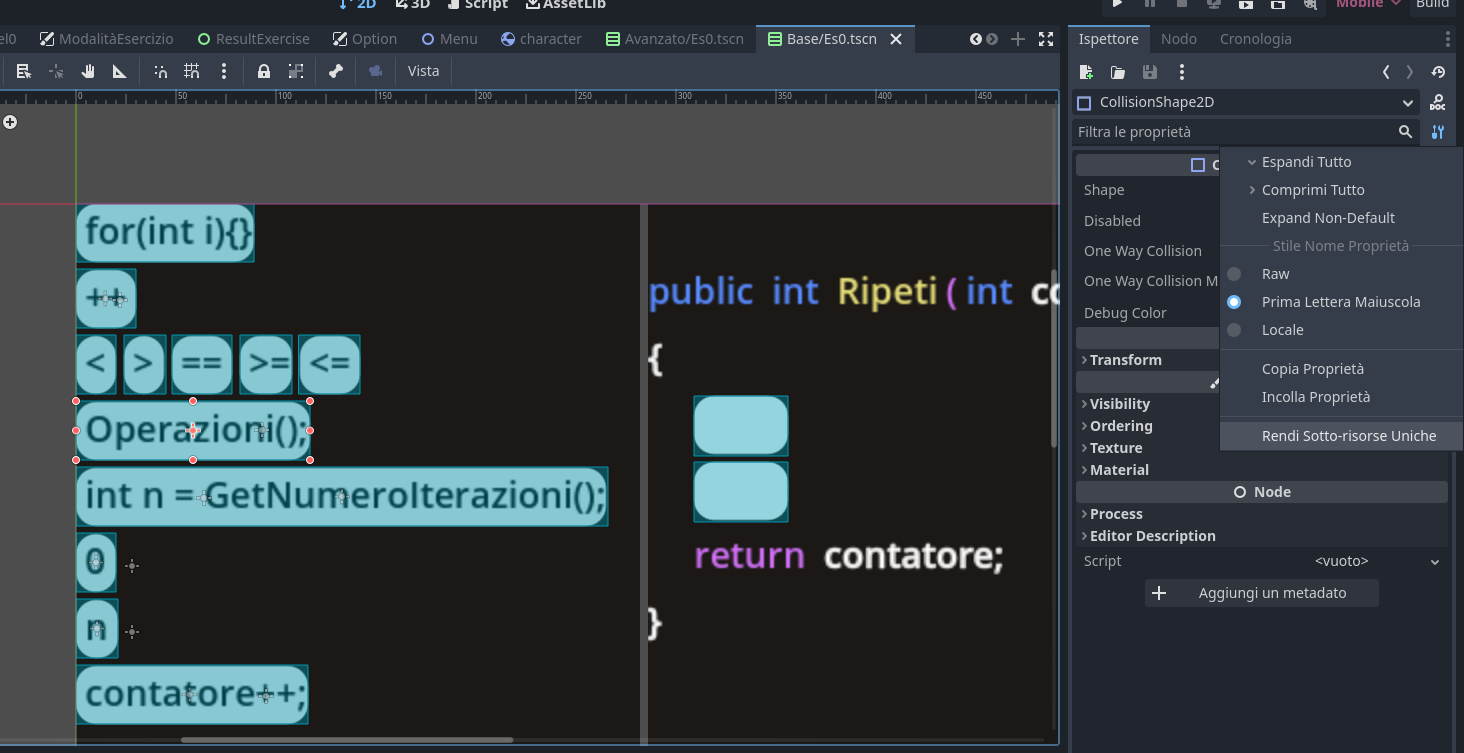
**In questo caso bisognerà fare attenzione**, infatti il riquadro blu che racchiude lo Spawner è la forma dell’Area2D che permette di rilevare quando il DraggableSpawner e l’InteractiveBox si sovrappongono. Andando a modificare il contenuto della label, la lunghezza del box cambierà e quindi bisognerà aggiornare manualmente la dimensione del CollisionShape.

Per farlo basterà selezionare il riquadro blu e spostare i vertici rossi in modo da ridimensionare le dimensioni del box.



**Prima di fare questo però bisogna avere un’ultima accortezza**: la forma del collisionShape è data da una sua propietà, gestibile attraverso l’inspector, chiamata **Shape**. Questa proprietà altro non è che **un riferimento ad un oggetto** che contiene le informazioni sulla forma. Pertanto nel caso avessimo istanziato lo spawner attraverso un prefab oppure duplicando un nodo preesistente, il riferimento sarà lo stesso in tutte le istanze. Quindi **modificando la forma del CollisionShape verranno modificati tutti i CollisionShape con lo stesso riferimento**.

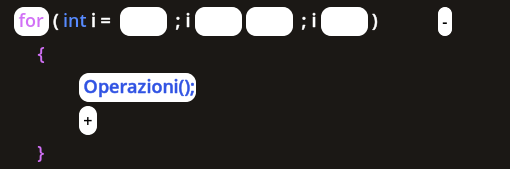
Per evitare questo problema basterà cliccare sull’icona visibile in figura dall’inspector e selezionare “Rendi Sotto-risorse Uniche”.



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.



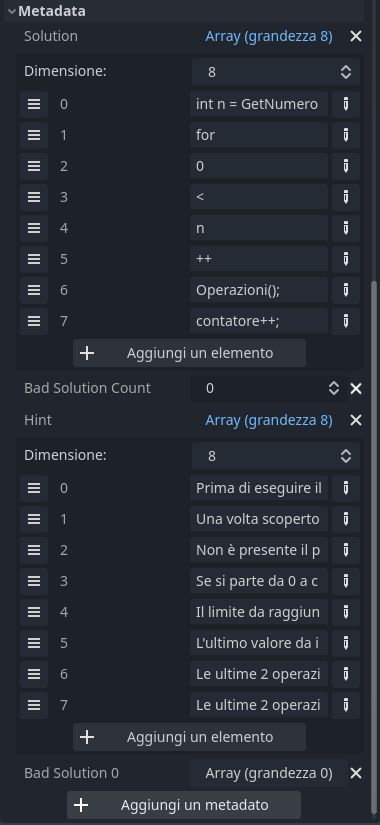
Questo è un esempio del for. Come è possibile vedere dall’immagine sono presenti varie InteractiveBox per poter definire tutti i parametri del for, mentre all’interno delle parentesi graffe non solo è presente una InteractiveBox, ma una volta riempita è possibile cliccare sul pulsante “+” per aggiungere 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:



* **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).

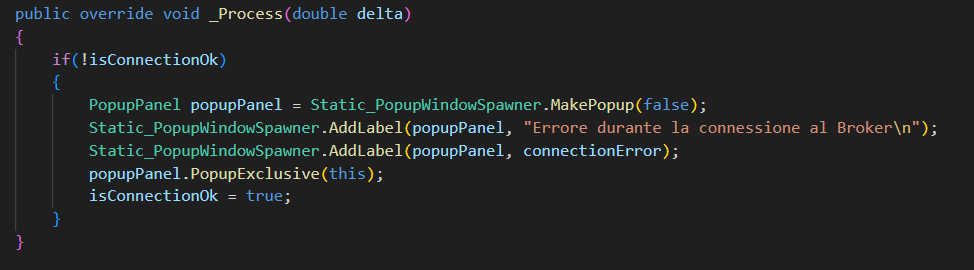


In questo modo è possibile creare nuovi esercizi da aggiungere all’applicazione senza dover scrivere alcun tipo di codice, andando semplicemente a sfruttare l’interfaccia grafica dell’editor.

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.

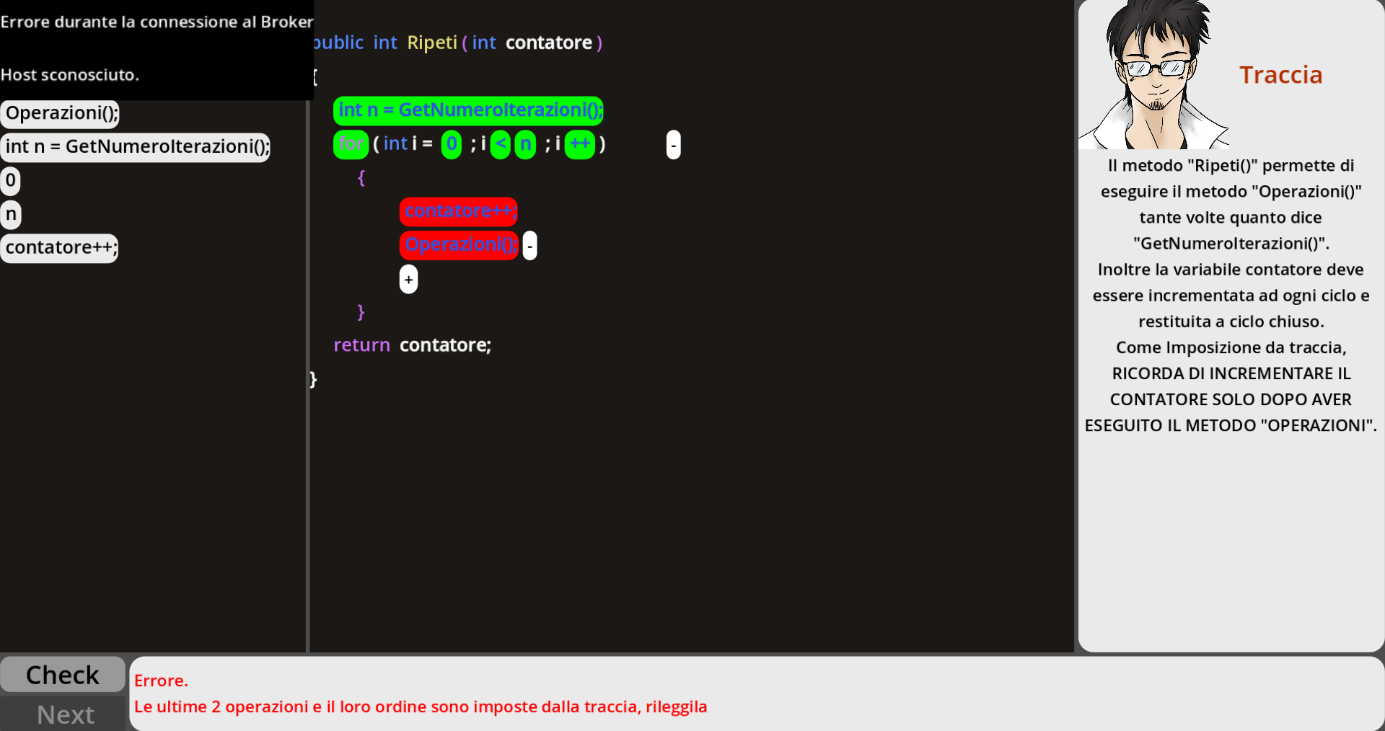
L’ultima cosa da dire riguardo la modalità esercizio è che la classe Exercise\_Control implementa anche l’interfaccia BrokerChatter.

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.

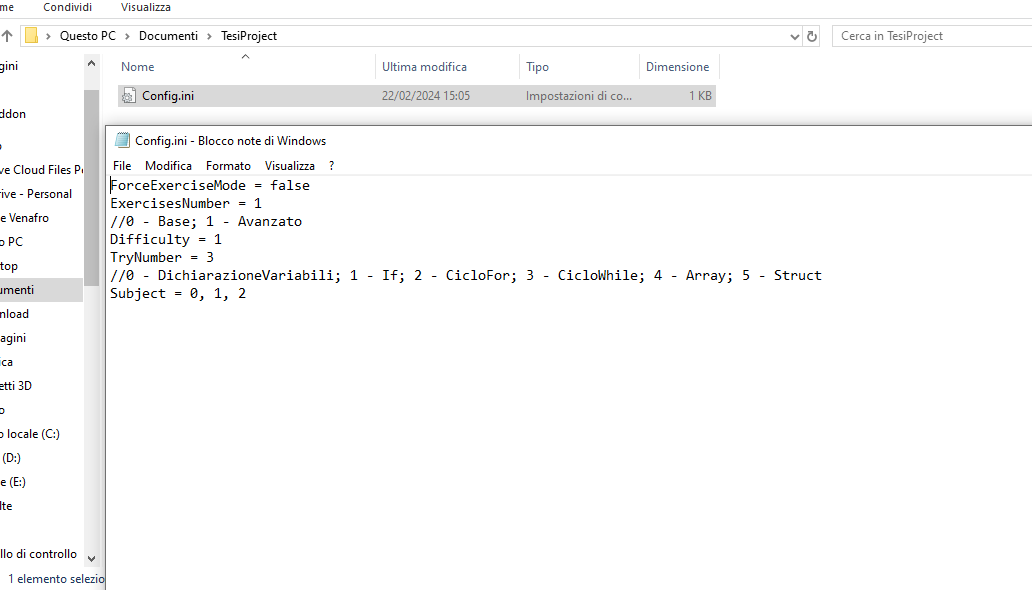


Per farlo ho scritto questo codice nel \_Process(), infatti in caso di errore la classe statica andrà a rendere false il booleano isConnectionOk e scriverà all’interna della stringa connectionError il messaggio di errore riscontrato.

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.



È possibile inoltre **configurare la modalità esercizio attraverso un file esterno**. Infatti al primo avvio del gioco verrà creata nella **cartella documenti** una cartella chiamata “**ScriptAdventure**” con all’interno un file ini di configurazione. Andando a modificare questo file sarà possibile avviare il gioco direttamente nella modalità esercizio configurata come scritto nel file.



**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.

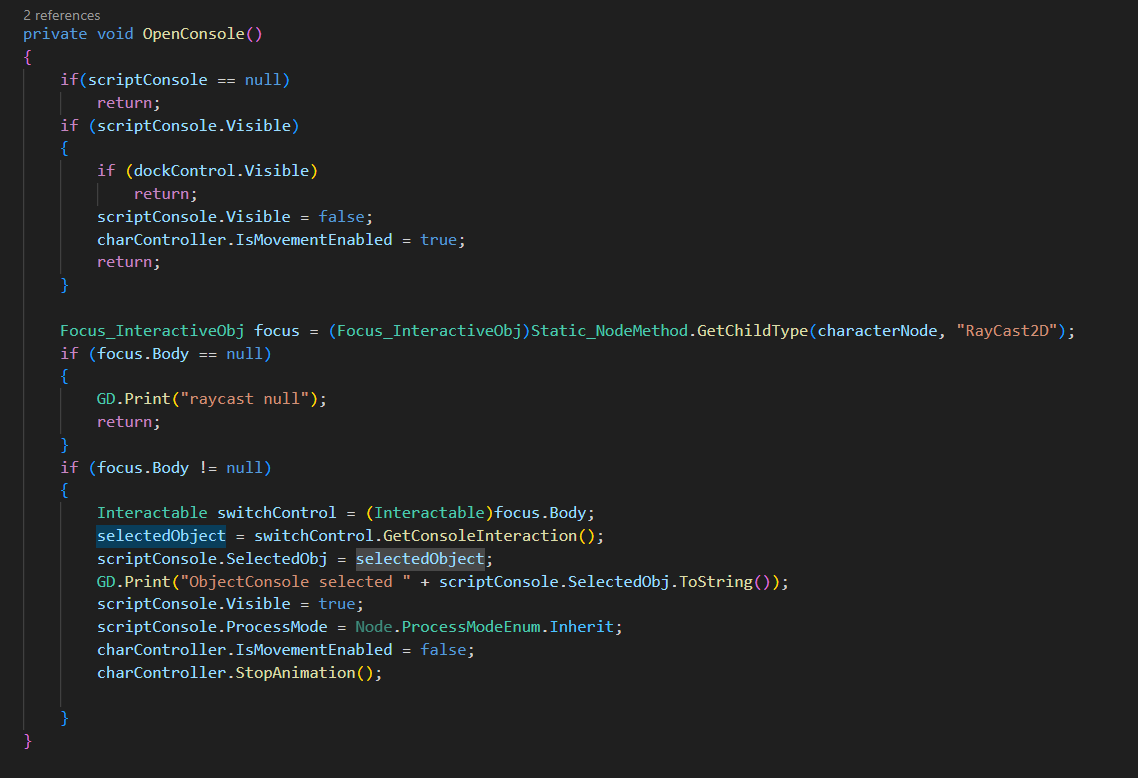


Utilizzando questo nodo è stato quindi possibile rilevare gli oggetti selezionati sui quali eseguire le operazioni della console. Al raycast ho quindi associato uno script che ho chiamato **Focus\_InteractiveObj** e gli ho assegnato un riferimento a CharController in modo da ruotare il vettore in base 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, la quale ha il compito di assegnare determinate azioni a determinati input della tastiera.



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 contente 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 contente tutte le proprietà dell’oggetto con cui è possibile interagire attraverso la ScriptConsole e la Doc;
* **methodList**: una lista contente 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:

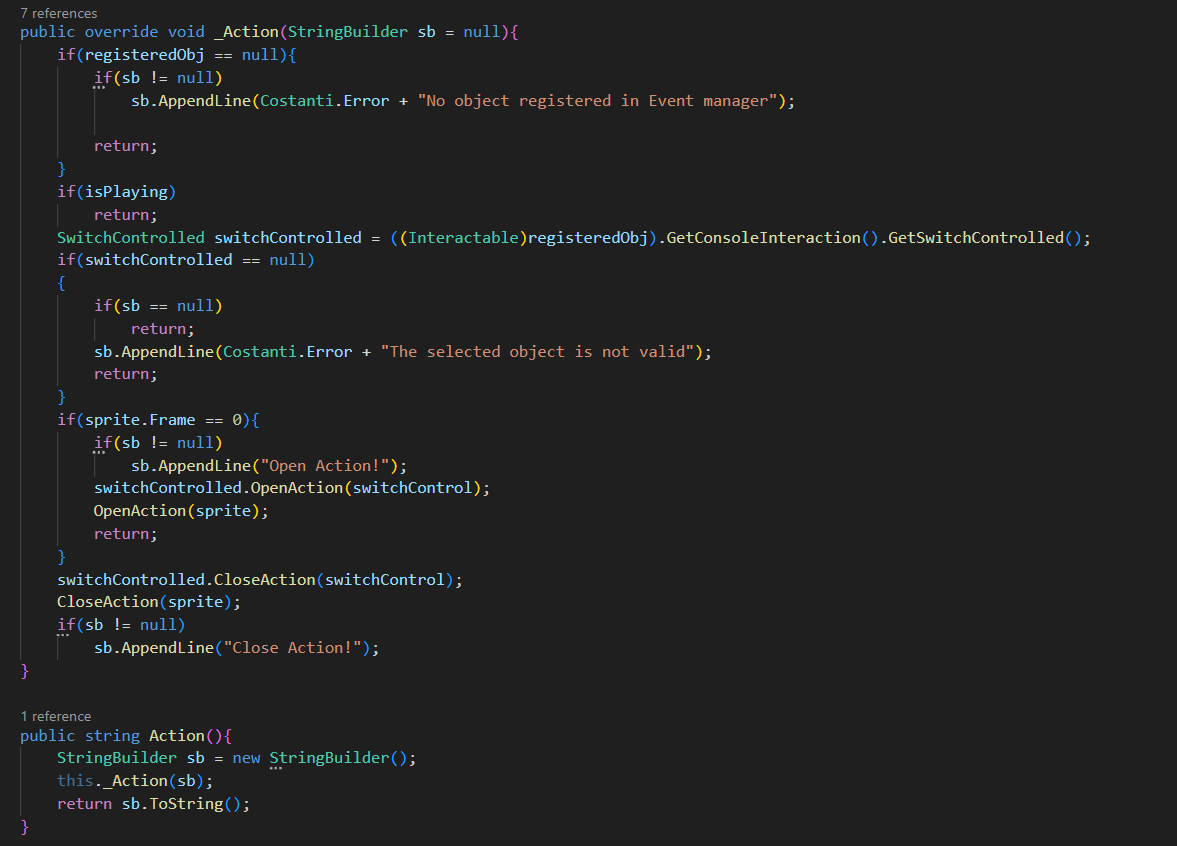


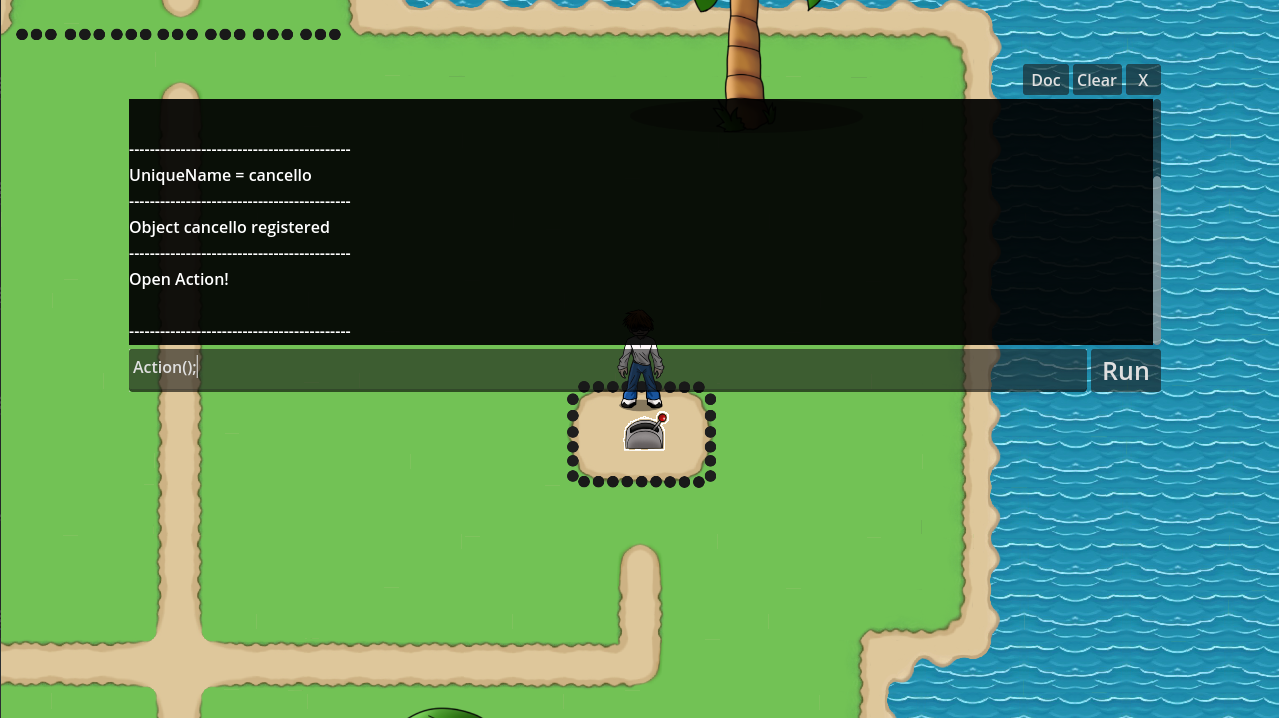
methodsList è invece vuota, in quanto questo oggetto specifico non avrà metodi richiamabili attraverso la console. Infatti ConsoleInt\_Door implementa anche un’altra interfaccia, ovvero **SwitchControlled**. Questo perché le porte devono essere aperte interagendo con altri oggetti come leve e pulsanti. 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;

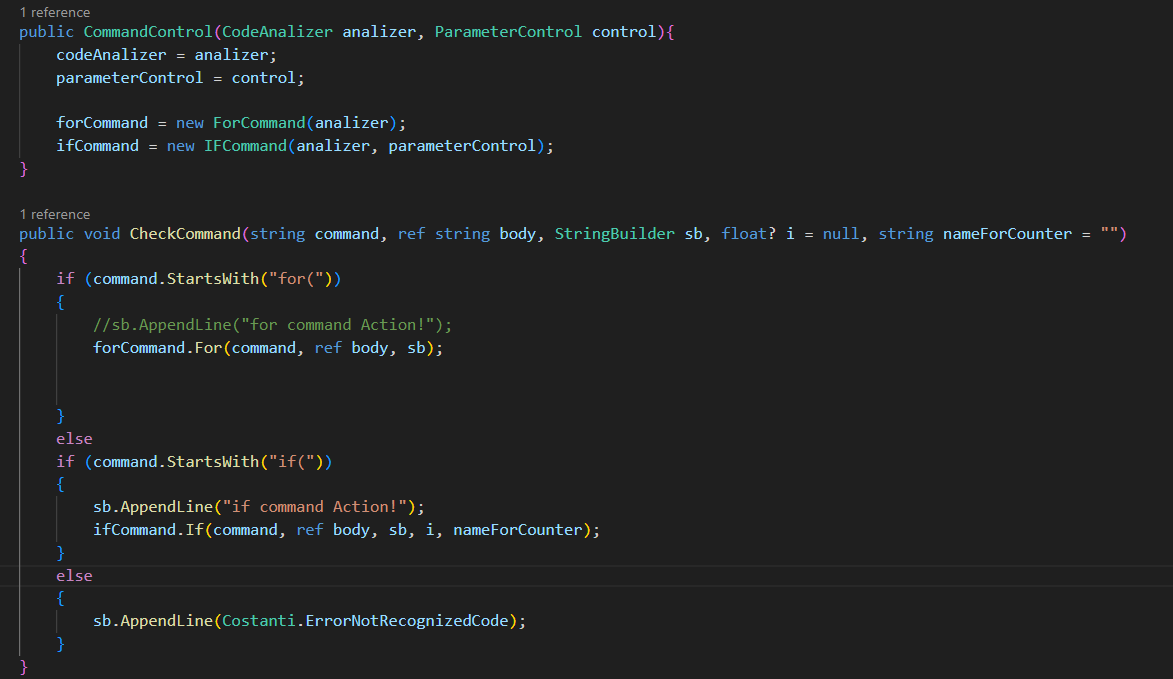
Questa è una differenza necessaria, come spiegherò successivamente, la scriptConsole utilizza uno StringBuilder durante le sue operazioni per salvare al suo interno tutti i messaggi, di errore e non, che dovrà far visualizzare al giocatore per informarlo sullo stato delle operazioni eseguite. Tuttavia 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 defininire 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 riporlarlo a schermo.





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

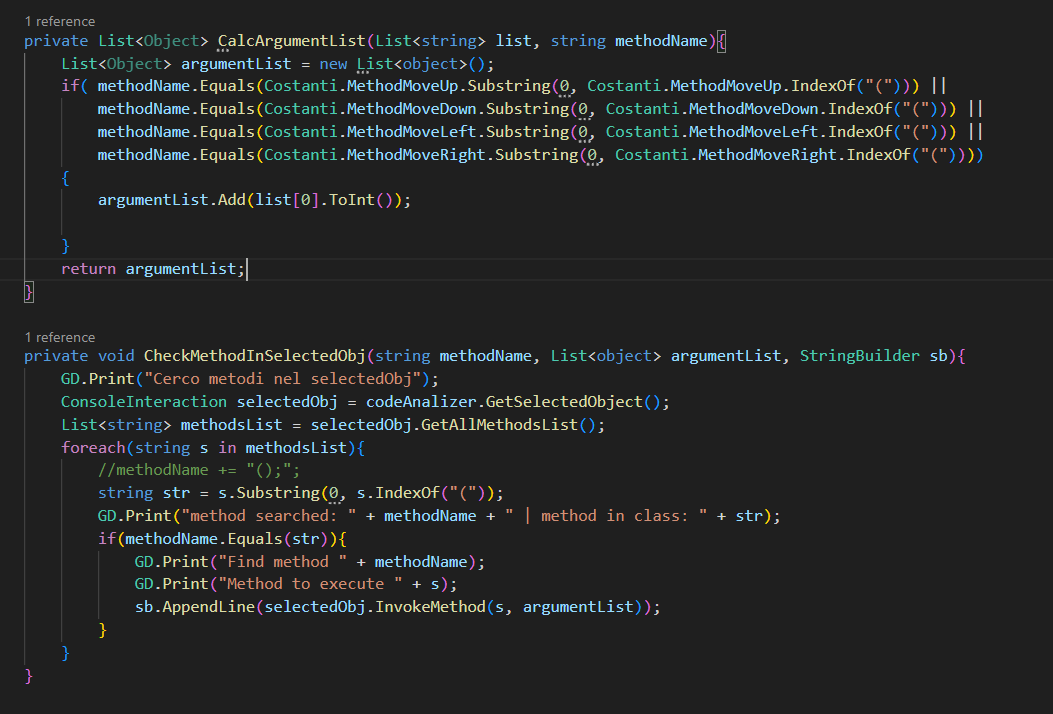
CodeAnalizer è in grado di riconoscere:

* **Comandi**: ovvero le funzionalità interne del codice, come ad esempio il for oppure l’if. Questi comandi sono caratterizzati da una stringa iniziale, delle parentesi tonde al cui interno vengono specificate le condizioni del comando e infine delle parentesi graffe contenenti il codice da eseguire quando le condizioni sono soddisfatte. In particolare CodeAnalizer individua il nome del comando, le condizione 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 comandi della console. Se CommandControl riceve il nome di un comando valido, allora eseguirà il metodo associato dell’oggetto adeguato;
* **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 contro 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 CodeAnalizer vengono salvati all’interno di una lista di stringhe, in modo da 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.

Per fare questo 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.



**Quindi facendo un riepilogo di come inserire un metodo**:

1. Implementiamo **ConsoleInteraction** in una nuova classe;
2. Creiamo una classe che implementa l’**interfaccia Interactable** contente una proprietà ConsoleInteraction definita con il tipo creato al passo precedente;
3. All’interno della classe scritta nel punto 1 creiamo un metodo di Initialize da richiamare nel costruttore in cui riempiamo i campi di **MethodList** e **propertiesList**;
4. I nomi dei metodi e delle proprietà li salviamo nella classe Costanti.
5. Scriviamo i metodi sempre nella stessa classe come **metodi di tipo stringa** per restituire le info sull’esecuzione del metodo.
6. Nella classe **MethodControl** definiamo il tipo dei parametri del metodo;
7. Scriviamo nella classe costanti le descrizioni di proprietà e metodi e modifichiamo la classe **DocDescReader** di conseguenza.

**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** contente 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.

